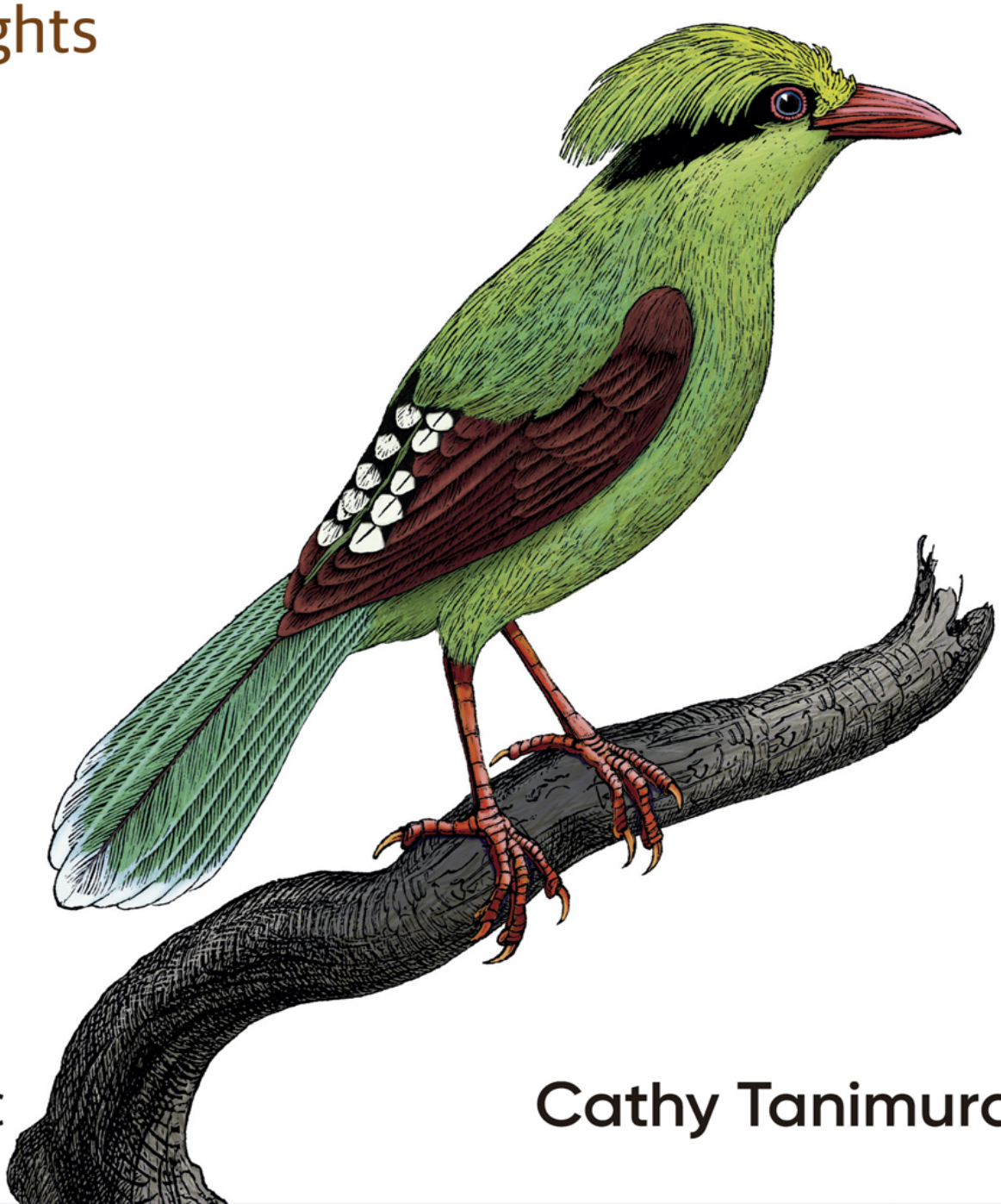
O'REILLY®

SQL para Análise de Dados

Técnicas avançadas para transformar dados em insights



novatec

Cathy Tanimura

SQL
para Análise de Dados
Técnicas avançadas para transformar dados em insights

Cathy Tanimura

O'REILLY®
Novatec

Authorized Portuguese translation of the English edition of *SQL for Data Analysis* ISBN 9781492088783 © 2021 Cathi Tanimura. This translation is published and sold by permission of O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

Tradução em português autorizada da edição em inglês da obra *SQL for Data Analysis* ISBN 9781492088783 © 2021 Cathi Tanimura. Esta tradução é publicada e vendida com a permissão da O'Reilly Media, Inc., detentora de todos os direitos para publicação e venda desta obra.

© Novatec Editora Ltda. [2022].

Editor: Rubens Prates
Tradução: Aldir Coelho Corrêa da Silva
Revisão gramatical: Tássia Carvalho

ISBN do impresso: 978-65-86057-75-1

ISBN do ebook: 978-65-86057-92-8

Histórico de impressões:

Julho/2022 Primeira edição

Novatec Editora Ltda.
Rua Luís Antônio dos Santos 110
02460-000 – São Paulo, SP – Brasil
Tel.: +55 11 2959-6529
Email: *novatec@novatec.com.br*
Site: *https://novatec.com.br*
Twitter: *twitter.com/novateceditora*
Facebook: *facebook.com/novatec*
LinkedIn: *linkedin.com/in/novatec*
GRA20220712

Sumário

Prefácio

capítulo 1 Análise com SQL

O que é análise de dados?
Por que SQL?
  O que é SQL?
  Benefícios do SQL
  SQL versus R ou Python
  SQL como parte do fluxo de trabalho de análise de dados
Tipos de bancos de dados e como trabalhar com eles
  Bancos de dados row-store
  Bancos de dados column-store
  Outros tipos de infraestrutura de dados
Conclusão

capítulo 2 Preparando os dados para análise

Tipos de dados
  Tipos de dados de banco de dados
  Estruturados versus não estruturados
  Dados quantitativos versus qualitativos
  Dados primários, secundários e de terceiros
  Dados esparsos
Estrutura da consulta SQL
Criação de perfis: distribuições
  Histogramas e frequências
  Discretização (Binning)
  Funções n-tiles
Criação de perfis: qualidade dos dados
  Detectando duplicidades
  Desduplicação com GROUP BY e DISTINCT
Preparação: limpeza dos dados
  Limpando dados com transformações CASE

- Conversões de tipos e casting
- Lidando com nulos: funções coalesce, nullif, nvl
- Dados ausentes

Preparação: modelando dados
- Para qual saída: BI, visualização, estatística, ML
- Pivotando com instruções CASE
- Despivotando com instruções UNION
- Funções pivot e unpivot

Conclusão

capítulo 3 Análise de séries temporais

Manipulações de data, data/hora e hora
- Conversões de fuso horário
- Conversões de formato de data e timestamp
- Matemática de datas
- Matemática de horas
- Unindo dados de diferentes origens

Conjunto de dados de vendas do varejo

Encontrando tendências nos dados
- Tendências simples
- Comparando componentes
- Percentual em cálculos de totais
- Indexação para a detecção de alterações percentuais com o passar do tempo

Janelas de tempo contínuas
- Calculando janelas de tempo contínuas
- Janelas de tempo contínuas com dados esparsos
- Calculando valores cumulativos

Analisando com sazonalidade
- Comparações período a período: YoY e MoM
- Comparações período a período: mesmo mês versus ano passado
- Fazendo a comparação com vários períodos anteriores

Conclusão

capítulo 4 Análise de Coorte

Coortes: uma estrutura de análise útil

Conjunto de dados de legisladores

Retenção
  SQL para uma curva de retenção básica
  Ajustando a série temporal para aumentar a exatidão da retenção
  Coortes derivadas da própria série temporal
  Definindo a coorte a partir de uma tabela separada
  Lidando com coortes esparsas
  Definindo coortes a partir de datas diferentes da primeira data
Análises de coorte relacionadas
  Sobrevivência
  Retorno, ou comportamento de compra repetida
  Cálculos cumulativos
Análise transversal, considerada com base em uma coorte
Conclusão

capítulo 5 Análise de texto

Por que fazer análise de texto com SQL?
  O que é análise de texto?
  Por que o SQL é uma boa opção para a análise de texto
  Quando o SQL não é uma boa opção
Conjunto de dados de avistamentos de OVNIs
Características do texto
Parsing do texto
Transformações de texto
Encontrando elementos dentro de blocos de texto maiores
  Correspondências com curingas: LIKE, ILIKE
  Correspondências exatas: IN, NOT IN
  Expressões regulares
Construindo e remodelando o texto
  Concatenação
  Remodelando o texto
Conclusão

capítulo 6 Detecção de anomalias

Recursos e limites do SQL para a detecção de anomalias
Conjunto de dados
Detectando valores discrepantes
  Ordenando para encontrar anomalias

Calculando percentis e desvios-padrão para encontrar anomalias
Representação gráfica para a busca de anomalias visualmente
Tipos de anomalias
Valores anômalos
Contagens ou frequências anômalas
Anomalias pela ausência de dados
Manipulando anomalias
Investigação
Remoção
Substituição por valores alternativos
Reescalonamento
Conclusão

capítulo 7 Análise experimental

Vantagens e limites da análise experimental com SQL
Conjunto de dados
Tipos de experimentos
Experimentos com resultados binários: o teste qui-quadrado
Experimentos com resultados contínuos: o teste t
Desafios dos experimentos e opções para o resgate de experimentos que falharam
Atribuição de variantes
Valores discrepantes (Outliers)
Time Boxing
Experimentos com exposição repetida
Se não for possível executar experimentos controlados: análises alternativas
Pré/pós-análise
Análise experimental natural
Análise de populações com base em um limite
Conclusão

capítulo 8 Criando conjuntos de dados complexos para análise

Quando usar SQL para conjuntos de dados complexos
As vantagens de usar SQL
Quando a lógica deve ser construída em ETL
Quando inserir a lógica em outras ferramentas

Organização do código
  Comentários
  Capitalização, indentação, parênteses e outros truques de formatação
  Armazenando o código
Organizando a computação
  Entendendo a ordem de avaliação de cláusulas SQL
  Subconsultas
  Tabelas temporárias
  Expressões de tabela comuns
  grouping sets
Gerenciando o tamanho do conjunto de dados e considerações sobre privacidade
  Amostragem com %, mod
  Reduzindo a dimensionalidade
  PII e privacidade dos dados
Conclusão

capítulo 9 Conclusão

Análise de funil
Desistência, inatividade e outras definições de afastamento
Análise de cesta de compras
Recursos
  Livros e blogs
  Conjuntos de dados
Considerações finais

Prefácio

Nos últimos 20 anos, passei várias horas de trabalho manipulando dados com SQL. Em grande parte desses anos, trabalhei em empresas de tecnologia que faziam parte de um amplo espectro de indústrias de produtos para consumidores e business-to-business. Nessa época, os volumes de dados cresceram drasticamente, e a tecnologia que eu precisava usar melhorou com rapidez. Os bancos de dados são mais rápidos do que jamais foram, e as ferramentas de relatórios e visualização usadas para demonstrar o significado dos dados são muito mais poderosas. Algo que permaneceu consideravelmente constante, entretanto, é o fato de o SQL ser a parte essencial de minha caixa de ferramentas.

Lembro-me de quando aprendi SQL. Comecei minha carreira em finanças, área dominada pelas planilhas, e me especializei em escrever fórmulas e memorizar todos os atalhos do teclado. Um dia deixei aflorar meu lado nerd e pressionei Ctrl e Alt para cada tecla de meu teclado só para ver o que ocorreria (e depois criei um resumo de referência para meus colegas). Fiz isso em parte por diversão e também por sobrevivência: quanto mais rápido eu fosse com minhas planilhas, mais probabilidades teria de terminar meu trabalho antes da meia-noite para poder ir para casa e dormir. Minha especialização em planilhas me levou ao meu próximo emprego, uma startup na qual fui apresentada aos bancos de dados e ao SQL.

Parte de minha função envolvia organizar dados de inventário em planilhas, e devido à escala inicial da internet, às vezes os conjuntos de dados tinham dezenas de milhares de linhas. Esse era o "big data" da época, pelo menos para mim. Adquiri o hábito de tomar café ou almoçar enquanto a CPU de meu computador estava ocupada fazendo sua mágica com a função `vlookup`. Um dia meu gerente entrou de férias e me pediu para cuidar do data warehouse[1] que ele tinha construído em seu laptop usando o Access. A atualização dos dados envolvia uma série de etapas: executar consultas em um portal, carregar os arquivos csv resultantes no

banco de dados e atualizar os relatórios das planilhas. Após o primeiro carregamento bem-sucedido, comecei a pesquisar, tentando entender como a operação funcionava, e a incomodar os engenheiros para que me mostrassem como poderia modificar as consultas SQL.

Viciei-me, e, mesmo quando achava que poderia dar uma guinada na minha carreira, acabava voltando aos dados. Manipular dados, dar respostas a perguntas, ajudar meus colegas a trabalhar melhor e de maneira mais inteligente, e conhecer os negócios e o mundo por meio de conjuntos de dados nunca deixou de ser divertido e empolgante.

Quando comecei a trabalhar com SQL, não havia muitos recursos de aprendizagem. Adquiri um livro sobre sintaxe básica, consegui lê-lo em uma noite, e a partir daí aprendi em grande parte por tentativa e erro. Na época em que estava aprendendo, consultei bancos de dados de produção diretamente e mais de uma vez deixei o site inativo devido ao meu SQL excessivamente ousado (ou provavelmente apenas mal escrito). Felizmente, minhas habilidades melhoraram, e com o passar dos anos aprendi a trabalhar tendo como ponto de partida os dados das tabelas, e invertendo também essa lógica trabalhando a partir do resultado desejado, para resolver desafios técnicos e lógicos, assim como enigmas, e escrever consultas que retornassem os dados certos. Acabei projetando e construindo data warehouses para coletar dados de diferentes fontes e evitar deixar inativos bancos de dados de produção críticos. Aprendi muito sobre como e quando agregar dados antes de escrever a consulta SQL e quando deixar os dados em um formato mais bruto.

Troquei ideias com outras pessoas que ingressaram na área de manipulação de dados quase na mesma época e ficou claro que em geral aprendemos da mesma maneira. Aqueles entre nós que tiveram sorte tinham colegas com quem compartilhar técnicas. A maioria dos textos sobre SQL é introdutória e básica (sem dúvida eles também são úteis) ou então se destina a desenvolvedores de bancos de dados. Há poucos recursos para usuários de SQL avançados dedicados ao trabalho de análise. As informações tendem a ficar restritas a pessoas ou pequenas equipes. Um dos objetivos deste livro é mudar isso, dando aos praticantes uma referência de como resolver problemas de análise comuns com SQL, e espero inspirar a execução de novas consultas com dados usando

técnicas que talvez você ainda não tenha visto.

Convenções usadas neste livro

As convenções tipográficas a seguir foram usadas no livro:

Itálico
: Indica novos termos, URLs, endereços de email, nomes de arquivo, extensões de arquivo e palavras-chave.

`Largura constante`
: Usada para listagens de programa, assim como dentro de parágrafos para indicar elementos de programa como nomes de variáveis ou funções, bancos de dados, variáveis de ambiente e instruções.

`Largura constante em negrito`
: Mostra comandos ou algum outro texto que precisem ser digitados literalmente pelo usuário.

`Largura constante em itálico`
: Mostra texto que deve ser substituído por valores fornecidos pelo usuário ou determinados pelo contexto.

 Este elemento representa uma dica ou sugestão.

 Este elemento representa uma nota geral.

 Este elemento indica um aviso ou cuidado.

Usando exemplos de código

Há material complementar (exemplos de código, exercícios etc.) disponível para download em *https://github.com/cathytanimura/sql_book*.

Se você tiver alguma dúvida ou problema técnico referente ao uso dos exemplos de código, envie um email para *bookquestions@oreilly.com*.

Este livro foi escrito para ajudá-lo a realizar seu trabalho. Se o exemplo de código estiver sendo oferecido com o livro, você poderá usá-lo em seus programas e na sua documentação. Não é preciso entrar em contato

conosco para obter permissão a menos que esteja reproduzindo uma parte significativa do código. Por exemplo, escrever um programa que use vários trechos de código deste livro não requer permissão. Vender ou distribuir exemplos dos livros da O'Reilly requer permissão. Responder a uma pergunta citando este livro e referindo-se a exemplos de código não requer permissão. Incorporar uma quantidade significativa de exemplos de código do livro à documentação do seu produto requer permissão.

Apreciamos, mas geralmente não exigimos, quando nos é atribuída autoria. Normalmente a atribuição de autoria inclui o título, o autor, a editora e o ISBN. Por exemplo, "*SQL for Data Analysis,* de Cathy Tanimura (O'Reilly). Copyright 2021 Cathy Tanimura, 978-1-492-08878-3".

Se você achar que o uso que está fazendo dos exemplos de código não se enquadra no uso ou na permissão legal mencionado anteriormente, fique à vontade para entrar em contato conosco em *permissions@oreilly.com*.

Como entrar em contato conosco

Envie comentários e dúvidas sobre este livro para: *novatec@novatec.com.br*.

Temos uma página da web para este livro, na qual incluímos a lista de erratas, exemplos e qualquer outra informação adicional.

- Página da edição em português
  https://novatec.com.br/livros/sql-para-analise-dados
- Página da edição original, em inglês
  https://oreil.ly/sql-data-analysis

Para obter mais informações sobre livros da Novatec, acesse nosso site em:
https://novatec.com.br

Agradecimentos

Este livro não teria sido possível sem os esforços de várias pessoas da O'Reilly. Andy Kwan me recrutou para este projeto. Amelia Blevins e Shira Evans me guiaram e deram feedbacks úteis ao longo do processo. Kristen Brown acompanhou o livro no processo de produção. Arthur Johnson melhorou a qualidade e a clareza do texto e inadvertidamente me fez pensar com mais profundidade nas palavras-chave SQL.

Com o passar dos anos, muitos colegas desempenharam um papel importante em minha jornada pelo SQL, e sou grato por seus tutoriais, dicas, e código compartilhado, e pelo tempo gasto tentando me ajudar a resolver problemas de análise. Sharon Lin abriu meus olhos para as expressões regulares. Elyse Gordon me deu muitos conselhos sobre a redação de um livro. Dave Hoch e nossas conversas sobre análise experimental inspiraram Capítulo 7. Dan, Jim e Stu da Star Chamber há muito tempo são meus companheiros favoritos para falar de "nerdices". Também agradeço a todos os colegas que me fizeram perguntas difíceis ao longo dos anos – e que, quando elas eram respondidas, faziam perguntas ainda mais difíceis.

Gostaria de agradecer ao meu marido Rick, ao meu filho Shea, às minhas filhas Lily e Fiona e à minha mãe Janet por seu amor, encorajamento, e principalmente pelo tempo para trabalhar neste projeto. Amy, Halle, Jessi e Den of Slack me mantiveram sã e rindo por meses de redação e confinamento pandêmico.

---

1 N.T.: Um data warehouse (armazém de dados ou depósito de dados) é utilizado para armazenar informações relativas às atividades de uma organização em bancos de dados de forma consolidada.

CAPÍTULO 1

Análise com SQL

Se você está lendo este livro, provavelmente tem interesse em análise de dados e no uso de SQL para executá-la. Pode ter experiência em análise de dados, mas ser iniciante em SQL, ou talvez tenha experiência em SQL, mas seja iniciante em análise de dados. Ou também pode ser totalmente iniciante nos dois tópicos. Qualquer que seja seu ponto de partida, este capítulo preparará o terreno para os tópicos abordados no restante do livro e garantirá que tenhamos um vocabulário em comum. Começarei com uma discussão sobre o que é análise de dados e depois abordarei o SQL: o que é, por que é tão popular, em que aspecto é semelhante a outras ferramentas e como se encaixa na análise de dados. Em seguida, já que a análise de dados moderna está tão ligada às tecnologias que permitiram que ela surgisse, concluirei com uma discussão dos diferentes tipos de bancos de dados que você pode encontrar em seu trabalho, por que eles são usados e o que tudo isso tem a ver com o SQL que você escreve.

O que é análise de dados?

Coletar e armazenar dados para análise é uma atividade muito humana. Sistemas que registram o inventário de grãos, impostos e dados demográficos existem há milhares de anos, e as raízes da estatística (*https://oreil.ly/1W6Jf*) datam de centenas de anos atrás. Disciplinas relacionadas, incluindo o controle estatístico de processos, a pesquisa operacional e a cibernética, surgiram em grande escala no século 20. Muitos nomes diferentes são usados para descrever a disciplina de análise de dados, como BI (business intelligence, inteligência empresarial), inteligência analítica, e ciência de decisão, e os praticantes têm cargos com várias denominações. A análise de dados também é feita por profissionais de marketing, engenheiros de produto e vários outros profissionais. Neste

livro, usarei os termos *analista de dados* e *cientista de dados* de maneira intercambiável para me referir à pessoa que trabalha com SQL para entender dados. Também vou me referir ao software usado para construir relatórios e painéis (dashboards) como *ferramentas de BI*.

A análise de dados contemporânea teve seu surgimento facilitado pela, e está ligada à, história da computação. Tendências tanto da pesquisa quanto da comercialização a moldaram, e a história inclui um quem é quem de pesquisadores e grandes empresas, assunto sobre o qual falaremos na seção sobre SQL. A análise de dados combina o poder da computação com técnicas de estatística tradicional. Ela é parte descoberta de dados, parte interpretação de dados e parte comunicação de dados. Com muita frequência, a finalidade da análise de dados é melhorar a tomada de decisões, realizada por humanos e cada vez mais por máquinas por meio da automação.

Uma metodologia sólida é essencial, mas a análise envolve mais do que apenas produzir o número certo. Envolve curiosidade, fazer perguntas e saber "por que" foram obtidos determinados números. Envolve padrões e anomalias, e descobrir e interpretar pistas de como as empresas e os humanos se comportam. Às vezes a análise é feita em um conjunto de dados coletado para responder a uma pergunta específica, como em uma definição científica ou um experimento online. A análise também é feita em dados que são gerados como resultado do fechamento de negócios, como nas vendas dos produtos de uma empresa, ou para fins analíticos, como no rastreio da interação do usuário em sites e aplicações móveis. Esses dados têm uma grande variedade de aplicações possíveis, da resolução de problemas ao planejamento de melhorias na UI (user interface, interface de usuário), mas com frequência chegam em um formato e volume que demandam que sejam processados antes de gerar respostas. O Capítulo 2 abordará a preparação de dados para a análise, e o Capítulo 8 discutirá alguns dos problemas éticos e de privacidade com os quais todos os praticantes da análise de dados devem estar familiarizados.

É difícil pensar em um setor que não tenha sido afetado pela análise de dados: manufatura, varejo, finanças, assistência médica, educação, e até mesmo o governo foram modificados por ela. As equipes esportivas

empregam a análise de dados desde os primeiros anos da nomeação de Billy Beane como gerente geral dos Oakland Athletics, popularizada pelo livro *Moneyball* (Norton) de Michael Lewis. A análise de dados é usada em marketing, vendas, logística, desenvolvimento de produtos, design de experiências do usuário, centrais de ajuda, recursos humanos e muito mais. A combinação de técnicas, aplicações e poder computacional levou ao surgimento de áreas relacionadas, como a engenharia de dados e a ciência de dados.

A análise de dados é por definição feita com dados históricos, e é importante lembrar que nem sempre o passado prevê o futuro. O mundo é dinâmico, e as organizações também – novos produtos e processos são introduzidos, concorrentes surgem e desaparecem, cenários sociopolíticos mudam. Críticas foram feitas contra a análise de dados por ela se basear no passado. Embora essa caracterização seja verdadeira, vi organizações tirarem grandes benefícios da análise de dados históricos. A mineração de dados históricos nos ajuda a entender as características e o comportamento de clientes, fornecedores e processos. Os dados históricos podem nos ajudar a desenvolver estimativas embasadas e a prever intervalos de resultados, que às vezes estarão errados, mas com frequência estarão certos. Os dados passados podem apontar lacunas, pontos fracos e oportunidades. Eles permitem que as organizações otimizem, economizem dinheiro e reduzam riscos e fraudes. Também podem ajudar as organizações a encontrar oportunidades, e podem se tornar a base de novos produtos que encantem os clientes.

São poucas as organizações que não fazem algum tipo de análise de dados atualmente, mas ainda há alguma resistência. Por que algumas organizações não usam a análise de dados? Uma das alegações é a relação custo-benefício. Coletar, processar e analisar dados dá trabalho e exige algum nível de investimento financeiro. Certas organizações também são novas ou atuam de maneira irregular. Se não houver um processo coerente, será difícil gerar dados com consistência suficiente para a análise. Para concluir, há considerações éticas. Coletar ou armazenar dados sobre certas pessoas em determinadas situações pode ser regulado por lei ou até mesmo ter sido banido. Dados sobre crianças e intervenções médicas são sigilosos, por exemplo, e há muitas leis que regulam sua coleta. Mesmo organizações direcionadas a dados têm de tomar cuidado com a privacidade do cliente e pensar bem em que dados devem ser

coletados, porque eles são necessários e por quanto tempo devem ser armazenados. Leis como o Regulamento Geral sobre a Proteção de Dados da União Europeia ou GDPR, e a Lei de Proteção da Privacidade do Consumidor da Califórnia, ou CCPA, mudaram a maneira como as empresas consideram os dados do consumidor. Discutiremos essas leis com mais detalhes no Capítulo 8. Como praticantes da análise de dados, devemos sempre pensar nas implicações éticas de nosso trabalho.

Quando trabalho com organizações, gosto de dizer às pessoas que a análise de dados não é um projeto que termine em uma data específica – é um estilo de vida. Desenvolver uma mentalidade baseada em dados é um processo e colher as recompensas é uma jornada. Detalhes desconhecidos tornam-se conhecidos, perguntas difíceis são tratadas gradualmente até haver respostas, e as informações mais críticas são embutidas em painéis que reforçam decisões táticas e estratégicas. Com essas informações, perguntas novas e mais difíceis são feitas e o processo se repete.

A análise de dados é acessível para quem quer começar, mas também é difícil dominá-la. A tecnologia pode ser aprendida, principalmente o SQL. Muitos problemas, como a otimização dos gastos com marketing ou a detecção de fraudes, são familiares e ocorrem em várias empresas. Cada organização é diferente e cada conjunto de dados tem suas peculiaridades, logo, até mesmo problemas conhecidos podem trazer novos desafios. Comunicar resultados é uma habilidade. Aprender a fazer boas recomendações e tornar-se um parceiro confiável de uma organização leva tempo. Pela minha experiência, uma análise simples apresentada de maneira persuasiva tem mais impacto do que uma análise sofisticada apresentada insatisfatoriamente. Uma análise de dados bem-sucedida também requer parceira. Você pode ter ótimos insights, mas se não houver ninguém para quem demonstrá-los, não será gerado impacto. Mesmo envolvendo tanta tecnologia, no fim das contas o que interessa são as pessoas, e relacionamentos importam.

Por que SQL?

Esta seção descreverá o que é SQL, os benefícios de usá-lo, em que ele se assemelha a outras linguagens normalmente usadas para análise e, para

concluir, como o SQL se encaixa no fluxo de trabalho de análise.

O que é SQL?

SQL é a linguagem usada na comunicação com bancos de dados. O acrônimo significa Structured Query Language e é pronunciado como "sequel" ou pela articulação de cada letra, como em "ess cue el". Essa é apenas uma das muitas controvérsias e inconsistências que veremos existentes em torno do SQL, mas a maioria das pessoas saberá o que você quer dizer independentemente de como o diga. Há algum debate sobre se o SQL é ou não uma linguagem de programação. Ele não é uma linguagem de uso geral como o são o C ou o Python. O SQL sem um banco de dados e sem dados em tabelas é apenas um arquivo de texto. Ele não pode construir um site, mas é poderoso para o trabalho com dados em bancos de dados. Do ponto de vista da praticidade, o que mais importa é que o SQL pode ajudá-lo a executar a tarefa de análise de dados.

A IBM foi a primeira a desenvolver bancos de dados SQL, a partir do modelo relacional inventado por Edgar Codd nos anos 1960. O modelo relacional era uma descrição teórica do gerenciamento de dados com o uso de relacionamentos. Com a criação dos primeiros bancos de dados, a IBM trouxe avanços para a teoria, mas também havia considerações comerciais, como as tinha a Oracle, a Microsoft e outras empresas que desde então comercializaram um banco de dados. Desde o início, houve tensão entre a teoria da computação e a realidade comercial. O SQL se tornou um padrão ISO (International Organization for Standards, da Organização Internacional de Normalização) em 1987 e um padrão ANSI (American National Standards Institute, do Instituto Nacional Americano de Padrões) em 1986. Embora os principais bancos de dados tenham começado a implementar o SQL a partir desses padrões, muitos têm variações e funções que facilitam a vida dos usuários. Isso apresenta a desvantagem de tornar mais difícil migrar o SQL entre bancos de dados sem algumas modificações.

O SQL é usado no acesso, na manipulação e na recuperação de dados a partir de objetos em um banco de dados. Os bancos de dados podem ter um ou mais *esquemas (schemas)*, que fornecem a organização e a estrutura e

contêm outros objetos. Dentro de um esquema, os objetos mais usados na análise de dados são as tabelas, as views e as funções. As tabelas contêm campos, que armazenam os dados. Elas podem ter um ou mais *índices*; um índice é um tipo especial de estrutura de dados que permite que os dados sejam recuperados com mais eficiência. Geralmente eles são definidos por um administrador de banco de dados. As views são basicamente consultas (queries) armazenadas que podem ser referenciadas da mesma forma que uma tabela. As funções permitem que conjuntos de cálculos ou procedimentos normalmente usados sejam armazenados e facilmente referenciados em consultas. Elas costumam ser criadas por um administrador de banco de dados, ou DBA (database administrator). A Figura 1.1 fornece uma visão geral da organização dos bancos de dados.

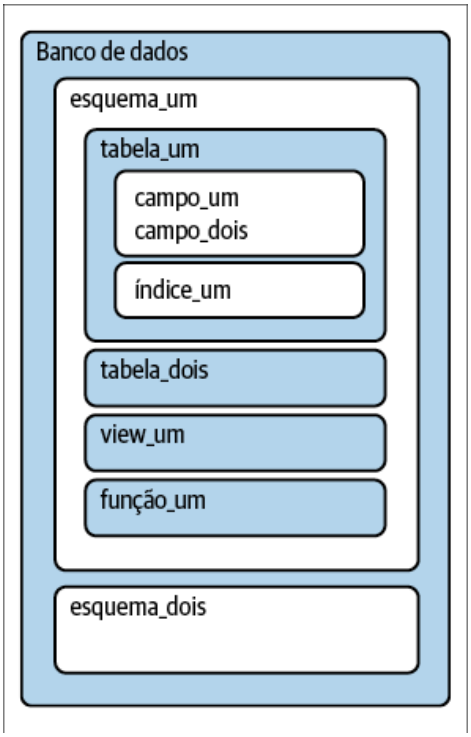


Figura 1.1: Visão geral da organização e dos objetos de um banco de dados.

Para se comunicar com os bancos de dados, o SQL tem quatro sublinguagens que lidam com diferentes tarefas, e elas são em grande parte padrão entre os tipos de bancos de dados. A maioria das pessoas que trabalham com análise de dados não precisa se lembrar dos nomes dessas sublinguagens no dia a dia, mas podem ter de conversar com administradores de banco de dados ou engenheiros de dados; logo, farei

uma breve introdução delas. Todos os comandos funcionam de maneira fluida e em conjunto, e alguns podem coexistir na mesma instrução SQL.

É principalmente com o *DQL*, ou *data query language (linguagem de consulta de dados)*, que este livro vai lidar. Ele é usado na *consulta* de dados, o que poderíamos considerar o uso de código para fazer perguntas a um banco de dados. Os comandos DQL incluem *SELECT*, que será familiar para quem já é usuário do SQL, mas de acordo com minha experiência o acrônimo DQL não é usado com frequência. As consultas SQL podem ser tão curtas a ponto de ter uma única linha ou se estender por muitas dezenas de linhas. Elas podem acessar uma única tabela (ou view), podem combinar dados de várias tabelas com o uso de junções (joins) e também podem consultar vários esquemas do mesmo banco de dados. Geralmente o SQL não pode fazer consultas entre bancos de dados, mas, em alguns casos, configurações de rede mais inteligentes ou um software adicional podem ser usados para recuperar dados de várias fontes, até mesmo de bancos de dados de diferentes tipos. As consultas SQL são autônomas e, além de tabelas, não referenciam variáveis ou saídas de etapas anteriores não contidas na consulta, ao contrário das linguagens de script.

O *DDL*, ou *data definition language (linguagem de definição de dados)*, é usado para criar e modificar tabelas, views, usuários e outros objetos do banco de dados. Ele afeta a estrutura, mas não o conteúdo. Há três comandos comuns: *CREATE*, *ALTER* e *DROP*. *CREATE* é usado na criação de novos objetos. *ALTER* altera a estrutura de um objeto, como pelo acréscimo de uma coluna a uma tabela. *DROP* exclui o objeto inteiro e sua estrutura. Você pode ouvir DBAs e engenheiros de dados falarem sobre trabalhar com DDLs – o que é apenas uma maneira abreviada de se referir aos arquivos ou trechos de código que executam as criações, alterações ou exclusões. Um exemplo de como o DDL é usado no contexto da análise é o código de criação de tabelas temporárias.

O *DCL*, ou *data control language (linguagem de controle de dados)*, é usado no controle de acesso. Os comandos incluem *GRANT* e *REVOKE*, que dão e removem permissões, respectivamente. Em um contexto de análise, *GRANT* pode ser necessário para permitir que um colega consulte uma tabela que você criou. Você também pode deparar com esse comando se alguém lhe disser que uma tabela existe, mas não for possível vê-la –

permissões podem ter de ser concedidas (com *GRANT*) ao seu usuário.

O *DML*, ou *data manipulation language (linguagem de manipulação de dados)*, é usado na atuação sobre os dados propriamente ditos. Os comandos são *INSERT*, *UPDATE* e *DELETE*. *INSERT* adiciona novos registros e é basicamente a etapa de "carregamento" do processo ETL (extract, transform, load; extração, transformação, carregamento). *UPDATE* altera os valores de um campo, e *DELETE* remove linhas. Você encontrará esses comandos se tiver algum tipo de tabela autogerenciada – tabelas temporárias, tabelas sandbox – ou se desempenhar ao mesmo tempo a função de proprietário e analisador do banco de dados.

Essas quatro sublinguagens estão presentes em todos os principais bancos de dados. Neste livro, darei ênfase ao DQL. Veremos alguns comandos DDL e DML no Capítulo 8, e você também pode ver alguns exemplos no site do GitHub referente ao livro (*https://github.com/cathytanimura/sql_book*), no qual eles são usados para criar e fornecer os dados usados nos exemplos. Devido a esse conjunto comum de comandos, um código SQL escrito para um banco de dados será familiar para qualquer pessoa acostumada a trabalhar com SQL. No entanto, ler SQL a partir de outro banco de dados seria como escutar alguém que fala o mesmo idioma que você, mas vindo de outra parte do país ou do mundo. A estrutura básica da linguagem é a mesma, porém o jargão é diferente, e algumas palavras têm significados distintos. As variações do SQL de um banco de dados para outro com frequência são chamadas de *dialetos*, e os usuários do banco de dados farão alusão a Oracle SQL, MSSQL ou outros dialetos.

Mesmo assim, uma vez que você dominar o SQL, poderá trabalhar com diferentes tipos de bancos de dados contanto que preste atenção em detalhes como a manipulação de valores nulos, datas e timestamps (carimbos de data/hora); a divisão de inteiros; e a diferenciação de letras maiúsculas e minúsculas.

Este livro está usando o PostgreSQL, ou Postgres, para os exemplos, mas tentarei destacar onde o código seria significativamente diferente em outros tipos de bancos de dados. Você pode instalar o Postgres (*https://www.postgresql.org/download/*) em um computador pessoal para acompanhar os exemplos.

Benefícios do SQL

Há muitas razões para usarmos SQL na análise de dados, que vão do poder computacional à sua onipresença em ferramentas de análise de dados e à sua flexibilidade.

Talvez a melhor razão para o uso do SQL seja que grande parte dos dados mundiais já está em bancos de dados. É provável que sua organização tenha um ou mais bancos de dados. Mesmo se os dados ainda não estiverem em um banco de dados, pode valer a pena você carregá-los em um para se beneficiar das vantagens computacionais e de armazenamento, principalmente em comparação com as alternativas, como as planilhas. O poder computacional passou por uma grande expansão nos últimos anos, e os data warehouses e a infraestrutura de dados evoluíram para se beneficiar dele. Alguns bancos de dados em nuvem mais novos permitem que quantidades massivas de dados sejam consultadas na memória, tornando tudo mais rápido. A época em que esperávamos minutos ou horas pelo retorno dos resultados das consultas pode ter terminado, embora os analistas estejam escrevendo consultas mais complexas em resposta.

O SQL é o padrão de fato[1] para a interação com bancos de dados e a recuperação de dados neles. Uma ampla variedade de softwares populares se conecta com bancos de dados com SQL, desde planilhas a ferramentas de BI e de virtualização e linguagens de codificação como Python e R (discutidas na próxima seção). Devido aos recursos computacionais disponíveis, executar o máximo de manipulação e agregação de dados possível no banco de dados com frequência traz vantagens posteriores. Discutiremos com detalhes estratégias para a construção de conjuntos de dados (data sets) complexos para ferramentas downstream no Capítulo 8.

Os componentes básicos do SQL podem ser combinados em uma quantidade interminável de maneiras. Começando com um número relativamente pequeno de componentes básicos – a sintaxe – o SQL pode executar várias tarefas. O SQL pode ser desenvolvido iterativamente, e é fácil rever os resultados ao longo do processo. Talvez ele não seja uma linguagem de programação plena, mas pode fazer muita coisa, desde transformar dados a efetuar cálculos complexos e responder perguntas.

Para concluir, o SQL é relativamente fácil de aprender, com uma quantidade finita de sintaxe. Você pode aprender as palavras-chave e a estrutura rapidamente e então desenvolver sua habilidade com o tempo trabalhando com conjuntos de dados variados. As aplicações do SQL são praticamente infinitas, se levarmos em consideração a variedade de conjuntos de dados existentes no mundo e as possíveis perguntas que podem ser feitas aos dados. O SQL é ensinado em várias universidades, e muitas pessoas desenvolvem habilidades no trabalho. Até mesmo funcionários que ainda não têm prática em SQL podem ser treinados, e a curva de aprendizagem pode ser mais branda que a de outras linguagens de programação. Isso torna o armazenamento de dados para análise em bancos de dados relacionais uma opção lógica para as organizações.

SQL versus R ou Python

Embora o SQL seja uma linguagem popular para a análise de dados, ele não é a única opção. R e Python estão entre as outras linguagens também populares para a análise de dados. R é uma linguagem estatística e de grafos, enquanto Python é uma linguagem de programação de uso geral que apresenta vantagens no trabalho com dados. As duas são open source, podem ser instaladas em um laptop e têm comunidades ativas desenvolvendo pacotes, ou extensões, que lidam com várias tarefas de manipulação e análise de dados. Escolher entre R e Python não faz parte do escopo deste livro, mas há muitas discussões online sobre as vantagens relativas de cada um. Vou considerá-las juntas aqui como linguagens de codificação alternativas ao SQL.

Uma diferença importante entre o SQL e outras linguagens de codificação é o local onde o código é executado e, portanto, a quantidade de poder computacional disponível. O SQL é sempre executado em um servidor de banco de dados, beneficiando-se de todos os seus recursos computacionais, Para a realização de análises, geralmente o R e o Python são executados localmente em nossa máquina, logo os recursos computacionais ficam limitados ao que estiver disponível localmente. É claro que há exceções: os bancos de dados podem ser executados em laptops, e o R e o Python podem ser executados em servidores com mais recursos. Quando você estiver realizando uma análise mais simples em

grandes conjuntos de dados, fazer o trabalho em um servidor de banco de dados com mais recursos será uma boa opção. Já que normalmente os bancos de dados são configurados para receber novos dados continuamente, o SQL também é uma boa opção quando um relatório ou painel precisa ser atualizado periodicamente.

Uma segunda diferença é a de como os dados são armazenados e organizados. Os bancos de dados relacionais sempre organizam os dados em linhas e colunas dentro de tabelas, logo o SQL assume essa estrutura para cada consulta. O R e o Python têm mais maneiras de armazenar dados, que incluem variáveis, listas e dicionários, entre outras opções. Elas dão maior flexibilidade, mas com uma curva de aprendizagem mais íngreme. Para facilitar a análise de dados, o R tem os data frames, que são semelhantes às tabelas do banco de dados e organizam os dados em linhas e colunas. O pacote pandas torna os DataFrames disponíveis em Python. Mesmo quando outras opções estão disponíveis, a estrutura de tabela continua sendo valiosa para a análise.

O looping é outra diferença importante entre o SQL e a maioria das outras linguagens de programação de computadores. Um *loop* é uma instrução ou um conjunto de instruções que se repete até uma condição especificada ser atendida. As agregações SQL executam um loop implicitamente pelo conjunto de dados, sem nenhum código adicional. Veremos posteriormente como a incapacidade de executar loops pelos campos pode resultar em instruções SQL longas quando pivotamos ou despivotamos[2] dados. Embora uma discussão mais detalhada não faça parte do escopo deste livro, alguns fornecedores criaram extensões do SQL, como o PL/SQL no Oracle e o T-SQL no Microsoft SQL Server, que permitem o uso de uma funcionalidade como o looping.

Uma desvantagem do SQL é que os dados precisam estar em um banco de dados,[3] enquanto o R e o Python podem importar dados de arquivos armazenados localmente ou acessar arquivos armazenados em servidores ou sites. Isso é conveniente para muitos projetos pontuais. Um banco de dados pode ser instalado em um laptop, mas isso adiciona uma camada extra de sobrecarga. Ao contrário, pacotes como dbplyr para R e SQLAlchemy para Python permitem que programas escritos nessas

linguagens se conectem com bancos de dados, executem consultas SQL e usem os resultados em etapas de processamento posteriores. Nesse aspecto, o R ou o Python podem complementar o SQL.

Tanto o R quanto o Python têm funções estatísticas sofisticadas que são internas ou estão disponíveis em pacotes. Embora o SQL tenha, por exemplo, funções para o cálculo de média e desvio-padrão, os cálculos de valores-p e significância estatística que são necessários na análise experimental (discutida no Capítulo 7) não podem ser feitos somente com SQL. Além da estatística sofisticada, o machine learning é outra área que uma dessas outras linguagens de codificação pode manipular melhor.

Quando estiver resolvendo se deve usar SQL, R ou Python para uma análise, considere:

- Onde os dados estão localizados – em um banco de dados, em um arquivo, em um site?
- Qual é o volume de dados?
- Para onde vão os dados – para um relatório, para uma visualização, para uma análise estatística?
- Eles precisarão ser atualizados ou substituídos por novos dados? Com que frequência?
- O que sua equipe ou organização usa? É importante que esteja em conformidade com os padrões existentes?

Há muito debate sobre que linguagens e ferramentas são melhores para a análise de dados ou a ciência de dados. Como ocorre em várias áreas, com frequência há mais de uma maneira de executar uma análise. As linguagens de programação evoluem e sua popularidade muda, e temos sorte de viver e trabalhar com tantas opções boas. O SQL já existe há muito tempo e provavelmente continuará sendo popular por anos. O objetivo que queremos atingir é usar a melhor ferramenta disponível para a realização da tarefa. Este livro o ajudará a aproveitar ao máximo o SQL para a análise de dados, independentemente do que mais houver em seu kit de ferramentas.

SQL como parte do fluxo de trabalho de análise de dados

Agora que expliquei o que é SQL, discuti alguns de seus benefícios e o

comparei a outras linguagens, passaremos à discussão de onde o SQL se encaixa no processo de análise de dados. O trabalho de análise sempre começa com uma pergunta, que poderia ser sobre quantos clientes novos foram atraídos, qual é a tendência das vendas ou por que alguns usuários permanecem fiéis por muito tempo enquanto outros testam um serviço e nunca mais voltam. Após a pergunta ser criada, temos de considerar qual é a origem dos dados, onde eles estão armazenados, qual é o plano de análise e como os resultados serão apresentados para o público-alvo. A Figura 1.2 mostra as etapas do processo. As consultas e a análise são o ponto central deste livro, mas abordarei as outras etapas brevemente para inserir o estágio de consultas e análise em um contexto mais amplo.

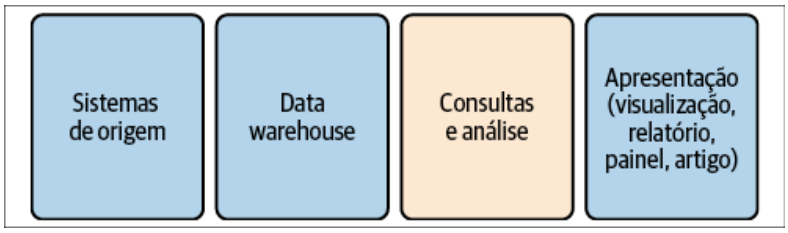


Figura 1.2: Etapas do processo de análise de dados.

Primeiro, os dados são gerados por *sistemas de origem*, um termo que inclui qualquer processo humano ou de máquina que gere dados de interesse. Os dados podem ser gerados pelas pessoas manualmente, como quando alguém preenche um formulário ou toma nota durante uma visita ao médico. Também podem ser gerados por máquina, como quando o banco de dados de uma aplicação registra uma compra, um sistema de eventos registra um clique no site ou uma ferramenta de gestão de marketing registra a abertura de um email. Os sistemas de origem podem gerar muitos tipos e formatos de dados diferentes, e o Capítulo 2 os discutirá, assim como abordará como o tipo de origem pode afetar a análise, com mais detalhes.

A segunda etapa é mover os dados e armazená-los em um banco de dados para análise. Usarei os termos *data warehouse*, que é um banco de dados que consolida dados de toda a organização em um repositório central, e *data store*, que se refere a qualquer tipo de sistema de armazenamento de dados que possa ser consultado. Outros termos que você pode encontrar são *data mart*, que normalmente é um subconjunto de um data warehouse,

ou um data warehouse com enfoque mais restrito; e *data lake*, um termo que pode significar que os dados residem em um sistema de armazenamento de arquivos ou que eles estão armazenados em um banco de dados, mas sem o nível de transformação de dados que é comum nos data warehouses. Os data warehouses variam de pequenos e simples a imensos e caros. Um banco de dados sendo executado em um laptop será suficiente para você acompanhar os exemplos deste livro. O importante é que você esteja com os dados de que precisa para executar uma análise juntos no mesmo local.

Geralmente uma pessoa ou equipe é responsável por obter dados e inseri-los no data warehouse. Esse processo chama-se *ETL*, ou extrair, transformar, carregar. A extração traz os dados do sistema de origem. A transformação opcionalmente altera a estrutura dos dados, executa a limpeza para a melhoria da qualidade dos dados, ou agrega os dados. O carregamento insere os dados no banco de dados. Esse processo também pode ser chamado de *ELT*, abreviação de extract, load, transform – a diferença é que, em vez de as transformações serem feitas antes de os dados serem carregados, todos os dados são carregados e então as transformações são executadas, geralmente com o uso de SQL. Você também pode ouvir os termos *origem* e *destino* no contexto do ETL. A origem é de onde os dados vêm e o destino é o ponto de chegada, isto é, o banco de dados e as tabelas que existem nele. Mesmo quando o SQL é usado para fazer a transformação, outra linguagem como o Python ou Java é empregada para agregar as etapas, coordenar o agendamento e emitir alertas quando algo dá errado. Há vários produtos comerciais assim como ferramentas open source disponíveis; logo, as equipes não precisam criar um sistema ETL totalmente a partir do zero.

Quando os dados estiverem em um banco de dados, a próxima etapa é executar as consultas e a análise. Nessa etapa, o SQL é aplicado à exploração, criação de perfis, limpeza, modelagem e análise dos dados. A Figura 1.3 mostra o fluxo geral do processo. A exploração dos dados envolve familiarizar-se com o tópico, com onde os dados foram gerados e com as tabelas de banco de dados em que eles estão armazenados. A criação de perfis envolve verificar os valores exclusivos e a distribuição de registros no conjunto de dados. A limpeza envolve corrigir dados incorretos ou incompletos, adicionar categorização e flags, e manipular valores nulos. Modelagem é o processo de organizar os dados nas linhas e

colunas necessárias no conjunto de resultados. Para concluir, a análise dos dados envolve procurar tendências, conclusões e insights na saída. Embora esse processo esteja sendo seja mostrado como linear, geralmente na prática ele é cíclico – por exemplo, quando a modelagem ou a análise revela dados que precisam ser limpos.

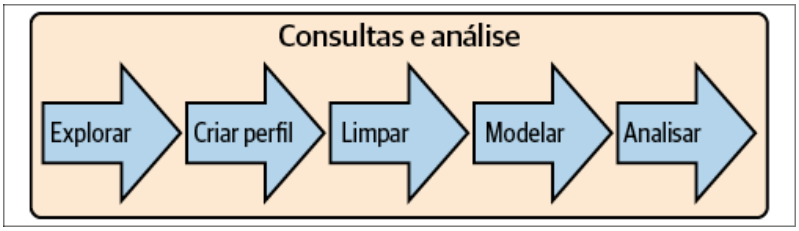


Figura 1.3: Estágios dentro da etapa de consultas e análise do fluxo de trabalho de análise.

A apresentação dos dados na forma de uma saída final é a última etapa do fluxo de trabalho geral. Executivos não gostam de receber um arquivo de código SQL; eles esperam que você apresente gráficos, diagramas e insights. A comunicação é essencial para a análise gerar impacto, e para que isso ocorra precisamos de uma maneira de compartilhar os resultados com outras pessoas. Em certas situações, pode ser preciso aplicar uma análise estatística mais sofisticada do que seria possível em SQL, ou talvez você queira fornecer os dados para um algoritmo de machine learning (ML). Felizmente, a maioria das ferramentas de relatórios e visualização tem conectores SQL que permitem trazer dados de tabelas inteiras ou de consultas SQL pré-gravadas. Os softwares e linguagens estatísticos normalmente usados para ML também costumam ter conectores SQL.

Os fluxos de trabalho de análise englobam várias etapas e com frequência incluem diversas ferramentas e tecnologias. As consultas e a análise SQL são a essência de muitas análises e são o que abordaremos nos próximos capítulos. O Capítulo 2 discutirá os tipos de sistemas de origem e os tipos de dados que eles geram. O restante deste capítulo examinará os tipos de bancos de dados que provavelmente você encontrará em sua jornada de análise.

Tipos de bancos de dados e como trabalhar com eles

Se você estiver trabalhando com SQL, estará trabalhando com bancos de dados. Há vários tipos de bancos de dados – de open source a proprietário, de row-store a column-store. Há bancos de dados locais e bancos de dados em nuvem, assim como há bancos de dados híbridos, caso em que a organização executa o software de banco de dados na infraestrutura do fornecedor de nuvem. Também há vários data stores que não são bancos de dados, mas podem ser consultados com SQL.

Os bancos de dados não são todos criados da mesma forma; cada tipo de banco de dados tem suas vantagens e desvantagens quando se trata de trabalho de análise. Ao contrário do que ocorre com as ferramentas usadas em outras partes do fluxo de trabalho de análise, talvez você não possa influir na tecnologia de banco de dados usada em sua organização. Conhecer as vantagens e desvantagens do banco de dados que você tem o ajudará a trabalhar com mais eficiência e a se beneficiar de qualquer função SQL especial que ele ofereça. A familiaridade com outros tipos de bancos de dados o ajudará se você estiver trabalhando em um projeto para construir ou fazer a migração para um novo data warehouse. Você pode querer instalar um banco de dados em seu laptop para projetos pessoais de pequena escala, ou obter uma instância de um warehouse na nuvem por razões semelhantes.

Os bancos de dados e os data stores têm sido uma área dinâmica do desenvolvimento tecnológico desde que foram introduzidos. Algumas tendências surgidas desde a passagem para o século 21 influenciaram a tecnologia de maneiras muito empolgantes para os praticantes atuais da análise de dados. Em primeiro lugar, os volumes de dados aumentaram enormemente com a internet, os dispositivos móveis e a IoT (Internet of Things, Internet das Coisas). Em 2020, a IDC previu (*https://oreil.ly/oEWDD*) que a quantidade de dados armazenada globalmente crescerá para 175 zettabytes até 2025. Essa escala de dados é difícil até mesmo de imaginar, e ela não estará toda armazenada em bancos de dados para análise. Não é raro empresas terem dados na escala de terabytes e petabytes atualmente, uma escala que teria sido impossível processar com a tecnologia dos anos 1990 e de anos anteriores. Em segundo lugar, a diminuição nos custos de armazenamento de dados e computação, além do surgimento da nuvem, tornou mais barato e fácil para as organizações coletarem e armazenarem

essas quantidades massivas de dados. A memória dos computadores ficou mais barata, o que significa que grandes quantidades de dados podem ser carregadas na memória, cálculos podem ser executados e resultados podem ser retornados, tudo isso sem leitura e gravação em disco, aumentando muito a velocidade. Em terceiro lugar, a computação distribuída permitiu a divisão de workloads em muitas máquinas. Isso possibilita que uma grande e ajustável quantidade de recursos computacionais seja direcionada a tarefas de dados complexas.

Os bancos de dados e os data stores combinaram essas tendências tecnológicas de várias maneiras diferentes para otimizar certos tipos de tarefas. Há duas categorias amplas de bancos de dados que são relevantes para o trabalho de análise: row-store e column-store. Na próxima seção irei introduzi-las, discutirei o que as torna semelhantes e diferentes umas das outras e abordarei o que tudo isso significa para a execução da análise com os dados armazenados nelas. Para concluir, introduzirei alguns tipos adicionais de infraestrutura de dados que você pode encontrar além dos bancos de dados.

Bancos de dados row-store

Os bancos de dados *row-store* – também chamados de bancos de dados *transacionais* – são projetados para serem eficientes no processamento de transações: *INSERT*s, *UPDATE*s e *DELETE*s. Entre os bancos de dados row-store open source estão o MySQL e o Postgres. No lado comercial, o Microsoft SQL Server, o Oracle e o Teradata são amplamente usados. Embora eles não sejam otimizados para a análise, por muitos anos os bancos de dados row-store foram a única opção para empresas que estavam construindo data warehouses. Por meio de um design com um ajuste e um esquema cuidadosos, esses bancos de dados podem ser usados para análise. Eles também são interessantes devido ao baixo custo das opções open source e porque são familiares para os administradores de banco de dados que os mantêm. Muitas organizações replicam seu banco de dados de produção na mesma tecnologia como uma primeira etapa em direção à construção da infraestrutura de dados. Por todas essas razões, provavelmente os analistas e cientistas de dados trabalharão com dados de um banco de dados row-store em algum momento de sua

carreira.

Imaginamos uma tabela como um conjunto de linhas e colunas, mas os dados têm de ser serializados para armazenamento. Uma consulta procura os dados necessários em um disco rígido. Os discos rígidos são organizados em uma série de blocos de tamanho fixo. Examinar os discos rígidos demanda tempo e recursos; logo, reduzir a parcela do disco que precisa ser examinada para retornar resultados de consulta é importante. Os bancos de dados row-store resolvem esse problema serializando os dados em uma linha. A Figura 1.4 mostra um exemplo do armazenamento de dados linha por linha (row-wise). No momento da consulta, a linha inteira é lida na memória. Essa abordagem é rápida para a execução de atualizações linha por linha, porém é mais lenta quando fazemos cálculos com muitas linhas quando apenas algumas colunas seriam necessárias.

id	sku	tipo	cor	tamanho	preço
1	123	camiseta	preto	P	19,99
2	124	calção	verde	M	24,99

Figura 1.4: Armazenamento row-wise, no qual todas as linhas são armazenadas juntas no disco.

Para reduzir a largura das tabelas, geralmente os bancos de dados transacionais são modelados na *terceira forma normal*, que é uma abordagem de design de banco de dados que tenta armazenar cada informação apenas uma vez, para evitar duplicação e inconsistências. Ela é eficiente para o processamento de transações, mas com frequência leva a um grande número de tabelas no banco de dados, cada uma com apenas algumas colunas. Para a análise desses dados, muitas junções podem ser necessárias, e pode ser difícil para não desenvolvedores entenderem como as tabelas estão relacionadas e onde um dado específico está armazenado. Na execução de uma análise, o objetivo costuma ser a desnormalização, ou a obtenção de todos os dados juntos no mesmo local.

Normalmente, as tabelas têm uma *chave primária* que impõe exclusividade – em outras palavras, impede que o banco de dados crie mais de um registro para o mesmo item. Com frequência, as tabelas têm uma coluna `id` com um inteiro que aumenta automaticamente, onde cada novo registro recebe o próximo inteiro após o último que foi inserido ou um

valor alfanumérico que é criado por um gerador de chave primária. Também é preciso haver um conjunto de colunas que juntas tornem a linha exclusiva; essa combinação de campos é chamada de *chave composta (composite key)*, ou também de *chave de negócio (business key)*. Por exemplo, em uma tabela de pessoas, juntas as colunas `first_name` (nome), `last_name` (sobrenome) e `birthdate` (data de nascimento) podem tornar a linha exclusiva. `Social_security_id` (número de previdência social) também seria um identificador exclusivo, além da coluna `person_id` (identificador pessoal) da tabela.

Opcionalmente as tabelas também podem ter índices para acelerar a busca de registros específicos e a criação de junções que envolvam essas colunas. Os índices armazenam os valores do campo ou dos campos indexados como dados exclusivos junto com um ponteiro de linha, e já que os índices são menores do que a tabela inteira, são mais fáceis de encontrar. Geralmente a chave primária é indexada, mas outros campos ou grupos de campos também podem ser indexados. No trabalho com bancos de dados row-store, é útil saber que campos das tabelas têm índices. A criação de junções comuns pode ser acelerada com o acréscimo de índices; logo, vale a pena investigar se as consultas da análise demorarão muito para ser executadas. Os índices cobram seu preço: eles ocupam espaço de armazenamento, e retardam o carregamento, à medida que novos valores precisam ser adicionados a cada inserção. Portanto, os DBAs podem não indexar tudo que seria útil para a análise. Além disso, exceto pela criação de relatórios, o trabalho de análise pode não ser suficientemente rotineiro para precisarmos nos importar com a otimização dos índices. Consultas exploratórias e complexas com frequência usam padrões de junção complicados, e podemos descartar uma abordagem ao descobrir uma nova maneira de resolver um problema.

A *modelagem de esquema estrela* (*https://oreil.ly/5WiSp*) foi desenvolvida em parte para tornar os bancos de dados mais amigáveis para workloads analíticos. As bases foram dispostas no livro *The Data Warehouse Toolkit*,[4] que defende a modelagem dos dados como uma série de tabelas de fatos e de dimensões. As tabelas de fatos representam eventos, como as transações de uma loja varejista. As dimensões contêm descritores como o nome do

cliente e o tipo de produto. Já que sem sempre os dados se encaixam perfeitamente nas categorias de fatos e dimensões, há uma extensão chamada *esquema floco de neve* (*snowflake schema*) (*https://oreil.ly/rpj4N*), na qual algumas dimensões também têm suas próprias dimensões.

Bancos de dados column-store

Os bancos de dados *column-store* se popularizaram no início do século 21, mas sua teoria é da mesma época dos bancos de dados row-store. Eles armazenam os valores de uma coluna juntos, em vez de armazenar os valores da linha. Esse design é ideal para consultas que leem muitos registros, mas não leem necessariamente todas as colunas. Os bancos de dados column-store populares são o Amazon Redshift, o Snowflake e o Vertica.

Os bancos de dados column-store são eficientes no armazenamento de grandes volumes de dados graças à compactação. Valores ausentes e repetidos podem ser representados por valores de marcador muito pequenos em vez do valor nulo. Por exemplo, em vez de armazenar "Reino Unido" milhares ou milhões de vezes, um banco de dados column-store armazenará um valor substituto que ocupará muito pouco espaço de armazenamento, junto com uma pesquisa contendo o valor "Reino Unido" completo. Os bancos de dados column-store também compactam dados beneficiando-se das repetições de valores nos dados armazenados. Por exemplo, o banco de dados pode armazenar o fato de que o valor do marcador de "Reino Unido" é repetido 100 vezes, e isso ocupará ainda menos espaço do que armazenar o marcador 100 vezes.

Os bancos de dados column-store não impõem o uso de chaves primárias e não têm índices. Valores repetidos não são um problema, graças à compactação. Como resultado, os esquemas podem ser personalizados para as consultas da análise, com todos os dados ficando juntos no mesmo local em vez de estarem em várias tabelas que precisem de junção. No entanto, dados duplicados podem ser facilmente introduzidos sem as chaves primárias; logo, conhecer a origem dos dados e verificar a qualidade são itens importantes.

As atualizações e exclusões são custosas na maioria dos bancos de dados column-store, já que os dados de uma única linha ficam distribuídos em

vez de serem armazenados juntos. Para tabelas muito grandes, pode existir uma política de somente gravação; logo, também precisamos saber como os dados são gerados para identificar que registros usar. Além disso, a leitura dos dados pode ser mais lenta, porque eles precisam ser descompactados antes de cálculos serem aplicados.

Geralmente os bancos de dados column-store são o padrão ouro para um trabalho de análise mais rápido. Eles usam SQL padrão (com algumas variações específicas do fornecedor) e em muitos aspectos trabalhar com eles não será diferente de trabalhar com um banco de dados row-store no que diz respeito às consultas que você escrever. O tamanho dos dados importa, assim como os recursos computacionais e de armazenamento que foram alocados para o banco de dados. Vi agregações percorrerem milhões e também bilhões de registros em segundos. Isso faz maravilhas para a produtividade.

Há alguns truques que é preciso aprender. Já que certos tipos de compactação dependem da classificação, conhecer os campos de acordo com os quais a tabela é classificada e usá-los para filtrar as consultas melhora o desempenho. A junção de tabelas pode ser lenta se as duas tabelas forem grandes.

Resumindo, pode ser mais fácil ou mais rápido trabalhar com alguns bancos de dados, mas não há nada inerente ao tipo do banco de dados que impeça a execução de nenhuma das análises deste livro. Como acontece em qualquer situação, usar uma ferramenta que seja apropriadamente poderosa para o volume de dados e a complexidade da tarefa permitirá que você se concentre na criação de uma análise significativa.

Outros tipos de infraestrutura de dados

Os bancos de dados não são a única maneira pela qual os dados podem ser armazenados e há uma variedade cada vez maior de opções para o armazenamento dos dados necessários na análise e a otimização de aplicações. Os sistemas de armazenamento de arquivos, às vezes chamados de *data lakes*, talvez sejam a principal alternativa aos database warehouses. Bancos de dados NoSQL e data stores baseados em pesquisa (search-based data stores) são sistemas de armazenamento de dados que oferecem

baixa latência para o desenvolvimento de aplicações e a busca de arquivos de log. Embora normalmente não façam parte do processo de análise, cada vez mais eles fazem parte da infraestrutura de dados das organizações; logo, também vou apresentá-los brevemente nesta seção. Uma tendência interessante a ser destacada é que, mesmo que inicialmente esses novos tipos de infraestrutura visassem nos libertar da dependência dos bancos de dados SQL, muitos acabaram implementando alguma variação de interface SQL para a consulta de dados.

O Hadoop, também conhecido como HDFS (abreviatura de "Hadoop distributed filesystem"), é um sistema de armazenamento de arquivos open source que se beneficia do custo sempre em queda do armazenamento de dados e do poder computacional, assim como de sistemas distribuídos. Os arquivos são divididos em blocos e o Hadoop os distribui em um sistema de arquivos que é armazenado em nós, ou computadores, em um cluster. O código para a execução de operações é enviado para os nós e eles processam os dados em paralelo. A grande inovação do Hadoop foi permitir que quantidades imensas de dados sejam armazenadas de forma barata. Muitas empresas grandes da internet, com quantidades massivas de dados com frequência não estruturados, acharam que essa era uma vantagem sobre o custo e as limitações de armazenamento dos bancos de dados tradicionais. As primeiras versões do Hadoop apresentavam duas grandes desvantagens: habilidades em codificação especializadas eram necessárias para a recuperação e o processamento de dados, já que ele não era compatível com o SQL, e o tempo de execução dos programas costumava ser muito longo. Desde então o Hadoop evoluiu e foram desenvolvidas várias ferramentas que permitem acesso SQL ou semelhante ao do SQL aos dados e aceleram os tempos de consulta.

Outros produtos comerciais e open source foram introduzidos nos últimos anos para se beneficiar do armazenamento de dados barato e do processamento de dados rápido com frequência na memória, oferecendo também recursos de consulta SQL. Alguns deles permitem até mesmo que o analista escreva uma única consulta que retorne dados de várias fontes subjacentes. Isso é empolgante para qualquer pessoa que trabalhe com grandes quantidades de dados e confirma que o SQL veio para ficar.

O NoSQL é uma tecnologia que permite que a modelagem de dados não seja estritamente relacional. Ele permite armazenamento e recuperação de latência muito baixa, o que é crítico em muitas aplicações online. A classe inclui o armazenamento de par chave-valor, os bancos de dados de grafos, que executam o armazenamento em um formato nó-borda (node-edge), e os armazenamentos por documentos (document stores). Exemplos desses data stores sobre os quais você pode ouvir falar em sua organização são o Cassandra, o Couchbase, o DynamoDB, o Memcached, o Giraph e o Neo4j. No início, o NoSQl era comercializado como tendo tornado o SQL obsoleto, porém mais recentemente o acrônimo tem sido anunciado como "not only SQL" (não só SQL). Para fins de análise, normalmente usar dados mantidos em um armazenamento de chave-valor NoSQL requer movê-los para um data warehouse SQL tradicional, já que o NoSQL não é otimizado para a consulta de muitos registros ao mesmo tempo. Os bancos de dados de grafos têm aplicações como a análise de rede, e o trabalho de análise pode ser feito diretamente neles com linguagens de consulta especiais. No entanto, o cenário das ferramentas está sempre evoluindo e talvez algum dia também possamos analisar esses dados com SQL.

Os data stores baseados em pesquisa incluem o Elasticsearch e o Splunk. Com frequência, o Elasticsearch e o Splunk são usados para analisar dados gerados por máquina, como os logs. Essas tecnologias e outras semelhantes têm linguagens de consulta não SQL, mas se você souber SQL, deve conseguir entendê-las. Reconhecendo como as habilidades em SQL são comuns, alguns data stores, como o Elasticsearch, adicionaram interfaces de consulta SQL. Essas ferramentas são úteis e poderosas para os casos de uso para os quais foram projetadas, mas geralmente não são apropriadas para os tipos de tarefas de análise que este livro aborda. Como tenho explicado para as pessoas com o passar dos anos, elas são ótimas para encontrar agulhas em palheiros. Contudo, não são tão boas para medir o palheiro propriamente dito.

Independentemente do tipo de banco de dados usado ou de qualquer outra tecnologia de armazenamento de dados, a tendência é clara: até mesmo quando os volumes de dados aumentam e os casos de uso tornam-se mais complexos, o SQL ainda é a ferramenta padrão para o

acesso aos dados. Sua grande base de usuários, a curva de aprendizagem acessível e o poder para tarefas analíticas fazem com que, mesmo que as tecnologias queiram se distanciar do SQL, elas voltem atrás e tentem incorporá-lo.

Conclusão

A análise de dados é uma disciplina empolgante com várias aplicações para empresas e outras organizações. O SQL traz muitos benefícios para o trabalho com dados, principalmente para dados armazenados em um banco de dados. Consultar e analisar dados faz parte do fluxo de trabalho maior de análise, e há vários tipos de data stores com os quais um cientista de dados pode vir a trabalhar. Agora que preparamos o terreno para a análise, o SQL e os data stores, o restante do livro abordará com mais detalhes o uso de SQL na análise. O Capítulo 2 enfocará a preparação de dados, começando com uma introdução aos tipos de dados e então passando para a criação de perfis, a limpeza e a modelagem de dados. Os capítulos 3 a 7 apresentarão aplicações de análise de dados, enfocando a análise de séries temporais, a análise de coorte, a análise de texto, a detecção de anomalias e a análise experimental. O Capítulo 8 abordará técnicas para o desenvolvimento de conjuntos de dados complexos para análise posterior em outras ferramentas. Para concluir, o Capítulo 9 terminará com reflexões sobre como os tipos de análise podem ser combinados para gerar novos insights e listará alguns recursos adicionais que poderão ajudá-lo em sua jornada analítica.

---

1 N.T.: Um padrão de fato é um costume ou convenção que alcançou uma posição dominante por aceitação pública ou forças de mercado.

2 N.T.: A operação de pivotar realizada com o operador PIVOT pega os dados que estão em formato horizontal (linhas) e os coloca em formato vertical (colunas). O operador UNPIVOT faz o inverso.

3 Existem algumas tecnologias mais novas que permitem a execução de consultas SQL com dados armazenados em fontes não relacionais.

4 Ralph Kimball e Margy Ross, *The Data Warehouse Toolkit*, terceira edição (Indianápolis: Wiley, 2013).

CAPÍTULO 2

Preparando os dados para análise

As estimativas de quanto tempo os cientistas gastam preparando seus dados variam, mas é seguro dizer que essa etapa ocupa uma parcela significativa do tempo gasto no trabalho com dados. Em 2014, o *New York Times* relatou (*https://oreil.ly/HX1cO*) que os cientistas de dados passavam de 50% a 80% de seu tempo limpando e manipulando dados. Uma pesquisa de 2016 da CrowdFlower (*https://oreil.ly/5h28Y*) descobriu que os cientistas de dados gastavam 60% de seu tempo limpando e organizando dados para prepará-los para o trabalho de análise ou modelagem. Preparar dados é uma tarefa tão comum que surgiram termos para descrevê-la, como data munging, data wrangling e data prep. ("Mung" é um acrônimo para Mash Until No Good [eliminar tudo que é ruim], o que certamente fiz em algumas ocasiões). Todo esse trabalho de preparação de dados é apenas uma tarefa sem sentido ou é parte importante do processo?

O trabalho de preparação é mais fácil quando um conjunto de dados tem um *dicionário de dados*, um documento ou repositório que apresenta descrições claras dos campos, dos valores possíveis, de como os dados foram coletados e de como eles estão relacionados com outros dados. Infelizmente, com frequência não é isso que ocorre. Geralmente a documentação não é priorizada, até mesmo pelas pessoas que conhecem seu valor, ou fica desatualizada à medida que novos campos e tabelas são adicionados ou quando a maneira de os dados serem fornecidos muda. A criação de perfis de dados gera vários elementos de um dicionário de dados; logo, se sua organização já tiver um dicionário, esse é um bom momento para usá-lo e melhorá-lo. Se atualmente não existir um dicionário de dados, considere iniciar um! Ele é um dos presentes mais valiosos que você pode dar para sua equipe e para você mesmo no futuro. Um dicionário de dados atualizado permitirá que você acelere o processo de criação de perfis de dados baseando-se em perfis já estabelecidos em vez de replicá-los. Também melhorará a qualidade dos resultados da sua

análise, já que você poderá verificar se usou os campos corretamente e se aplicou filtros apropriados.

Mesmo se existir um dicionário de dados, provavelmente você precisará executar o trabalho de preparação de dados como parte da análise. Neste capítulo, começarei com uma descrição dos tipos de dados que você deve encontrar. Ela será seguida de um resumo sobre a estrutura da consulta SQL. Depois falarei sobre a criação de perfis de dados como uma maneira de conhecer o conteúdo e verificar a qualidade dos dados. Em seguida, abordarei algumas técnicas de modelagem de dados que retornarão as colunas e linhas necessárias para a análise posterior. Para concluir, descreverei ferramentas úteis para a limpeza de dados que podem ajudar a lidar com problemas de qualidade.

Tipos de dados

Os dados são a base da análise, e todos eles têm um tipo de dado de banco de dados e também pertencem a uma ou mais categorias de dados. Ter um conhecimento sólido das várias formas que os dados podem assumir o ajudará a ser um analista de dados mais eficiente. Começarei com os tipos de dados de banco de dados encontrados com mais frequência na análise. Em seguida, passarei para alguns agrupamentos conceituais que podem nos ajudar a conhecer a origem, a qualidade e as possíveis aplicações dos dados.

Tipos de dados de banco de dados

Todos os campos das tabelas dos bancos de dados têm tipos de dados definidos. A maioria dos bancos de dados tem uma boa documentação sobre os tipos que suportam, e esse é um bom recurso para a obtenção de algum detalhe necessário que não tiver sido apresentado aqui. Você não precisa ser necessariamente um especialista nas nuances dos tipos de dados para ser bom em análise, mas posteriormente no livro encontraremos situações em que considerar o tipo de dado será importante; logo, esta seção abordará os aspectos básicos. Os principais tipos de dados são as strings, os dados numéricos, os dados lógicos, e os de data/hora (datetime), como resumido na Tabela 2.1. Usei como base o

Postgres, mas há tipos de dados semelhantes em grande parte dos bancos de dados mais importantes.

Os tipos de dados string são os mais versáteis. Eles podem conter letras, números e caracteres especiais, incluindo caracteres não imprimíveis como tabulações e novas linhas. Os campos string podem ser definidos para conter um número de caracteres fixo ou variável. Um campo CHAR poderia ser definido para permitir apenas dois caracteres para abreviações de estados dos Estados Unidos, por exemplo, enquanto um campo que armazenasse os nomes completos dos estados precisaria ser um VARCHAR para permitir um número de caracteres variável. Dependendo do banco de dados, podem ser definidos campos como TEXT, CLOB (Character Large Object) ou BLOB (Binary Large Object, que pode incluir tipos de dados adicionais, como imagens) para conter strings mais longas, mas já que geralmente eles ocupam muito espaço, esses tipos de dados tendem a ser usados moderadamente. Quando os dados forem carregados, se chegarem strings muito grandes para o tipo de dado definido, elas podem ser truncadas ou totalmente rejeitadas. O SQL tem várias funções de string que usaremos para diferentes fins analíticos.

Tabela 2.1: Resumo dos tipos de dados de banco de dados comuns

Tipo	Nome	Descrição
String	CHAR / VARCHAR	Contém strings. Um CHAR tem sempre tamanho fixo, enquanto um VARCHAR tem tamanho variável, indo até algum tamanho máximo (256 caracteres, por exemplo).
	TEXT / BLOB	Contém strings mais longas que não cabem em um VARCHAR. Descrições de texto livre inseridas por participantes de pesquisas podem ser armazenadas nesses campos.
Numérico	INT / SMALLINT / BIGINT	Contém inteiros (números inteiros). Alguns bancos de dados têm o tipo SMALLINT e/ou BIGINT. SMALLINT pode ser usado para um campo que só tiver valores com um número pequeno de dígitos. SMALLINT usa menos memória do que um INT comum. BIGINT pode conter números com mais dígitos do que INT, mas ocupa mais espaço do que ele.

Tipo	Nome	Descrição
	FLOAT / DOUBLE / DECIMAL	Contém números decimais, às vezes com o número de casas decimais sendo especificado.
Lógico	BOOLEAN	Contém os valores TRUE (verdadeiro) ou FALSE (falso).
Data/hora	DATETIME / TIMESTAMP	Contém as datas com as horas. Normalmente é usado um formato AAAA-MM-DD hh:mi:ss, em que AAAA é o ano com quatro dígitos, MM é o mês com dois dígitos, DD é o dia com dois dígitos, hh é a hora com dois dígitos (geralmente a hora no intervalo de 24 horas, ou valores de 0 a 23), mi são os minutos com dois dígitos e ss são os segundos com dois dígitos. Alguns bancos de dados armazenam apenas timestamps (carimbos de data/hora) sem o fuso horário, enquanto outros têm tipos específicos com e sem os fusos horários.
	TIME	Armazena a hora.

Os tipos de dados numéricos são todos os que armazenam números, tanto positivos quanto negativos. Funções e operadores matemáticos podem ser aplicados aos campos numéricos. Eles incluem os tipos INT assim como os tipos FLOAT, DOUBLE e DECIMAL que permitem casas decimais. Os tipos de dados inteiros é que costumam ser implementados porque usam menos memória do que suas contrapartidas decimais. Em alguns bancos de dados, como o Postgres, a divisão de inteiros resulta em um inteiro, em vez de em um valor com casas decimais como seria esperado. Discutiremos a conversão de tipos de dados numéricos para a obtenção de resultados corretos posteriormente neste capítulo.

O tipo de dado lógico chama-se BOOLEAN. Ele tem os valores TRUE e FALSE e é uma maneira eficiente de armazenar informações em casos em que essas opções sejam apropriadas. As operações que comparam dois campos retornam um valor BOOLEAN como resultado. Esse tipo de dado costuma ser usado na criação de *flags*, campos que resumem a presença ou a ausência de uma propriedade nos dados. Por exemplo, uma tabela armazenando dados de email poderia ter um campo BOOLEAN `has_opened`.

Os tipos de data/hora incluem DATE, TIMESTAMP e TIME. Os dados de data e hora devem ser armazenados em um campo de um desses tipos sempre que possível, já que o SQL tem várias funções úteis que operam com eles. Os timestamps e as datas são muito comuns em bancos de dados e são críticos para vários tipos de análise, principalmente de séries temporais (abordadas no Capítulo 3) e de coorte (abordadas no Capítulo 4). O Capítulo 3 discutirá a formatação, as transformações e os cálculos de data e hora.

Outros tipos de dados, como os tipos JSON e geográficos, são suportados por alguns bancos de dados, mas não por todos. Não darei detalhes deles aqui porque esses tipos não fazem parte do escopo deste livro. No entanto, eles são um sinal de que o SQL continua a evoluir para manipular novas tarefas de análise.

Além dos tipos de dados de banco de dados, há várias maneiras conceituais de categorização dos dados. Elas podem ter impacto tanto sobre como os dados serão armazenados quanto sobre como os analisaremos. Discutiremos esses tipos de dados categóricos a seguir.

Estruturados versus não estruturados

Com frequência os dados são descritos como estruturados ou não estruturados, ou às vezes como semiestruturados. A maioria dos bancos de dados foi projetada para manipular *dados estruturados*, caso em que cada atributo é armazenado em uma coluna e as instâncias de cada entidade são representadas como linhas. Primeiro um modelo de dados é criado e, em seguida, os dados são inseridos de acordo com esse modelo. Por exemplo, uma tabela de endereços poderia ter campos para o endereço residencial, a cidade, o estado e o código postal. Cada linha conteria o endereço de um cliente específico. Cada campo teria um tipo de dado e só permitiria que dados desse tipo fossem inseridos. Quando dados estruturados são inseridos em uma tabela, cada campo é verificado para que seja confirmado se ele está em conformidade com o tipo de dado correto. É fácil consultar dados estruturados com SQL.

Os *dados não estruturados* são o oposto dos dados estruturados. Não há estrutura, modelo de dados ou tipos de dados predeterminados. Com frequência os dados não estruturados são aquele "excedente" que não são

dados do banco de dados. Documentos, emails e páginas web são dados não estruturados. Fotos, imagens, vídeos e arquivos de áudio também são exemplos de dados não estruturados. Eles não se encaixam nos tipos de dados tradicionais e, portanto, é mais difícil serem armazenados com eficiência pelos bancos de dados relacionais e consultados com SQL. Como resultado, os dados não estruturados costumam ser armazenados fora de bancos de dados relacionais. Isso permite que os dados sejam carregados rapidamente, mas a falta de validação pode resultar em baixa qualidade dos dados. Como vimos no Capítulo 1, a tecnologia continua a evoluir e novas ferramentas estão sendo desenvolvidas para permitir a consulta SQL de muitos tipos de dados não estruturados.

Os *dados semiestruturados* estão entre essas duas categorias. Muitos dados "não estruturados" têm alguma estrutura que podemos usar. Por exemplo, os emails têm endereços de remetente e de destinatário, linhas de assunto, texto do corpo, e timestamps de envio que podem ser armazenados separadamente em um modelo de dados com esses campos. Metadados, ou dados sobre dados, podem ser extraídos de outros tipos de arquivo e armazenados para análise. Por exemplo, arquivos de áudio musical podem ter tags de artista, nome da canção, gênero e duração. Geralmente, as partes estruturadas dos dados semiestruturados podem ser consultadas com SQL, e este talvez possa ser usado para extrair ou fazer o parsing de dados estruturados para uma consulta posterior. Veremos algumas aplicações disso na discussão de análise de texto no Capítulo 5.

Dados quantitativos versus qualitativos

Os *dados quantitativos* são numéricos. Eles medem pessoas, coisas e eventos. Podem incluir descritores, como informações do cliente, o tipo de produto ou configurações de dispositivo, mas também vêm com informações numéricas como preço, quantidade ou duração da visita. Contagens, somas, média ou outras funções numéricas são aplicadas aos dados. Atualmente, com frequência os dados quantitativos são gerados por máquina, mas isso não é obrigatório. A altura, o peso e a pressão sanguínea registrados em um formulário de papel para a admissão de um paciente são quantitativos, assim como o são os pontos obtidos por estudantes em um questionário e digitados em uma planilha por um

professor.

Geralmente os *dados qualitativos* são baseados em texto e incluem opiniões, percepções e descrições que não são estritamente quantitativas. Temperatura e níveis de umidade são quantitativos, enquanto descritores como "quente e úmido" são qualitativos. O preço que um cliente pagou por um produto é quantitativo; o fato de ter gostado ou não do produto é qualitativo. O feedback de uma pesquisa, perguntas feitas ao suporte ao cliente e postagens de mídia social são qualitativos. Há profissões apenas para a manipulação de dados qualitativos. No contexto da análise de dados, normalmente tentamos quantificar o que é qualitativo. Uma técnica é extrair palavras-chave ou expressões e contar suas ocorrências. Examinaremos isso com mais detalhes quando nos aprofundarmos na análise de texto no Capítulo 5. Outra técnica é a análise de sentimentos, na qual a estrutura da linguagem é usada na interpretação do significado das palavras usadas, além de sua frequência. Frases ou outros corpos de texto podem ser pontuados por seu nível de positividade ou negatividade e, em seguida, contagens ou médias são usadas para derivar insights que de outra forma seriam difíceis de resumir. Houve avanços empolgantes na área do processamento de linguagem natural, ou NLP (natural language processing), embora grande parte desse trabalho seja feita com ferramentas como o Python.

Dados primários, secundários e de terceiros

Os dados primários (*first-party data*) são coletados pela própria organização. Isso pode ser feito por meio de logs de servidor, bancos de dados que registram transações e informações do cliente, ou outros sistemas que são construídos e controlados pela organização e geram dados de interesse para análise. Já que os sistemas foram criados internamente, costuma ser possível encontrar as pessoas que os construíram e saber como os dados são gerados. Os analistas de dados talvez também possam influenciar ou ter controle sobre como certos dados são criados e armazenados, principalmente quando bugs forem responsáveis por uma qualidade de dados insatisfatória.

Os dados secundários (*second-party data*) vêm de fornecedores que oferecem um serviço ou plataforma ou que executam uma atividade empresarial em

nome da organização. Com frequência envolvem produtos SaaS (software as a service, de software como serviço); alguns exemplos comuns seriam o CRM, as ferramentas de automação de email e marketing, softwares de habilitação de e-commerce, e rastreadores de interação na web e móvel. Os dados são semelhantes aos first-party já que são inerentes à organização e criados por seus funcionários e clientes. Contudo, tanto o código que gera e armazena os dados quanto o modelo de dados são controlados externamente, e normalmente o analista de dados tem pouca influência sobre esses aspectos. Os dados second-party estão cada vez mais sendo importados para o data warehouse da organização para análise. Isso pode ser feito com código personalizado, com conectores ETL ou com fornecedores de SaaS que oferecem integração de dados.

Muitos fornecedores de SaaS oferecem recursos de relatório; logo, pode surgir a dúvida de se seria necessário copiar os dados em um data warehouse. Um departamento que interaja com uma ferramenta pode achar o relatório suficiente, como um departamento de atendimento ao cliente que gere relatórios sobre o tempo de resolução de problemas e a produtividade dos agentes a partir de seu software de central de ajuda. Por outro lado, as interações do atendimento ao cliente podem ser uma entrada importante para um modelo de retenção de clientes, o que demandaria integrar esses dados a um data store com dados de vendas e cancelamento. Aqui está uma regra de ouro para a decisão de se você deve importar dados de uma fonte de dados específica: se os dados gerarem valor quando combinados com dados de outros sistemas, importe-os; se não gerarem, espere até que haja uma situação de maior necessidade.

Os dados de terceiros (*third-party data*) podem ser comprados ou obtidos de fornecedores gratuitos como os publicados pelos governos. A menos que tenham sido coletados especificamente em nome da organização, geralmente as equipes de dados têm pouco controle sobre o formato, a frequência e a qualidade dos dados. Esses dados geralmente não têm a granularidade dos dados first-party e second-party. Por exemplo, a maioria das fontes de dados third-party não tem dados de nível de usuário e em vez disso os dados podem ser acrescidos aos dados first-party no nível do código postal, da cidade ou em um nível mais alto. No entanto, os dados third-party podem ter informações exclusivas e úteis, como padrões de gasto agregado, dados demográficos e tendências de

mercado, cuja coleta de outra forma seria muito cara ou impossível.

Dados esparsos

Os *dados esparsos* ocorrem quando há uma quantidade pequena de informações dentro de um conjunto maior de informações ausentes ou irrelevantes. Eles podem aparecer como uma quantidade maior de nulos e somente alguns valores em uma coluna específica. O valor nulo, que é diferente do valor 0, é a *ausência* de dados; isso será abordado posteriormente na seção sobre limpeza de dados. Os dados esparsos podem ocorrer quando eventos forem raros, como erros de software ou compras de produtos da cauda longa (long tail) de um catálogo. Também podem ocorrer nos dias iniciais do lançamento de um recurso ou produto, quando só os testadores ou os clientes beta têm acesso. O JSON é uma abordagem desenvolvida para lidar com dados esparsos do ponto de vista da gravação e armazenamento, já que ele só armazena os dados que estão presentes e omite o restante. Isso é o oposto do que ocorre em um banco de dados relacional, que precisa ter memória para um campo mesmo se não houver valor nele.

Os dados esparsos podem ser problemáticos para a análise. Quando eventos são raros, as tendências não são necessariamente significativas e pode ser difícil fazer a distinção entre correlações e flutuações ocasionais. É útil criar perfis de dados, como discutido posteriormente neste capítulo, para saber se e onde seus dados são esparsos. Algumas opções são agrupar eventos ou itens pouco frequentes em categorias que sejam mais comuns, excluir totalmente os dados esparsos ou o período de tempo da análise, ou exibir uma estatística descritiva junto com explicações de alerta de que as tendências não são necessariamente significativas.

Há muitos tipos de dados e diversas maneiras de os dados serem descritos, muitas das quais são coincidentes ou não mutuamente exclusivas. É útil familiarizar-se com esses tipos não só para a criação de uma boa consulta SQL, mas também para a decisão de como analisar os dados de maneiras apropriadas. Nem sempre conhecemos os tipos de dados com antecedência, e é por isso que a criação de perfis de dados é tão crítica. Antes de examinarmos isso, e nossos primeiros exemplos de código, fornecerei uma breve explicação da estrutura da consulta SQL.

Estrutura da consulta SQL

As consultas SQL têm cláusulas e sintaxe comuns, embora elas possam ser combinadas em um número quase infinito de maneiras para atingirmos os objetivos da análise. Este livro presume que você tenha algum conhecimento anterior de SQL, mas explicarei os principais aspectos aqui para termos uma base comum para os exemplos de código que veremos.

A cláusula *SELECT* determina as colunas que serão retornadas pela consulta. Uma coluna será retornada para cada expressão da cláusula *SELECT*, e as expressões são separadas por vírgulas. Uma expressão pode ser um campo da tabela, uma agregação como `sum`, ou qualquer número de cálculos, como instruções CASE, conversões de tipo, e várias funções que serão discutidas posteriormente neste capítulo e no decorrer do livro.

A cláusula *FROM* determina as tabelas das quais as expressões da cláusula *SELECT* são derivadas. A "tabela" pode ser uma tabela de banco de dados, uma view (um tipo de consulta salva que também funciona como uma tabela) ou uma subconsulta. Uma subconsulta é ela própria uma consulta, colocada entre parênteses, e o resultado é tratado como qualquer outra tabela pela consulta que a referencia. Uma consulta pode referenciar várias tabelas na cláusula *FROM*, mas elas devem usar apenas um dos tipos de *JOIN* junto com uma condição que especifique como as tabelas estão relacionadas. Geralmente a condição de *JOIN* especifica uma igualdade entre os campos de cada tabela, como `orders.customer_id = customers.customer_id`. As condições de *JOIN* podem incluir vários campos e também podem especificar desigualdades ou intervalos de valores, como intervalos de datas. Veremos várias condições de *JOIN* que atingem objetivos de análise específicos no decorrer do livro. Uma *INNER JOIN* retorna todos os registros coincidentes de duas tabelas. Uma *LEFT JOIN* retorna todos os registros da primeira tabela e apenas os da segunda tabela que sejam coincidentes. Uma *RIGHT JOIN* retorna todos os registros da segunda tabela e apenas os da primeira tabela que sejam coincidentes. Uma *FULL OUTER JOIN* retorna todos os registros das duas tabelas. Uma *JOIN* Cartesiana pode ocorrer quando cada registro da primeira tabela coincide com mais de um registro da segunda tabela. Geralmente as *JOIN*s Cartesianas devem ser evitadas, embora haja alguns casos de uso

específicos, como a geração de dados para preencher uma série temporal, em que elas são usadas intencionalmente. Para concluir, as tabelas da cláusula *FROM* podem ganhar *aliases*, ou receber um nome mais curto de uma ou mais letras que possa ser referenciado em outras cláusulas da consulta. Os aliases evitam que os criadores de consultas precisem digitar longos nomes de tabelas repetidamente e tornam as consultas mais fáceis de ler.

Embora tanto *LEFT JOIN* quanto *RIGHT JOIN* possam ser usadas na mesma consulta, será muito mais fácil acompanhar sua lógica se você adotar apenas uma ou outra. Na prática, *LEFT JOIN* é muito mais usada do que *RIGHT JOIN*.

A cláusula *WHERE* especifica restrições ou filtros necessários para a exclusão ou a remoção de linhas do conjunto de resultados. *WHERE* é opcional.

A cláusula *GROUP BY* é necessária quando a cláusula *SELECT* contém agregações e pelo menos um campo não agregado. Uma maneira fácil de saber o que deve entrar na cláusula *GROUP BY* é lembrar-se de que ela deve ter todos os campos que não fazem parte de uma agregação. Na maioria dos bancos de dados, há duas maneiras de listar os campos de *GROUP BY*: pelo nome do campo ou por posição, como 1, 2, 3 e assim por diante. Algumas pessoas preferem usar a notação de nome de campo, e o SQL Server exige isso. Prefiro a notação posicional, principalmente quando os campos de *GROUP BY* contêm expressões complexas ou quando estou fazendo muitas iterações. Este livro usará a notação posicional.

Como não travar seu banco de dados: LIMIT e amostragem

As tabelas de banco de dados podem ser muito grandes, contendo milhões ou bilhões de registros. Fazer uma consulta em todos esses registros pode no mínimo causar problemas e no máximo travar os bancos de dados. Para não receber chamadas raivosas dos administradores de banco de dados ou ser bloqueado, é uma boa ideia limitar os resultados retornados durante a criação de perfis ou ao testar consultas. As cláusulas *LIMIT* e a amostragem são duas técnicas que devem fazer parte de sua caixa de ferramentas.

LIMIT é adicionada como a última linha da consulta, ou subconsulta, e pode receber qualquer valor inteiro positivo:

```
SELECT column_a, column_b
FROM table
LIMIT 1000
;
```

Quando usado em uma subconsulta, o limite será aplicado a essa etapa, e só o conjunto de resultados restrito será avaliado pela consulta externa:

```
SELECT...
FROM
(
    SELECT column_a, column_b, sum(sales) as total_sales
    FROM table
    GROUP BY 1,2
    LIMIT 1000
) a
;
```

O SQL Server não suporta a cláusula *LIMIT*, mas um resultado semelhante pode ser obtido com o uso de `top`:

```
SELECT top 1000
column_a, column_b
FROM table
;
```

A amostragem pode ser obtida com o uso de uma função em um campo ID que tenha uma distribuição aleatória de dígitos no começo ou no fim. A função de módulo ou `mod` retorna o resto quando um inteiro é dividido por outro. Se o campo ID for um inteiro, `mod` poderá ser usada para encontrar o último dígito, os dois últimos dígitos ou mais dígitos finais e filtrar o resultado:

```
WHERE mod(integer_order_id,100) = 6
```

Essa linha retornará todos os pedidos nos quais os dois últimos dígitos são 06, o que deve resultar em cerca de 1% do total. Se o campo for alfanumérico, você poderá usar uma função `right()` para encontrar um número específico de dígitos no final:

```
WHERE right(alphanum_order_id,1) = 'B'
```

Essa linha retornará todos os pedidos em que o último dígito é B, o que resultará em cerca de 3% do total se todas as letras e números forem igualmente comuns, uma suposição que é útil validar.

Limitar o conjunto de resultados também acelera o trabalho, mas lembre-se de que os subconjuntos de dados podem não conter todas as variações de valores e casos extremos que existem no conjunto de dados completo. Não se esqueça de

remover a cláusula *LIMIT* ou a amostragem antes de fazer sua análise ou relatório final com a consulta, ou acabará obtendo resultados estranhos!

Abordamos os aspectos básicos da estrutura da consulta SQL. O Capítulo 8 dará mais detalhes sobre cada uma dessas cláusulas, sobre algumas cláusulas adicionais que são encontradas com menos frequência, mas aparecem neste livro, e sobre a ordem na qual cada cláusula é avaliada. Agora que temos essa base, podemos passar para uma das partes mais importantes do processo de análise: a criação de perfis de dados.

Criação de perfis: distribuições

A criação de perfis é a primeira tarefa que executo quando começo a trabalhar com qualquer conjunto de dados novo. Examino como os dados estão organizados em esquemas e tabelas. Verifico os nomes das tabelas para me familiarizar com os tópicos abordados, como clientes, pedidos ou visitas. Vejo os nomes das colunas em algumas tabelas e começo a construir um modelo mental de como as tabelas se relacionam umas com as outras. Por exemplo, poderia haver uma tabela `order_detail` (detalhe do pedido) com vários itens de linha se relacionando com a tabela `order` (pedido) por meio de um `order_id` (identificador de pedido), enquanto com a tabela `customer` (cliente), a tabela `order` poderia se relacionar por meio de um `customer_id` (identificador de cliente). Quando há um dicionário de dados, eu o examino e o comparo com os dados vistos em uma amostragem das linhas.

Geralmente as tabelas representam as operações de uma organização, ou algum subconjunto das operações; logo, penso sobre que área ou subárea está sendo abordada, como e-commerce, marketing ou interações com produtos. É mais fácil trabalhar com dados quando temos conhecimento de como eles foram gerados. A criação de perfis pode fornecer pistas sobre isso, ou sobre que perguntas fazer à fonte, ou às pessoas de dentro ou de fora da organização responsáveis pela coleta ou pela geração dos dados. Mesmo quando você coletar os dados por conta própria, a criação de perfis será útil.

Outro detalhe que procuro é como a história é representada, caso o seja. Conjuntos de dados que são réplicas de bancos de dados de produção

podem não conter os valores anteriores de endereços de clientes ou status de pedidos, por exemplo, enquanto um data warehouse bem construído pode ter snapshots diários de campos de dados variáveis.

A criação de perfis de dados está relacionada ao conceito de *análise exploratória de dados*, ou EDA (exploratory data analysis), nome dado por John Tukey. Em seu livro, que tem esse nome,[1] Tukey descreve como analisar conjuntos de dados calculando vários resumos e visualizando os resultados. Ele demonstra técnicas de verificação de distribuições de dados, incluindo diagramas de ramos e folhas, diagramas de caixas (box plots) e histogramas.

Após verificar algumas amostras de dados, começo a examinar as distribuições. Elas permitem conhecer o intervalo de valores que existe nos dados e com que frequência eles ocorrem, se há valores nulos, e se existem valores negativos junto com os positivos. As distribuições podem ser criadas com dados contínuos ou categóricos e também são chamadas de frequências. Nesta seção, examinaremos como criar histogramas, como a discretização (binning) pode nos ajudar a entender a distribuição de valores contínuos e como usar funções n-tiles para aumentar a precisão das distribuições.

Histogramas e frequências

Uma das melhores maneiras de conhecer um conjunto de dados, e de conhecer campos específicos do conjunto, é verificar a frequência de valores em cada campo. As verificações de frequência também serão úteis sempre que você tiver alguma dúvida sobre se certos valores são possíveis ou se detectar um valor inesperado e quiser saber com que constância ele ocorre. Elas podem ser feitas em qualquer tipo de dado, incluindo strings, dados numéricos, datas e booleanos. As consultas de frequência também são uma ótima maneira de detectar dados esparsos.

A consulta é simples. O número de linhas pode ser encontrado com `count(*)`, e o campo do perfil está em *GROUP BY*. Por exemplo, poderíamos verificar a frequência de cada tipo de fruta (`fruit`) em uma tabela fictícia `fruit_inventory`:

```
SELECT fruit, count(*) as quantity
```

```
FROM fruit_inventory
GROUP BY 1
;
```

Antes de usar `count`, é aconselhável considerar se pode haver algum registro duplicado no conjunto de dados. Você pode usar `count(*)` quando quiser o número de registros, mas use `count distinct` para descobrir quantos itens únicos existem.

Um *diagrama de frequências* é uma maneira de visualizar quantas vezes algo ocorre no conjunto de dados. Geralmente o campo cujo perfil está sendo criado é representado no eixo x, com a contagem de observações aparecendo no eixo y. A Figura 2.1 mostra um exemplo de diagramação da frequência das frutas de nossa consulta (apples, bananas, oranges – maças, bananas, laranjas). Os gráficos de frequências também podem ser desenhados horizontalmente, o que ajuda a acomodar nomes de valores longos. Lembre-se de que esses são dados categóricos sem nenhuma ordem inerente.

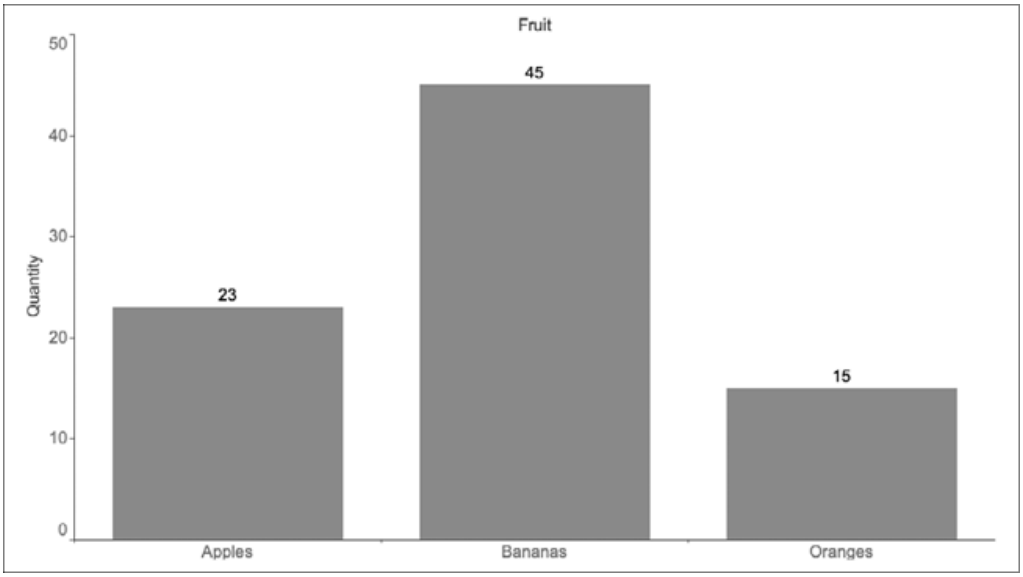


Figura 2.1: Diagrama de frequências do estoque de frutas.

Um *histograma* é uma maneira de visualizar a distribuição de valores numéricos em um conjunto de dados e será familiar para quem tiver experiência em estatística. Um histograma básico poderia mostrar a distribuição de faixas etárias em um grupo de clientes. Suponhamos que tivéssemos uma tabela `customers` (clientes) contendo nomes, a data de

cadastramento, a idade e outros atributos. Para criar um histograma por idade, faça o agrupamento (com *GROUP BY*) pelo campo numérico `age` (idade) e conte as ocorrências de `customer_id`:

```
SELECT age, count(customer_id) as customers
FROM customers
GROUP BY 1
;
```

Os resultados de nossa distribuição fictícia de idades estão representados na Figura 2.2.

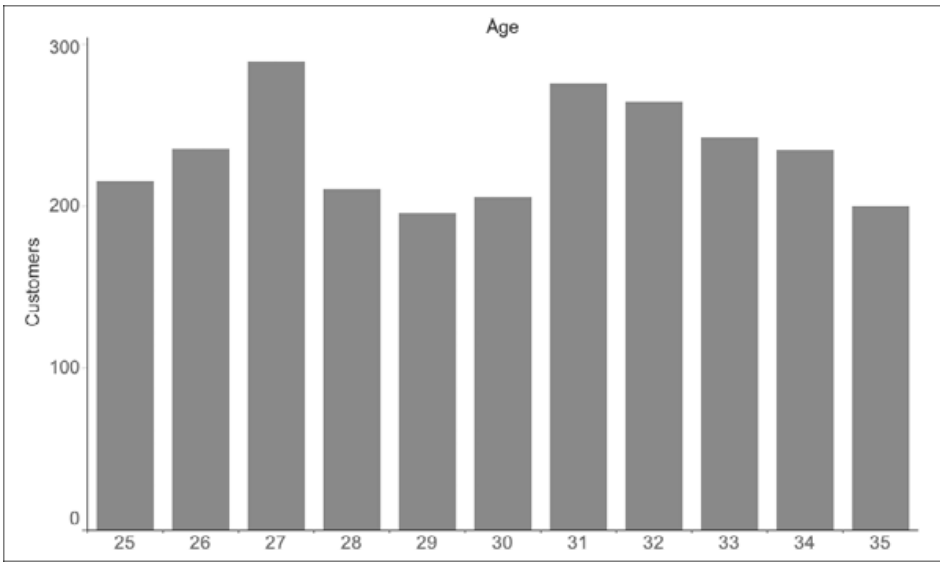


Figura 2.2: Clientes por idade.

Outra técnica que usei muito e que se tornou a base para uma de minhas perguntas de entrevista favoritas envolve uma agregação seguida de uma contagem de frequência. Forneço para os candidatos uma tabela fictícia chamada `orders` (pedidos), que tem uma data, o identificador do cliente (`customer_id`), o identificador do pedido (`order_id`) e uma quantidade, e, em seguida, peço a eles que escrevam uma consulta SQL que retorne a distribuição de pedidos por cliente. Isso não pode ser resolvido com uma simples consulta; requer uma etapa de agregação intermediária, que pode ser executada com uma subconsulta. Primeiro, conte o número de pedidos feitos por cada `customer_id` na subconsulta. A consulta externa usará o número de pedidos como categoria e contará o número de clientes:

```
SELECT orders, count(*) as num_customers
FROM
(
    SELECT customer_id, count(order_id) as orders
    FROM orders
    GROUP BY 1
) a
GROUP BY 1
;
```

Esse tipo de criação de perfil pode ser aplicado sempre que você precisar ver com que frequência certas entidades ou atributos aparecem nos dados. Nesses exemplos, `count` foi usada, mas as outras agregações básicas (`sum`, `avg`, `min` e `max`) também podem ser usadas para criar histogramas. Por exemplo, poderíamos criar perfis de clientes pela soma (`sum`) de seus pedidos, pelo tamanho médio (`avg`) do pedido e pela data mínima (`min`) ou máxima (`max`) (mais recente) do pedido.

Discretização (Binning)

A discretização (binning) é útil no trabalho com valores contínuos. Em vez de ser usado o número de observações ou registros de cada valor que está sendo contado, intervalos de valores são agrupados, e esses grupos são chamados de *bins* (compartimentos) ou *buckets*. O número de registros contido em cada intervalo é então contado. Os bins podem ter tamanho variável ou fixo, dependendo de se o objetivo for agrupar os dados em bins que tenham um significado específico para a organização, que tenham largura aproximadamente igual ou que contenham números de registros semelhantes. Eles podem ser criados com instruções CASE, arredondamento e logaritmos.

A instrução CASE permite que uma lógica condicional seja avaliada. Essas instruções são muito flexíveis e voltaremos a vê-las no decorrer do livro, aplicando-as à criação de perfis e à limpeza de dados, à análise de texto e a várias outras situações. A estrutura básica da instrução CASE é:

```
case when condition1 then return_value_1
     when condition2 then return_value_2
     ...
```

```
    else return_value_default
    end
```

A condição WHEN pode ser uma igualdade, uma diferença ou outra condição lógica. O valor de retorno de THEN pode ser uma constante, uma expressão, ou um campo da tabela. Qualquer número de condições pode ser incluído, mas a instrução parará de ser executada e retornará o resultado na primeira vez em que uma condição for avaliada como TRUE. ELSE informa ao banco de dados o que usar como valor padrão se nenhuma correspondência for encontrada e também pode ser uma constante ou um campo. ELSE é opcional, e se não for incluída, a falta de ocorrência de correspondências retornará nulo. As instruções CASE também podem ser aninhadas para que o valor de retorno seja outra instrução CASE.

 Todos os valores de retorno que vêm após THEN devem ter o mesmo tipo de dado (strings, numérico, BOOLEAN etc.) ou haverá um erro. Considere fazer a conversão para um tipo de dado comum como uma string se isso ocorrer.

Uma instrução CASE é uma maneira flexível de controlar o número de bins, o intervalo de valores contido em cada bin e como os bins serão nomeados. Considero-as úteis principalmente quando há uma cauda longa de valores muito pequenos ou muito grandes e acho melhor agrupá-los em vez de deixar bins vazios em parte da distribuição. Certos intervalos de valores têm um significado empresarial que precisa ser recriado nos dados. Várias empresas B2B separam seus clientes nas categorias "enterprise" (empresa) e "SMB" (small- and medium-sized businesses, negócios pequenos e médios) de acordo com o número de funcionários ou com a receita, porque seus padrões de compra são diferentes. Como exemplo, suponhamos que estivéssemos considerando fornecer ofertas com desconto na entrega e quiséssemos saber quantos clientes serão afetados. Poderíamos agrupar `order_amount` (quantidade de pedidos) em três buckets usando uma instrução CASE:

```
SELECT
case when order_amount <= 100 then 'up to 100'
     when order_amount <= 500 then '100 - 500'
     else '500+' end as amount_bin
,case when order_amount <= 100 then 'small'
```

```
      when order_amount <= 500 then 'medium'
      else 'large' end as amount_category
,count(customer_id) as customers
FROM orders
GROUP BY 1,2
;
```

Bins de tamanho arbitrário podem ser úteis, mas em certas situações bins de tamanho fixo são mais apropriados para a análise. Bins de tamanho fixo podem ser obtidos de algumas maneiras, que incluem arredondamento, logaritmos e funções ntile. Para a criação de bins com a mesma largura, o arredondamento é útil. Ele reduz a precisão dos valores, e geralmente pensamos em usá-lo para reduzir o número de casas decimais ou removê-las totalmente pelo arredondamento para o próximo inteiro. A função `round` tem a forma a seguir:

```
round(value,number_of_decimal_places)
```

O número de casa decimais também pode ser negativo, permitindo que essa função faça o arredondamento para as dezenas, centenas, milhares etc., mais próximos. A Tabela 2.2 demonstra os resultados do arredondamento com argumentos que variam de –3 a 2.

Tabela 2.2: O número 123.456,789 arredondado com várias casas decimais

Casas decimais	Fórmula	Resultado
2	`round(123456.789,2)`	`123456.79`
1	`round(123456.789,1)`	`123456.8`
0	`round(123456.789,0)`	`123457`
-1	`round(123456.789,-1)`	`123460`
-2	`round(123456.789,-2)`	`123500`
-3	`round(123456.789,-3)`	`123000`

```
SELECT round(sales,-1) as bin
,count(customer_id) as customers
FROM table
GROUP BY 1
;
```

Os logaritmos são outra maneira de criar bins, principalmente em

conjuntos de dados em que os valores mais altos tenham ordem de magnitude maior do que os valores menores. A distribuição de renda familiar, o número de visitantes de sites em diferentes propriedades na internet[2] e a intensidade do tremor causado por terremotos são exemplos de fenômenos que têm essa característica. Embora eles não criem bins com a mesma largura, os logaritmos geram bins que aumentam de tamanho com um padrão útil. Para refrescar sua memória, um logaritmo é o expoente ao qual 10 deve ser elevado para produzir esse número:

```
log(número) = expoente
```

Nesse caso, 10 é chamado de base, e geralmente essa é a implementação padrão nos bancos de dados, mas tecnicamente a base pode ser qualquer número. A Tabela 2.3 mostra os logaritmos de várias potências de 10.

Tabela 2.3: Resultados da função log em potências de 10

Fórmula	Resultado
`log(1)`	0
`log(10)`	1
`log(100)`	2
`log(1000)`	3
`log(10000)`	4

Em SQL, a função `log` retorna o logaritmo de seu argumento, que pode ser uma constante ou um campo:

```
SELECT log(sales) as bin
,count(customer_id) as customers
FROM table
GROUP BY 1
;
```

A função `log` pode ser usada com qualquer valor positivo, e não apenas com múltiplos de 10. No entanto, a função logarítmica não funcionará se os valores forem menores ou iguais a 0; ela retornará nulo ou um erro, dependendo do banco de dados.

Funções n-tiles

Você deve estar familiarizado com a *mediana*, ou o valor médio, de um

conjunto de dados. Ela é o valor de percentil 50°. Metade dos valores é maior do que a mediana e a outra metade é menor. No caso dos quartis, temos os valores dos percentis 25° e 75°. Um quarto dos valores é menor e três quartos são maiores para o percentil 25°; três quartos são menores e um quarto é maior no percentil 75.° Os decis dividem o conjunto de dados em 10 partes iguais. Generalizando esse conceito, as funções *n-tile* permitem calcular qualquer percentil do conjunto de dados: o percentil 27°, o percentil 50,5° e assim por diante.

Funções de janela

As funções ntile fazem parte de um grupo de funções SQL chamadas funções de janela ou analíticas. Ao contrário da maioria das funções SQL, que só podem operar com a linha de dados atual, as funções de janela fazem cálculos que se estendem por várias linhas. As funções de janela têm uma sintaxe especial que inclui o nome da função e uma cláusula *OVER* que é usada para determinar as linhas incluídas na operação e a ordem dessas linhas. O formato geral de uma função de janela é:

```
function(campo_nome) over (partition by campo_nome order by
  campo_nome)
```

A função pode ser qualquer uma das agregações comuns (`count`, `sum`, `avg`, `min`, `max`) assim como várias funções especiais, incluindo `rank`, `first_value` e `ntile`. A cláusula *PARTITION BY* pode incluir zero ou mais campos. Quando nenhum campo é especificado, a função opera na tabela inteira, mas, quando um ou mais campos são especificados, ela opera somente nessa seção de linhas. Por exemplo, poderíamos usar *PARTITION BY* com um `customer_id` para fazer cálculos de todos os registros por cliente, reiniciando o cálculo para cada cliente. A cláusula *ORDER BY* determina a ordem das linhas para funções que precisem disso; por exemplo, para classificar (com `rank`) os clientes, precisamos especificar um campo de acordo com o qual eles serão ordenados, como o número de pedidos. Todos os principais tipos de bancos de dados têm funções de janela, exceto pelas versões do MySQL anteriores à 8.0.2. Veremos essas funções úteis no decorrer do livro, junto com explicações adicionais de como elas funcionam e como definir os argumentos corretamente.

Muitos bancos de dados têm uma função `median` interna, mas usam as funções mais genéricas ntile para o restante. Elas são funções de janela que fazem cálculos com um intervalo de linhas para retonar um valor para uma única linha. Essas funções recebem um argumento que especifica o número de bins dentro dos quais os dados serão divididos e,

opcionalmente, usam uma cláusula *PARTITION BY* e/ou *ORDER BY*:

```
ntile(num_bins) over (partition by... order by...)
```

Como exemplo, suponhamos que tivéssemos 12 transações com valores de pedidos (`order_amount`) de $19.99, $9.99, $59.99, $11.99, $23.49, $55.98, $12.99, $99.99, $14.99, $34.99, $4.99 e $89.99. A execução de um cálculo com `ntile` envolvendo 10 bins classificará cada `order_amount` e atribuirá um bin de 1 a 10:

```
order_amount  ntile
------------  -----
4.99          1
9.99          1
11.99         2
12.99         2
14.99         3
19.99         4
23.49         5
34.99         6
55.98         7
59.99         8
89.99         9
99.99         10
```

Na prática, essa operação poderia ser usada para agrupar registros em bins, primeiro calculando a função `ntile` para cada linha em uma subconsulta e, em seguida, incluindo-a em uma consulta externa que use `min` e `max` para encontrar os limites superior e inferior do intervalo de valores:

```
SELECT ntile
,min(order_amount) as lower_bound
,max(order_amount) as upper_bound
,count(order_id) as orders
FROM
(
    SELECT customer_id, order_id, order_amount
    ,ntile(10) over (order by order_amount) as ntile
    FROM orders
) a
GROUP BY 1
```

```
;
```

Uma função relacionada é `percent_rank`. Em vez de retornar os bins nos quais os dados serão incluídos, `percent_rank` retorna o percentil. Ela não recebe argumentos, mas requer parênteses e opcionalmente usa uma cláusula *PARTITION BY* e/ou *ORDER BY*:

```
percent_rank() over (partition by... order by...)
```

Embora não seja tão útil quanto `ntile` para o binning, `percent_rank` pode ser usada para criar uma distribuição contínua ou como saída para um relatório ou análise posterior. Tanto `ntile` quanto `percent_rank` podem ser dispendiosas no cálculo com grandes conjuntos de dados, já que requerem a classificação de todas as linhas. Filtrar a tabela para usar apenas o conjunto de dados do qual você precisa ajuda. Alguns bancos de dados implementaram versões aproximadas das funções cujo cálculo é mais rápido e geralmente retorna resultados de alta qualidade se uma precisão absoluta não for requerida. Examinaremos usos adicionais para as funções n-tiles na discussão sobre detecção de anomalias no Capítulo 6.

Em muitos contextos, não há apenas uma maneira correta ou objetivamente melhor de examinar distribuições de dados. Existe alguma flexibilidade para os analistas usarem as técnicas anteriores e entender os dados e apresentá-los para outras pessoas. No entanto, os cientistas de dados precisam usar o bom senso e devem empregar seu "radar ético" sempre que compartilharem distribuições de dados sensíveis.

Criação de perfis: qualidade dos dados

A qualidade dos dados é absolutamente crítica no que diz respeito a criar uma boa análise. Embora possa parecer óbvio, foi uma das lições mais difíceis que aprendi em meus anos de trabalho com dados. É fácil nos concentrarmos excessivamente na mecânica de processamento dos dados, encontrando técnicas de consulta inteligentes e a visualização correta, e os stakeholders acabarem ignorando tudo isso e detectando a única inconsistência existente nos dados. Assegurar a qualidade dos dados pode ser uma das partes mais complexas e desmotivadoras da análise. O ditado "entra lixo, sai lixo" captura apenas parte do problema. O uso de bons

ingredientes, porém com suposições incorretas, também pode levar à saída de lixo.

Comparar dados com valores de referência reais (ground truth), ou com o que poderia ser considerado verdadeiro, é ideal, mas nem sempre possível. Por exemplo, se você estivesse trabalhando com uma réplica de um banco de dados de produção, poderia comparar as contagens de linhas de cada sistema para verificar se todas as linhas chegaram no banco de dados replicado. Em outra situação, você poderia conhecer o valor do dólar e a contagem de vendas de um mês específico e, portanto, poderia consultar essas informações no banco de dados para se certificar se a soma (`sum`) das vendas e a contagem (`count`) dos registros coincidem. Com frequência a diferença entre os resultados da consulta e o valor esperado se resume a se você aplicou os filtros corretos, como excluir pedidos cancelados ou contagens de teste; como manipulou valores nulos e anomalias na grafia; e se definiu as condições *JOIN* certas entre as tabelas.

A criação de perfis é uma maneira de revelar problemas de qualidade de dados logo no início, antes que eles afetem negativamente os resultados e as conclusões provenientes dos dados. Ela revela valores nulos, codificações categóricas que precisam ser decifradas, campos com vários valores que precisam de parsing, e formatos de data/hora incomuns. Também pode revelar lacunas e alterações de etapas nos dados que resultaram do rastreamento de alterações ou ausências. Raramente os dados são perfeitos, e com frequência é somente por meio do seu uso na análise que problemas de qualidade são descobertos.

Detectando duplicidades

Uma *duplicata* ocorre quando temos duas (ou mais) linhas com as mesmas informações. Podem existir duplicidades por várias razões. Um erro pode ser cometido durante a entrada de dados, se houver alguma etapa manual. Uma chamada de rastreamento pode ser acionada duas vezes. Uma etapa de processamento pode ser executada várias vezes. Você poderia criá-la acidentalmente com uma *JOIN* muitos para muitos oculta. Independentemente de como elas surgirem, as duplicidades podem realmente prejudicar sua análise. Lembro-me de situações no início de minha carreira em que pensei que tinha chegado a um ótimo resultado

até o gerente de produtos apontar que meus valores de vendas eram o dobro das vendas reais. É embaraçoso, diminui a confiança e requer retrabalho e às vezes revisões cuidadosas do código para encontrar o problema. Aprendi a procurar duplicidades enquanto trabalho.

Felizmente, é relativamente fácil encontrar duplicidades nos dados. Uma maneira é inspecionando uma amostra, com todas as colunas ordenadas:

```
SELECT column_a, column_b, column_c...
FROM table
ORDER BY 1,2,3...
;
```

Esse código revelará se os dados têm muitas duplicidades, por exemplo, na verificação de um novo conjunto de dados, se você suspeitar que um processo está gerando duplicidades, ou após uma possível *JOIN* Cartesiana. Se só houver algumas duplicidades, elas podem não aparecer na amostra. E percorrer os dados para tentar detectar duplicidades cansa a vista e o cérebro. Uma maneira mais sistemática de encontrar duplicidades é selecionando as colunas (com *SELECT*) e depois contando (com `count`) as linhas (você deve se lembrar de que já viu isso na discussão de histogramas!):

```
SELECT count(*)
FROM
(
    SELECT column_a, column_b, column_c...
    , count(*) as records
    FROM...
    GROUP BY 1,2,3...
) a
WHERE records > 1
;
```

Esse código informará se há algum caso de duplicidades. Se a consulta retornar 0 é porque não há problemas. Para ver mais detalhes, você pode listar o número de registros (2, 3, 4 etc.):

```
SELECT records, count(*)
FROM
(
```

```
    SELECT column_a, column_b, column_c..., count(*) as records
    FROM...
    GROUP BY 1,2,3...
) a
WHERE records > 1
GROUP BY 1
;
```

 Como alternativa a uma subconsulta, você pode usar uma cláusula *HAVING* e manter tudo em uma única consulta principal. Já que ela será avaliada depois da agregação e de *GROUP BY*, *HAVING* pode ser usada para filtrar o valor da agregação:

```
SELECT column_a, column_b, column_c..., count(*) as records
FROM...
GROUP BY 1,2,3...
HAVING count(*) > 1
;
```

Prefiro usar subconsultas, porque acho que elas são uma maneira útil de organizar minha lógica. O Capítulo 8 discutirá a ordem de avaliação e as estratégias para você manter suas consultas SQL organizadas.

Para ver detalhes completos de que registros têm duplicidades, você pode listar todos os campos e, em seguida, usar essas informações para detectar quais registros são problemáticos:

```
SELECT *
FROM
(
    SELECT column_a, column_b, column_c..., count(*) as records
    FROM...
    GROUP BY 1,2,3...
) a
WHERE records = 2
;
```

Detectar duplicidades é uma coisa; descobrir o que fazer com elas é outra. Quase sempre é útil entender por que as duplicidades estão ocorrendo e, se possível, corrigir o problema em uma etapa inicial. Um processo de dados pode ser melhorado para a redução ou a remoção da duplicação?

Há um erro em um processo ETL? Você deixou de manipular um relacionamento um para muitos em uma *JOIN*? A seguir, veremos algumas opções para a manipulação e a remoção de duplicidades com SQL.

Desduplicação com GROUP BY e DISTINCT

Duplicidades ocorrem, e nem sempre elas são resultado de dados ruins. Por exemplo, suponhamos que quiséssemos obter uma lista de todos os clientes que concluíram com sucesso uma transação para podermos enviar para eles um cupom para seu próximo pedido. Poderíamos unir (com *JOIN*) a tabela `customers` à tabela `transactions`, o que restringiria os registros retornados apenas aos clientes que aparecem na tabela `transactions`:

```
SELECT a.customer_id, a.customer_name, a.customer_email
FROM customers a
JOIN transactions b on a.customer_id = b.customer_id
;
```

Esse código retornará uma linha para cada cliente de cada transação e deve haver pelo menos alguns clientes que fizeram transações mais de uma vez. Criamos duplicidades acidentalmente, não porque há algum problema subjacente de qualidade de dados, mas porque não evitamos a duplicação nos resultados. Felizmente, existem várias maneiras de evitar isso com SQL. Uma maneira de remover duplicidades é usando a palavra-chave *DISTINCT*:

```
SELECT distinct a.customer_id, a.customer_name, a.customer_email
FROM customers a
JOIN transactions b on a.customer_id = b.customer_id
;
```

Outra opção é usar uma cláusula *GROUP BY*, que, embora normalmente seja vista junto com uma agregação, também fará a desduplicação da mesma maneira que *DISTINCT*. Lembro-me da primeira vez em que vi um colega usar *GROUP BY* sem uma desduplicação de agregação – nem mesmo sabia que isso era possível. Achei um pouco menos intuitivo do que *DISTINCT*, mas o resultado é o mesmo:

```
SELECT a.customer_id, a.customer_name, a.customer_email
FROM customers a
```

```
JOIN transactions b on a.customer_id = b.customer_id
GROUP BY 1,2,3
;
```

Outra técnica útil é executar uma agregação que retorne uma linha por entidade. Embora tecnicamente não seja uma desduplicação, tem efeito semelhante. Por exemplo, se tivéssemos várias transações executadas pelo mesmo cliente e precisássemos retornar um registro por cliente, poderíamos encontrar a `transaction_date` (data da transação) que foi a `min` (primeira) e/ou a que foi a `max` (mais recente):

```
SELECT customer_id
,min(transaction_date) as first_transaction_date
,max(transaction_date) as last_transaction_date
,count(*) as total_orders
FROM table
GROUP BY customer_id
;
```

Os dados duplicados, ou dados que contenham vários registros por entidade mesmo se tecnicamente não forem duplicidades, são uma das razões mais comuns para resultados de consulta incorretos. Você pode suspeitar das duplicidades como sendo a causa se repentinamente o número de clientes ou o total de vendas retornado por uma consulta for muitas vezes maior do que o esperado. Felizmente, há várias técnicas que podem ser aplicadas para impedir que isso ocorra.

Outro problema comum é o dos dados ausentes, que veremos a seguir.

Preparação: limpeza dos dados

Com frequência a criação de perfis revela onde alterações podem tornar os dados mais úteis para análise. Algumas das etapas são as transformações CASE, o ajuste de valores nulos e a alteração de tipos de dados.

Limpando dados com transformações CASE

As instruções CASE podem ser usadas na execução de várias tarefas de limpeza, acréscimo e resumo. Os dados podem existir e serem precisos,

mas seria mais útil para a análise se os valores fossem padronizados ou agrupados em categorias. A estrutura das instruções CASE foi apresentada anteriormente neste capítulo, na seção sobre binning.

Valores não padrão ocorrem por várias razões. Os valores podem vir de diferentes sistemas com listas de opções distintas, o código do sistema pode ter sido alterado, as opções podem ter sido apresentadas para o cliente em diferentes idiomas ou o cliente pode ter optado por fornecer o valor em vez de selecioná-lo em uma lista.

Imagine um campo contendo informações sobre o gênero de uma pessoa. Os valores que indicam que ela é do sexo feminino aparecem como "F," "female" e "femme". Podemos padronizar os valores desta forma:

```
CASE when gender = 'F' then 'Female'
     when gender = 'female' then 'Female'
     when gender = 'femme' then 'Female'
     else gender
     end as gender_cleaned
```

As instruções CASE também podem ser usadas para adicionar categorização ou acréscimos não existentes nos dados originais. Como exemplo, muitas organizações usam um Net Promoter Score, or NPS, para monitorar o sentimento do cliente. As pesquisas do NPS pedem aos participantes que classifiquem, em uma escala de 0 a 10, qual é a probabilidade de eles recomendarem uma empresa ou produto para um amigo ou colega. As pontuações de 0 a 6 são consideradas depreciadoras, 7 e 8 são passivas e 9 e 10 são incentivadoras. A pontuação final é calculada pela subtração do percentual de pontuações depreciadoras do percentual de pontuações incentivadoras. Geralmente os conjuntos de dados do resultado da pesquisa incluem comentários opcionais em texto livre e às vezes são acrescidos de informações que a organização conhece sobre a pessoa que participou. Após a obtenção de um conjunto de dados de respostas de uma pesquisa NPS, a primeira etapa é agrupá-las nas categorias de detrator, passivo e incentivador:

```
SELECT response_id
,likelihood
,case when likelihood <= 6 then 'Detractor'
      when likelihood <= 8 then 'Passive'
```

```
      else 'Promoter'
      end as response_type
FROM nps_responses
;
```

Observe que o tipo de dado pode diferir entre o campo que está sendo avaliado e o tipo de dado de retorno. Nesse caso, estamos verificando um inteiro e retornando uma string. Listar todos os valores com uma lista IN também é uma opção. O operador IN permite que você especifique uma lista de itens em vez de ter de escrever uma igualdade para cada um separadamente. Ele é útil quando a entrada não é contínua ou quando valores em ordem não devam ser agrupados:

```
CASE when likelihood in (0,1,2,3,4,5,6) then 'Detractor'
     when likelihood in (7,8) then 'Passive'
     when likelihood in (9,10) then 'Promoter'
     end as response_type
```

As instruções CASE podem considerar várias colunas e conter uma lógica AND/OR. Elas também podem ser aninhadas, embora com frequência isso possa ser evitado com a lógica AND/OR:

```
CASE when likelihood <= 6
          and country = 'US'
          and high_value = true
          then 'US high value detractor'
     when likelihood >= 9
          and (country in ('CA','JP')
               or high_value = true
              )
          then 'some other label'
      ... end
```

Alternativas para a limpeza de dados

A limpeza ou o acréscimo de dados com uma instrução CASE funciona bem quando há uma lista de variações relativamente curta, quando conseguimos encontrar todas elas na lista e não é esperado que a lista de valores mude. Para listas mais longas e que mudem com frequência, uma tabela de pesquisa (lookup table) pode ser uma opção melhor. A tabela de pesquisa fica no banco de dados e é estática ou preenchida com um código que procura novos valores periodicamente. A consulta será unida (com *JOIN*) à tabela de pesquisa para que

sejam obtidos dados limpos. Dessa forma, os valores limpos podem ser mantidos fora do código e ser usados por muitas consultas, sem precisarmos nos preocupar com a manutenção da consistência entre elas. Um exemplo poderia ser uma tabela de pesquisa que fizesse o mapeamento de abreviações de estado para os nomes completos dos estados. Em meu trabalho, costumo começar com uma instrução CASE e crio uma tabela de pesquisa somente depois de a lista se tornar difícil de controlar ou quando fica claro que minha equipe ou eu precisaremos usar essa etapa de limpeza repetidamente.

É claro que é útil investigar se os dados podem ser limpos em uma etapa inicial. Houve uma situação em que comecei com uma instrução CASE de cerca de 5 linhas que cresceu para 10 e acabou ficando com mais de 100 linhas, ponto em que a lista ficou difícil de controlar e de manter. Os insights eram suficientemente valiosos, logo consegui convencer os engenheiros a alterar o código de rastreamento e enviar as categorizações significativas no fluxo de dados.

Outra tarefa útil que você pode executar com instruções CASE é criar flags que indiquem se um valor específico está presente, sem retornar o valor real. Isso pode ser útil durante a criação de perfis para sabermos o quanto a existência de um atributo é comum. Outro uso para as flags é durante a preparação de um conjunto de dados para análise estatística. Nesse caso, uma flag também é conhecida como variável dummy, recebendo um valor 0 ou 1 e indicando a presença ou a ausência de alguma variável qualitativa. Por exemplo, podemos criar as flags `is_female` e `is_promoter` com instruções CASE nos campos `gender` e `likelihood` (para recomendar):

```
SELECT customer_id
,case when gender = 'F' then 1 else 0 end as is_female
,case when likelihood in (9,10) then 1 else 0 end as is_promoter
FROM ...
;
```

Se você estiver trabalhando com um conjunto de dados que tenha várias linhas por entidade, como com os itens de linha de um pedido, pode achatar (flatten) os dados com uma instrução CASE inserida em uma agregação e transformá-la ao mesmo tempo em uma flag usando 1 e 0 como valor de retorno. Vimos anteriormente que um tipo de dado BOOLEAN costuma ser usado na criação de flags (campos que representam a presença ou a ausência de algum atributo). Aqui, 1 é

substituído por TRUE e 0 por FALSE para que uma agregação `max` possa ser aplicada. Essa técnica funciona assim: para cada cliente, a instrução CASE retorna 1 para qualquer linha que tenha o tipo de fruta "apple." Em seguida, `max` é avaliada e retornará o maior valor de qualquer uma das linhas. Contanto que um cliente tenha comprado uma maçã pelo menos uma vez, a flag será 1; caso contrário, será 0:

```
SELECT customer_id
,max(case when fruit = 'apple' then 1
          else 0
          end) as bought_apples
,max(case when fruit = 'orange' then 1
          else 0
          end) as bought_oranges
FROM ...
GROUP BY 1
;
```

Você também pode construir condições mais complexas para as flags, como requerer um limite ou uma quantidade de algo antes da rotulação com o valor 1:

```
SELECT customer_id
,max(case when fruit = 'apple' and quantity > 5 then 1
          else 0
          end) as loves_apples
,max(case when fruit = 'orange' and quantity > 5 then 1
          else 0
          end) as loves_oranges
FROM ...
GROUP BY 1
;
```

As instruções CASE são poderosas e, como vimos, podem ser usadas na limpeza e no enriquecimento de conjuntos de dados, e para adicionar flags ou variáveis dummy a eles. Na próxima seção, examinaremos algumas funções especiais relacionadas às instruções CASE que manipulam especificamente valores nulos.

Conversões de tipos e casting

Cada campo de um banco de dados é definido com um tipo de dado, algo que examinamos no começo deste capítulo. Quando dados são inseridos em uma tabela, valores que não têm o tipo do campo são rejeitados pelo banco de dados. Strings não podem ser inseridas em campos de inteiros e booleanos não são permitidos em campos de data. Quase sempre, podemos supor que os tipos de dados estão corretos e aplicar funções de string a strings, funções de data a datas e assim por diante. Ocasionalmente, entretanto, temos de sobrepor o tipo de dado do campo e transformá-lo em algo diferente. É nesse momento que as conversões de tipo e o casting entram em cena.

As *funções de conversão de tipo* permitem que dados com o formato apropriado sejam alterados de um tipo de dado para outro. A sintaxe apresenta algumas notações que são basicamente equivalentes. Uma maneira de alterar o tipo de dado é com a função `cast`, por meio de `cast (input as data_type)` ou da notação com dois pontos duplos, `input :: data_type`. Essas duas formas são equivalentes e convertem o inteiro 1.234 em uma string:

```
cast (1234 as varchar)
1234::varchar
```

Converter um inteiro em uma string pode ser útil em instruções CASE na categorização de valores numéricos com algum valor superior ou inferior ilimitado. Por exemplo, no código a seguir, deixar os valores que são menores ou iguais a 3 como inteiros, mas retornar a string "4+" para valores mais altos, resultaria em um erro:

```
case when order_items <= 3 then order_items
     else '4+'
     end
```

Converter os inteiros para o tipo VARCHAR resolve o problema:

```
case when order_items <= 3 then order_items::varchar
     else '4+'
     end
```

As conversões de tipo também são úteis quando valores que deveriam ser inteiros têm seu parsing feito a partir de uma string para em seguida serem agregados ou serem usados em funções matemáticas. Suponhamos

que tivéssemos um conjunto de dados de preços, mas os valores incluíssem o cifrão ($) e, portanto, o tipo de dado do campo fosse VARCHAR. Podemos remover o caractere $ com uma função chamada `replace`, que será abordada com mais detalhes quando examinarmos a análise de texto no Capítulo 5:

```
SELECT replace('$19.99','$','');
replace
-------
9.99
```

No entanto, o resultado ainda é um VARCHAR; logo, tentar aplicar uma agregação retornará um erro. Para corrigir isso, podemos converter (com `cast`) o resultado para FLOAT:

```
replace('$19.99','$','')::float
cast(replace('$19.99','$','')) as float
```

Podemos encontrar as datas e as datas/horas em um confuso conjunto de formatos e é útil saber como *convertê-las* para o formato desejado. Mostrarei alguns exemplos sobre conversão de tipos aqui, e o Capítulo 3 fornecerá mais detalhes sobre cálculos de data e data/hora. Como um exemplo simples, suponhamos que geralmente dados de transações ou de eventos chegassem no banco de dados como um TIMESTAMP, mas quiséssemos resumir algum valor, como o de transações por dia. Simplesmente agrupar pelo timestamp resultará em mais linhas do que o necessário. Converter TIMESTAMP para DATE reduz o tamanho dos resultados e atinge o objetivo de criar o resumo:

```
SELECT tx_timestamp::date, count(transactions) as num_transactions
FROM ...
GROUP BY 1
;
```

Da mesma forma, podemos converter DATE para TIMESTAMP quando uma função SQL demandar um argumento TIMESTAMP. Às vezes o ano, o mês e o dia são armazenados em colunas separadas, ou acabam sendo considerados elementos separados por seu parsing ter sido feito a partir de uma string mais longa. Eles têm então de ser reunidos novamente como uma data. Para fazer isso, usamos o operador de concatenação || (pipe duplo) ou a função `concat` e depois convertemos o resultado para

DATE. Qualquer uma dessas sintaxes funciona e retorna o mesmo valor:

```
(year || ',' || month|| '-' || day)::date
```

Ou, de maneira equivalente:

```
cast(concat(year, '-', month, '-', day) as date)
```

Mais uma maneira de fazer a conversão entre strings e datas é usando a função `date`. Por exemplo, podemos construir uma string como fizemos anteriormente e convertê-la em uma data:

```
date(concat(year, '-', month, '-', day))
```

As funções *to_datatype* podem receber tanto um valor quanto uma string e, portanto, nos dão mais controle sobre como a data será convertida. A Tabela 2.4 resume as funções e suas finalidades. Elas são particularmente úteis na conversão a partir de e para os formatos DATE ou DATETIME, já que nos permitem especificar a ordem dos elementos de data e hora.

Tabela 2.4: As funções to_datatype

Função	Finalidade
`to_char`	Converte outros tipos para string
`to_number`	Converte outros tipos para um valor numérico
`to_date`	Converte outros tipos para data, com as partes da data especificadas
`to_timestamp`	Converte outros tipos para data, com as partes de data e hora especificadas

Às vezes o banco de dados converte um tipo de dado automaticamente. Isso se chama *coerção de tipo*. Por exemplo, geralmente os tipos numéricos INT e FLOAT podem ser usados juntos em funções matemáticas ou agregações sem a alteração explícita do tipo. Valores CHAR e VARCHAR costumam ser combinados. Alguns bancos de dados fazem a coerção de campos BOOLEAN para os valores 0 e 1, em que 0 é FALSE e 1 é TRUE, mas outros demandam que a conversão dos valores seja feita explicitamente. Certos bancos de dados são mais exigentes do que outros no que diz respeito a combinar datas e datas/horas em conjuntos de resultados e funções. Você pode ler a documentação ou fazer alguns testes simples com consultas para saber como o banco de dados com o qual está trabalhando manipula tipos de dados implícita e explicitamente. Normalmente há uma maneira de fazer o que queremos, mas às vezes é

preciso ser criativo para usar funções nas consultas.

Lidando com nulos: funções coalesce, nullif, nvl

O valor nulo foi um dos conceitos mais estranhos com o qual tive de me acostumar quando comecei a trabalhar com dados. Ele não é algo no qual pensemos no dia a dia, quando lidamos com quantidades concretas. O valor *nulo* tem um significado especial nos bancos de dados e foi introduzido por Edgar Codd, o inventor do banco de dados relacional, para assegurar que eles tenham uma maneira de representar informações ausentes. Se alguém me perguntar quantos paraquedas eu tenho, posso responder "nenhum". No entanto, se a pergunta não for feita, terei paraquedas nulos.

Os nulos podem representar campos para os quais nenhum dado foi coletado ou que não sejam aplicáveis a essa linha. Quando novas colunas são adicionadas a uma tabela, os valores de linhas criadas anteriormente são nulos a menos que algum outro valor seja fornecido explicitamente. Quando duas tabelas forem unidas por meio de uma *OUTER JOIN*, os nulos aparecerão em campos para os quais não houver um registro coincidente na segunda tabela.

Os valores nulos são problemáticos para certas agregações e agrupamentos, e diferentes tipos de bancos de dados os manipulam de maneiras distintas. Por exemplo, suponhamos que eu tivesse cinco registros, com 5, 10, 15, 20, e nulo. A soma deles é 50, mas a média será 10 ou 12,5 caso o valor nulo tenha ou não sido contado no denominador. A questão inteira também pode ser considerada inválida já que um dos valores é nulo. Para a maioria das funções de banco de dados, uma entrada nula retorna uma saída nula. Igualdades e diferenças envolvendo nulo também retornam nulo. Vários resultados inesperados e frustrantes podem ser exibidos por suas consultas se você não tomar cuidado com os nulos.

Quando as tabelas são definidas, elas podem permitir nulos, rejeitar nulos ou fornecer um valor padrão se, caso contrário, o campo for deixado com nulo. Na prática, isso significa que nem sempre você poderá ter certeza de que um campo aparecerá como nulo se os dados estiverem ausentes, porque ele pode ter sido preenchido com um valor padrão, como 0. Uma

vez tive uma longa discussão com um engenheiro de dados quando foi descoberto que datas nulas no sistema de origem eram representadas pelo padrão "1970-01-01" em nosso data warehouse. Insisti que as datas deviam ser nulas, para refletir o fato de que elas eram desconhecidas ou não aplicáveis. O engenheiro ressaltou que eu poderia filtrar essas datas ou alterá-las novamente para nulo com uma instrução CASE. Acabei fazendo minha opinião prevalecer ao destacar que algum dia outro usuário que não conhecesse tão bem as nuances das datas padrão faria uma consulta e obteria o enigmático conjunto de clientes de cerca de um ano antes de a empresa ser fundada.

Com frequência os nulos são inconvenientes ou inapropriados para a análise que queremos fazer. Eles também podem tornar a saída confusa para o público-alvo e atrapalhar a análise. Os empresários não sabem como interpretar um valor nulo ou podem presumir que os valores nulos representam um problema de qualidade de dados.

Strings vazias

Um conceito relacionado aos nulos, mas que é um pouco diferente, é a *string vazia*, na qual não há valor, porém o campo não é tecnicamente nulo. Uma razão para uma string vazia ser usada seria para indicar que é sabido que um campo está vazio, em vez de ser nulo, caso em que o valor pode estar faltando ou ser desconhecido. Por exemplo, o banco de dados poderia ter um campo `name_suffix` para conter um valor como "Jr". Muitas pessoas não têm um sufixo no nome; logo, uma string vazia é apropriada. A string vazia também pode ser usada como valor padrão em vez de nulo, ou como uma maneira de resolver uma restrição NOT NULL pela inserção de um valor, mesmo se for vazio. Uma string vazia pode ser especificada em uma consulta com duas aspas:

```
WHERE my_field = '' or my_field <> 'apple'
```

Criar perfis com as frequências de valores deve revelar se seus dados incluem nulos, strings vazias ou ambos.

Há algumas maneiras de substituir nulos por valores alternativos: com instruções CASE e com as funções especializadas `coalesce` e `nullif`. Vimos anteriormente que as instruções CASE podem verificar uma condição e retornar um valor. Elas também podem ser usadas para procurar um valor nulo e, se um for encontrado, substituí-lo por outro valor:

```
case when num_orders is null then 0 else num_orders end
case when address is null then 'Unknown' else address end
```

```
case when column_a is null then column_b else column_a end
```

A função `coalesce` é uma maneira mais compacta de fazer isso. Ela recebe dois ou mais argumentos e retorna o primeiro que não é nulo:

```
coalesce(num_orders,0)
coalesce(address,'Unknown')
coalesce(column_a,column_b)
coalesce(column_a,column_b,column_c)
```

 A função `nvl` está presente em alguns bancos de dados e é semelhante a `coalesce`, mas só permite dois argumentos.

A função `nullif` compara dois números, e se eles não forem iguais, retorna o primeiro; se eles *forem* iguais, a função retorna nulo. A execução desse código:

```
nullif(6,7)
```

retorna 6, enquanto nulo é retornado por:

```
nullif(6,6)
```

`nullif` é equivalente à instrução case mais verbosa a seguir:

```
case when 6 = 7 then 6
     when 6 = 6 then null
     end
```

Essa função pode ser útil para transformar valores novamente em nulos quando você souber que um valor padrão específico foi inserido no banco de dados. Por exemplo, em meu exemplo de hora padrão, poderíamos alterá-la novamente para nulo usando:

```
nullif(date,'1970-01-01')
```

 Os nulos podem ser problemáticos na filtragem de dados da cláusula *WHERE*. É muito fácil retornar valores que sejam nulos:

```
WHERE my_field is null
```

No entanto, suponhamos que `my_field` contivesse alguns nulos e alguns nomes de frutas. Eu gostaria de retornar todas as linhas que não fossem maçãs. Parece que o seguinte irá funcionar:

```
WHERE my_field <> 'apple'
```

Contudo, alguns bancos de dados excluirão tanto as linhas "apple" quanto todas as linhas com valores nulos em `my_field`. Para corrigir isso, o SQL deve

filtrar "apple" deixando-a de fora e incluir explicitamente os nulos conectando as condições com OR:

```
WHERE my_field <> 'apple' or my_field is null
```

Os nulos são um fato da vida quando trabalhamos com dados. Independentemente de por que ocorreram, com frequência precisamos considerá-los na criação de perfis e como alvos da limpeza de dados. Felizmente, há várias maneiras de detectá-los com SQL, assim como diversas funções úteis que permitem substituir os nulos por valores alternativos. A seguir examinaremos os dados ausentes, um problema que pode causar nulos, mas tem implicações ainda maiores e, portanto, merece uma seção própria.

Dados ausentes

Dados podem faltar por várias razões, cada uma tendo suas próprias maneiras de afetar como manipularemos a ausência. Um campo pode não ter sido demandado pelo sistema ou processo que o coletou, como ocorre com um campo opcional "onde você ouviu falar de nós?" em um fluxo de checkout de e-commerce. Demandar a inserção desse campo pode irritar o cliente e diminuir os checkouts bem-sucedidos. Alternativamente, dados podem ser necessários, mas não ser coletados devido a um bug no código ou a um erro humano, como em um questionário médico em que o entrevistador pule a segunda página de perguntas. Uma alteração na maneira de os dados serem coletados pode resultar em registros anteriores ou posteriores à alteração terem valores ausentes. Uma ferramenta rastreando interações de aplicação móvel poderia adicionar um registro adicional de um campo independentemente de a interação ter sido com um toque ou uma rolagem, por exemplo, ou remover outro campo devido a uma alteração na funcionalidade. Os dados podem ficar órfãos se uma tabela referenciar um valor em outra tabela e essa linha ou a tabela inteira tiver sido excluída ou ainda não tiver sido carregada no data warehouse. Para concluir, os dados podem estar disponíveis, mas não no nível de detalhes, ou granularidade, necessário para a análise. Um exemplo seriam as empresas de vendas por assinatura, nas quais os clientes pagam anualmente por um produto mensal e o que se quer é analisar a receita mensal.

Além de criar perfis dos dados com histogramas e a análise de frequência, geralmente podemos detectar dados ausentes comparando valores de duas tabelas. Por exemplo, poderíamos esperar que cada cliente da tabela `transactions` também tivesse um registro na tabela `customer`. Para verificar isso, consulte as tabelas usando uma *LEFT JOIN* e adicione uma condição *WHERE* para encontrar os clientes que não existem na segunda tabela:

```
SELECT distinct a.customer_id
FROM transactions a
LEFT JOIN customers b on a.customer_id = b.customer_id
WHERE b.customer_id is null
;
```

Os dados ausentes podem dar indicações importantes, logo não presuma que seja sempre preciso corrigi-los ou fornecê-los. Eles podem revelar o design do sistema subjacente ou tendências do processo de coleta de dados.

Registros com campos ausentes podem ser totalmente removidos por meio de filtragem, mas geralmente é bom mantê-los e fazer alguns ajustes de acordo com o que se sabe sobre os valores esperados ou típicos. Temos algumas opções, chamadas técnicas de *imputação*, para o fornecimento de dados ausentes. Elas incluem fornecer a média ou a mediana do conjunto de dados ou o valor anterior. Documentar os dados ausentes e como eles foram substituídos é importante, já que isso pode afetar a interpretação e o uso dos dados em um momento posterior. Valores imputados podem ser particularmente problemáticos quando os dados são usados em machine learning, por exemplo.

Uma opção comum é fornecer um valor constate para os dados ausentes. Fornecer um valor constante pode ser útil quando o valor é conhecido para alguns registros ainda que ele não tenha sido fornecido no banco de dados. Por exemplo, suponhamos que um bug em um software impedisse o fornecimento do preço (`price`) para um item chamado "xyz", mas soubéssemos que o preço é sempre 20 dólares. Uma instrução CASE pode ser adicionada à consulta para manipular isso:

```
case when price is null and item_name = 'xyz' then 20
                                              else price
                                              end as price
```

Outra opção é fornecer um valor derivado, seja de uma função matemática de outras colunas ou de uma instrução CASE. Por exemplo, digamos que tivéssemos um campo para o valor de vendas líquidas (`net_sales`) de cada transação. Devido a um bug, esse campo não é preenchido em algumas linhas, mas seus campos `gross_sales` e `discount` são preenchidos. Podemos calcular `net_sales` subtraindo `discount` de `gross_sales`:

```
SELECT gross_sales - discount as net_sales...
```

Os valores ausentes também podem ser fornecidos com o uso de valores de outras linhas do conjunto de dados. Trazer um valor da linha anterior chama-se *preenchimento progressivo (fill forward)*, enquanto usar um valor da próxima linha chama-se *preenchimento regressivo (fill backward)*. Esses preenchimentos podem ser executados com as funções de janela `lag` e `lead`, respectivamente. Por exemplo, suponhamos que nossa tabela de transações tivesse um campo `product_price` que armazenasse o preço sem desconto que o cliente paga por um produto (`product`). Ocasionalmente esse campo não é preenchido, mas podemos supor que o preço é o mesmo pago pelo último cliente para comprar o produto. Podemos fornecer o valor anterior usando a função `lag`, usando *PARTITION BY* com `product` para assegurar que somente o preço desse produto seja trazido e usando *ORDER BY* com a data apropriada para garantir que o preço seja extraído da transação anterior mais recente:

```
lag(product_price) over (partition by product order by order_date)
```

A função `lead` poderia ser usada para fornecer o preço pago pelo produto na transação seguinte. Alternativamente, poderíamos calcular a média (`avg`) dos preços do produto e usá-la para fornecer o valor ausente. Fornecer o valor anterior, o próximo valor ou a média envolve fazer algumas suposições sobre os valores típicos e o que é sensato incluir em uma análise. É sempre uma boa ideia verificar os resultados para termos certeza de que eles são plausíveis e nos certificarmos de que dados tenham sido interpolados quando não estavam disponíveis.

Para dados que estão disponíveis, mas não têm a granularidade necessária, com frequência temos de criar linhas adicionais no conjunto de dados. Por exemplo, digamos que tivéssemos uma tabela `customer_subscriptions` (assinaturas de clientes) com os campos

`subscription_date` e `annual_amount`. Podemos distribuir esse valor anual da assinatura em 12 valores iguais de receita mensal fazendo a divisão por 12, convertendo assim a ARR (annual recurring revenue, receita recorrente anual) em MRR (monthly recurring revenue, receita recorrente mensal):

```
SELECT customer_id
,subscription_date
,annual_amount
,annual_amount / 12 as month_1
,annual_amount / 12 as month_2
...
,annual_amount / 12 as month_12
FROM customer_subscriptions
;
```

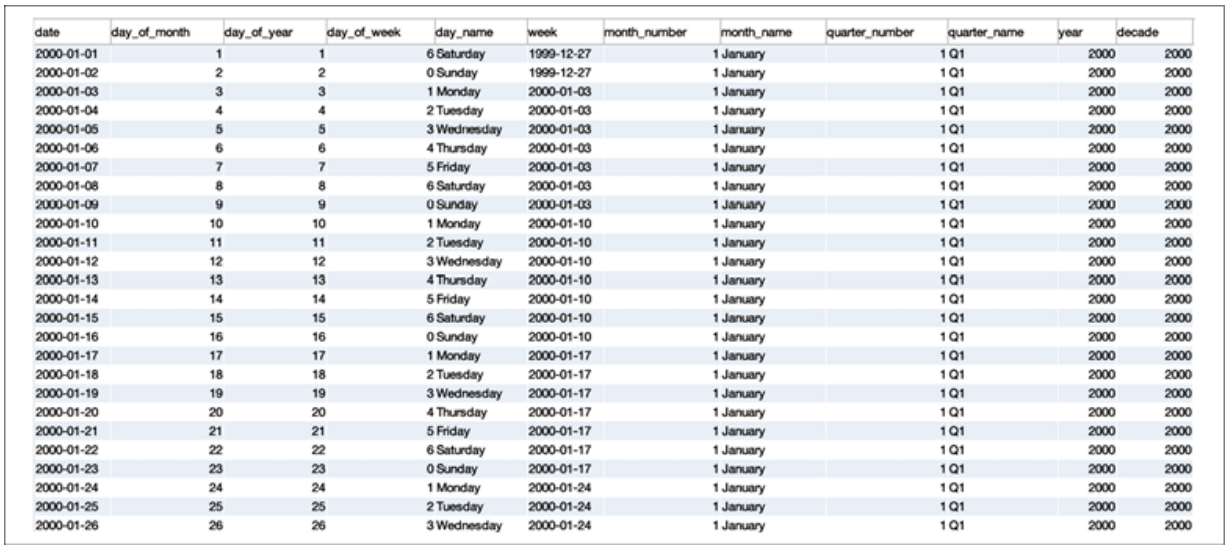
Isso pode ficar um pouco tedioso, principalmente se os períodos de assinatura puderem ser de dois, três ou cinco anos, assim como de um ano. Também não será útil se quisermos as datas reais dos meses. Teoricamente, poderíamos escrever uma consulta como esta:

```
SELECT customer_id
,subscription_date
,annual_amount
,annual_amount / 12 as '2020-01'
,annual_amount / 12 as '2020-02'
...
,annual_amount / 12 as '2020-12'
FROM customer_subscriptions
;
```

No entanto, se os dados incluírem pedidos feitos pelos clientes com o passar do tempo, embutir os nomes dos meses no código não trará precisão. Poderíamos usar instruções CASE junto com os nomes dos meses embutidos no código, mas novamente isso seria tedioso e propenso a erros à medida que fosse adicionada uma lógica mais complexa. Criar novas linhas por meio de uma junção (*JOIN*) com uma tabela, como a de dimensão de data, forneceria uma solução elegante.

Uma *dimensão de data* é uma tabela estática que tem uma linha por dia, com atributos de datas estendidos e opcionais, como dia da semana,

nome do mês, fim do mês e ano fiscal. As datas se estendem o suficiente para o passado e para o futuro para abranger todos os usos previstos. Como só temos 365 ou 366 dias por ano, tabelas que abrangem até mesmo 100 anos não ocupam muito espaço. A Figura 2.3 exibe uma amostra dos dados de uma tabela de dimensão de data. O exemplo de código para a criação de uma dimensão de data com o uso de funções SQL pode ser visto no site do GitHub referente ao livro (*https://oreil.ly/kv3dZ*).



date	day_of_month	day_of_year	day_of_week	day_name	week	month_number	month_name	quarter_number	quarter_name	year	decade
2000-01-01	1	1	6	Saturday	1999-12-27	1	January	1	Q1	2000	2000
2000-01-02	2	2	0	Sunday	1999-12-27	1	January	1	Q1	2000	2000
2000-01-03	3	3	1	Monday	2000-01-03	1	January	1	Q1	2000	2000
2000-01-04	4	4	2	Tuesday	2000-01-03	1	January	1	Q1	2000	2000
2000-01-05	5	5	3	Wednesday	2000-01-03	1	January	1	Q1	2000	2000
2000-01-06	6	6	4	Thursday	2000-01-03	1	January	1	Q1	2000	2000
2000-01-07	7	7	5	Friday	2000-01-03	1	January	1	Q1	2000	2000
2000-01-08	8	8	6	Saturday	2000-01-03	1	January	1	Q1	2000	2000
2000-01-09	9	9	0	Sunday	2000-01-03	1	January	1	Q1	2000	2000
2000-01-10	10	10	1	Monday	2000-01-10	1	January	1	Q1	2000	2000
2000-01-11	11	11	2	Tuesday	2000-01-10	1	January	1	Q1	2000	2000
2000-01-12	12	12	3	Wednesday	2000-01-10	1	January	1	Q1	2000	2000
2000-01-13	13	13	4	Thursday	2000-01-10	1	January	1	Q1	2000	2000
2000-01-14	14	14	5	Friday	2000-01-10	1	January	1	Q1	2000	2000
2000-01-15	15	15	6	Saturday	2000-01-10	1	January	1	Q1	2000	2000
2000-01-16	16	16	0	Sunday	2000-01-10	1	January	1	Q1	2000	2000
2000-01-17	17	17	1	Monday	2000-01-17	1	January	1	Q1	2000	2000
2000-01-18	18	18	2	Tuesday	2000-01-17	1	January	1	Q1	2000	2000
2000-01-19	19	19	3	Wednesday	2000-01-17	1	January	1	Q1	2000	2000
2000-01-20	20	20	4	Thursday	2000-01-17	1	January	1	Q1	2000	2000
2000-01-21	21	21	5	Friday	2000-01-17	1	January	1	Q1	2000	2000
2000-01-22	22	22	6	Saturday	2000-01-17	1	January	1	Q1	2000	2000
2000-01-23	23	23	0	Sunday	2000-01-17	1	January	1	Q1	2000	2000
2000-01-24	24	24	1	Monday	2000-01-24	1	January	1	Q1	2000	2000
2000-01-25	25	25	2	Tuesday	2000-01-24	1	January	1	Q1	2000	2000
2000-01-26	26	26	3	Wednesday	2000-01-24	1	January	1	Q1	2000	2000

Figura 2.3: Uma tabela de dimensão de data com atributos de data.

Se você está usando um banco de dados Postgres, a função `generate_series` pode ser usada na criação de uma dimensão de data para o preenchimento inicial da tabela ou se a criação de uma tabela não for uma opção. Ela tem o formato a seguir:

```
generate_series(start, stop, step interval)
```

Nessa função, `start` é a primeira data que queremos na série, `stop` é a última data, e `step interval` é o período de tempo entre os valores. O argumento `step interval` pode receber qualquer valor, mas um dia é apropriado para uma dimensão de data:

```
SELECT *
FROM generate_series('2000-01-01'::timestamp,'2030-12-31', '1 day')
```

A função `generate_series` requer que pelo menos um dos argumentos seja um TIMESTAMP; logo, "2000-01-01" é convertido para TIMESTAMP.

Podemos então criar uma consulta que resulte em uma linha para cada dia, independentemente de se um cliente fez um pedido em um dia específico. Isso será útil quando quisermos assegurar que um cliente seja contado para cada dia ou se quisermos contar ou analisar especificamente os dias nos quais um cliente não fez uma compra:

```
SELECT a.generate_series as order_date, b.customer_id, b.items
FROM
(
    SELECT *
    FROM generate_series('2020-01-01'::timestamp,'2020-12-31','1
  day')
) a
LEFT JOIN
(
    SELECT customer_id, order_date, count(item_id) as items
    FROM orders
    GROUP BY 1,2
) b on a.generate_series = b.order_date
;
```

Voltando ao nosso exemplo da assinatura, podemos usar a dimensão de data para criar um registro para cada mês executando *JOIN* em datas que estejam entre `subscription_date` e 11meses depois (para um total de 12 meses):

```
SELECT a.date
,b.customer_id
,b.subscription_date
,b.annual_amount / 12 as monthly_subscription
FROM date_dim a
JOIN customer_subscriptions b on a.date between b.subscription_date
and b.subscription_date + interval '11 months'
;
```

Os dados podem estar ausentes por várias razões, e conhecer a causa raiz é importante para a decisão de como lidar com isso. Há muitas opções para a descoberta e a substituição de dados ausentes. Elas incluem usar instruções CASE para definir valores padrão, derivar valores executando cálculos em outros campos da mesma linha e fazer uma interpolação a

partir de outros valores da mesma coluna.

A limpeza de dados é uma parte importante do processo de preparação de dados. Os dados podem ter de ser limpos por muitas razões. Há a limpeza de dados que precisa ser feita para corrigir uma qualidade de dados insatisfatória, como quando existem valores inconsistentes ou ausentes nos dados brutos, mas também há a limpeza que é feita para tornar a análise posterior mais fácil e significativa. A flexibilidade do SQL nos permite executar tarefas de limpeza de várias maneiras.

Após os dados serem limpos, normalmente a próxima etapa do processo de preparação é modelar o conjunto de dados.

Preparação: modelando dados

Modelar dados é manipular a forma como os dados serão representados em colunas e linhas. Cada tabela do banco de dados tem uma forma. O conjunto de resultados de cada consulta tem uma forma. Modelar dados pode parecer um conceito um pouco abstrato, mas, se você trabalhar com dados suficientes, verá seu valor. É uma habilidade que pode ser aprendida, praticada e dominada.

Um dos conceitos mais importantes da modelagem de dados é descobrir a *granularidade* de dados necessária. Assim como as pedras podem variar de tamanho, de rochedos gigantes a grãos de areia, e chegando até a poeira microscópica, os dados também podem ter vários níveis de detalhes. Por exemplo, se considerarmos a população de um país como um rochedo, a de uma cidade seria uma pedra pequena e a de uma residência seria um grão de areia. Dados em um nível menor de detalhe poderiam incluir os nascimentos e mortes, ou mudanças de uma cidade ou país para outro.

Achatar dados é outro conceito importante da modelagem. Trata-se de reduzir o número de linhas que representam uma entidade, chegando inclusive a uma única linha. Unir várias tabelas para criar um único conjunto de dados de saída é uma maneira de achatar dados. Outra maneira é por meio da agregação.

Nesta seção, primeiro abordaremos algumas considerações referentes à seleção das formas dos dados. Em seguida, examinaremos certos casos de uso comuns: pivotar e despivotar. Veremos exemplos de modelagem de

dados para análises específicas no decorrer dos capítulos restantes. O Capítulo 8 fornecerá mais detalhes sobre como manter organizado um código SQL complexo quando da criação de conjuntos de dados para análise posterior.

Para qual saída: BI, visualização, estatística, ML

Decidir como será a sua modelagem de dados com SQL vai depender do que você quiser fazer com os dados. Geralmente é uma boa ideia exibir um conjunto de dados que tenha o menor número de linhas possível, atendendo ao mesmo tempo o requisito de granularidade. Isso alavancará o poder computacional do banco de dados, reduzirá o tempo necessário para mover os dados do banco de dados para algum outro local e diminuirá o volume de processamento que você ou outra pessoa terá de executar em outras ferramentas. Algumas das outras ferramentas para as quais sua saída poderá ir são uma ferramenta de BI para uso de relatórios e painéis, uma planilha para usuários do negócio examinarem, uma ferramenta estatística como o R, ou um modelo de machine learning em Python – ou você poderia exibir os dados diretamente em uma visualização criada com várias ferramentas.

Na exibição de dados em uma ferramenta de inteligência empresarial para a utilização de relatórios e painéis, é importante entender o caso de uso. Os conjuntos de dados podem ter de ser mais detalhados para permitir a exploração e o fatiamento pelos usuários finais. Eles podem ter de ser pequenos e agregados e incluir cálculos específicos para permitir tempos de carregamento e resposta rápidos em painéis executivos. Saber como a ferramenta funciona, e se seu desempenho é melhor com conjuntos de dados menores ou se foi projetada para executar suas próprias agregações com conjuntos de dados maiores, é importante. Não há uma resposta do tipo "uma para tudo". Quanto mais você souber sobre como os dados serão usados, mais preparado estará para modelar os dados apropriadamente.

Com frequência conjuntos de dados menores, agregados e altamente específicos funcionam melhor para visualizações, independentemente de terem sido criadas em um software comercial ou com o uso de uma linguagem de programação como R, Python ou JavaScript. Pense no nível

de agregação e de fatias, ou de outros elementos, que os usuários finais precisarão usar como base para a filtragem. Os conjuntos de dados podem requerer uma linha para cada fatia, assim como também podem requerer uma fatia que englobe "tudo". Você pode precisar unir (com *UNION*) duas consultas – uma no nível de detalhe e a outra no nível que engloba "tudo".

Na criação de saídas para pacotes estatísticos ou modelos de machine learning, é importante conhecer a entidade principal que está sendo estudada, o nível de agregação desejado e os atributos ou recursos necessários. Por exemplo, um modelo pode precisar de um registro por cliente com vários atributos, ou de um registro por transação com seus atributos e com os atributos do cliente. Geralmente, a saída para a modelagem segue a noção de "dados limpos" (tidy data) proposta por Hadley Wickham.[3] Os dados limpos têm estas propriedades:

1. Cada variável forma uma coluna.
2. Cada observação forma uma linha.
3. Cada valor é uma célula.

Examinaremos a seguir como usar SQL para transferir dados da estrutura na qual eles existem no banco de dados para outra estrutura pivotada ou despivotada que seja necessária para a análise.

Pivotando com instruções CASE

Uma *tabela dinâmica* é uma maneira de resumir conjuntos de dados organizando os dados em linhas, de acordo com os valores de um atributo, e em colunas, de acordo com os valores de outro atributo. Na interseção de cada linha e coluna, uma síntese estatística como `sum`, `count` ou `avg` é calculada. As tabelas dinâmicas costumam ser uma boa maneira de resumir dados para públicos-alvo empresariais, já que elas remodelam os dados em uma forma mais compacta e facilmente compreensível. Elas são amplamente conhecidas por sua implementação no Microsoft Excel, que tem uma interface de arrastar e soltar para a criação dos resumos de dados.

As tabelas dinâmicas, ou saída pivotada (pivoted output), podem ser criadas em SQL com o uso de uma instrução CASE junto com uma ou mais funções de agregação. Já vimos as instruções CASE várias vezes, e

remodelar dados é outro caso de uso importante em que elas estão envolvidas. Por exemplo, suponhamos que tivéssemos uma tabela `orders` (pedidos) com uma linha para cada compra feita pelos clientes. Para achatar os dados, use *GROUP BY* para fazer o agrupamento pelo `customer_id` (id do cliente) e some (com `sum`) a quantidade de pedidos (`order_amount`):

```
SELECT customer_id
,sum(order_amount) as total_amount
FROM orders
GROUP BY 1
;
customer_id  total_amount
-----------  ------------
123          59.99
234          120.55
345          87.99
...          ...
```

Para criar uma tabela dinâmica, geraremos adicionalmente colunas para cada um dos valores de um atributo. Suponhamos que a tabela `orders` também tivesse um campo `product` contendo o tipo de item comprado e a `order_date`. Para criar a saída pivotada, faça o agrupamento por (*GROUP BY*) data do pedido (`order_date`) e some o resultado de uma instrução CASE que retorne a `order_amount` sempre que a linha atender aos critérios de nome do produto:

```
SELECT order_date
,sum(case when product = 'shirt' then order_amount
          else 0
          end) as shirts_amount
,sum(case when product = 'shoes' then order_amount
          else 0
          end) as shoes_amount
,sum(case when product = 'hat' then order_amount
          else 0
          end) hats_amount
FROM orders
GROUP BY 1
```

```
;
order_date  shirts_amount  shoes_amount  hats_amount
----------  -------------  ------------  -----------
2020-05-01  5268.56        1211.65       562.25
2020-05-02  5533.84        522.25        325.62
2020-05-03  5986.85        1088.62       858.35
...         ...            ...           ...
```

Observe que, com a agregação `sum`, opcionalmente você pode usar "else 0" para evitar nulos no conjunto de resultados. No entanto, com `count` ou `count distinct`, você não deve incluir uma instrução ELSE, já que fazê-lo inflaria o conjunto de resultados. Isso ocorre porque o banco de dados não contará um nulo, mas contará um valor substituto, como zero.

É muito útil pivotar com instruções CASE, e conseguir fazer isso cria designs de tabela de data warehouse longos e estreitos em vez de amplos, o que pode ser melhor para o armazenamento de dados esparsos, porque adicionar colunas a uma tabela pode ser uma operação cara. Por exemplo, em vez de armazenar vários atributos de clientes em muitas colunas diferentes, uma tabela poderia conter diversos registros por cliente, com cada atributo em uma linha separada, e com os campos `attribute_name` e `attribute_value` especificando qual é o atributo e qual é seu valor. Os dados poderiam então ser pivotados quando necessário para unir um registro de cliente aos atributos desejados. Esse design é eficiente quando há muitos atributos esparsos (só um subconjunto de clientes tem valores para muitos atributos).

Pivotar dados com uma combinação de agregação e instruções CASE funciona bem quando há um número finito de itens para serem pivotados. Para quem já trabalhou com outras linguagens de programação, trata-se basicamente de um looping, mas escrito explicitamente linha a linha. Isso fornece um nível alto de controle, como em um caso em que, por exemplo, você quisesse calcular diferentes métricas em cada coluna, mas também pode ser tedioso. Pivotar com instruções CASE não funciona bem quando novos valores chegam constantemente ou são alterados rapidamente, já que o código SQL precisaria ser atualizado continuamente. Nesses casos, levar a computação para outra camada da pilha de análise, como uma ferramenta de BI ou

uma linguagem estatística, pode ser mais apropriado.

Despivotando com instruções UNION

Poderíamos ter o problema oposto e precisar mover dados armazenados em colunas para linhas em vez de criar dados limpos. Essa operação chama-se *despivotar*. Os conjuntos de dados que precisam ser despivotados são aqueles que estão em um formato de tabela dinâmica. Como exemplo, as populações de países norte-americanos em intervalos de 10 anos começando em 1980 são mostradas na Figura 2.4.

Country	year_1980	year_1990	year_2000	year_2010
Canada	24,593	27,791	31,100	34,207
Mexico	68,347	84,634	99,775	114,061
United States	227,225	249,623	282,162	309,326

Figura 2.4: População dos países por ano (em milhares).[4]

Para transformar isso em um conjunto de resultados com uma linha por país para cada ano, podemos usar um operador *UNION*. *UNION* é uma maneira de combinar conjuntos de dados de várias consultas em um único conjunto de resultados. Há dois formatos, *UNION* e *UNION ALL*. No uso de *UNION* ou *UNION ALL*, os números de colunas da consulta de cada componente devem coincidir. Os tipos de dados devem coincidir ou ser compatíveis (inteiros e floats podem ser combinados, mas inteiros e strings não). Os nomes das colunas do conjunto de resultados vêm da primeira consulta. Logo, fornecer aliases para os campos das consultas restantes é opcional, mas pode tornar a consulta mais fácil de ler:

```
SELECT country
,'1980' as year
,year_1980 as population
FROM country_populations
    UNION ALL
SELECT country
,'1990' as year
,year_1990 as population
FROM country_populations
    UNION ALL
```

```
SELECT country
,'2000' as year
,year_2000 as population
FROM country_populations
    UNION ALL
SELECT country
,'2010' as year
,year_2010 as population
FROM country_populations
;
country        year  population
-------------  ----  ----------
Canada         1980  24593
Mexico         1980  68347
United States  1980  227225
...            ...   ...
```

Nesse exemplo, usamos uma constante para embutir o ano em código, a fim de rastrear o ano ao qual o valor da população corresponde. Os valores embutidos em código podem ser de qualquer tipo, dependendo do caso de uso. Você pode ter de converter explicitamente certos valores embutidos em código, como na inserção de uma data:

```
'2020-01-01'::date as date_of_interest
```

Qual é a diferença entre *UNION* e *UNION ALL*? As duas podem ser usadas para acrescentar ou empilhar dados, mas são um pouco diferentes. *UNION* remove duplicidades do conjunto de resultados, enquanto *UNION ALL* retém todos os registros, sejam ou não duplicidades. *UNION ALL* é mais rápida, já que o banco de dados não precisa percorrer os dados para encontrar duplicidades. Ela também assegura que todos os registros entrem no conjunto de resultados. Costumo usar *UNION ALL*, e só uso *UNION* quando tenho motivos para achar que existem dados duplicados.

Usar *UNION* também pode ser útil na união de dados de diferentes fontes. Por exemplo, suponhamos que tivéssemos uma tabela `populations` com dados anuais por país e outra tabela `gdp` com o produto interno bruto anual, ou GDP (gross domestic product). Uma opção seria unir as tabelas com *JOIN* e obter um conjunto de resultados com uma coluna para a

população e outra para o GDP:

```
SELECT a.country, a.population, b.gdp
FROM populations a
JOIN gdp b on a.country = b.country
;
```

Outra opção é usar *UNION ALL* com os conjuntos de dados para obtermos um conjunto empilhado:

```
SELECT country, 'population' as metric, population as metric_value
FROM populations
    UNION ALL
SELECT country, 'gdp' as metric, gdp as metric_value
FROM gdp
;
```

A abordagem que você vai usar vai depender da saída que precisar para sua análise. A última opção pode ser útil quando você tiver várias métricas diferentes em tabelas distintas e nenhuma tabela tiver um conjunto completo de entidades (nesse caso, países). Essa é uma abordagem alternativa a uma *FULL OUTER JOIN*.

Funções pivot e unpivot

Reconhecendo que os casos de uso em que as pessoas pivotam e despivotam dados são comuns, alguns fornecedores de bancos de dados implementaram funções que fazem isso com menos linhas de código. O Microsoft SQL Server e o Snowflake têm funções `pivot` que assumem a forma de expressões adicionais na cláusula *WHERE*. Aqui, a agregação pode usar qualquer função, como `sum` ou `avg`, `value_column` é o campo a ser agregado, e uma coluna será criada para cada valor de `label_column` listado como rótulo:

```
SELECT...
FROM...
    pivot(aggregation(value_column)
          for label_column in (label_1, label_2, ...)
;
```

Poderíamos reescrever o exemplo de pivotagem anterior que usou instruções CASE da seguinte forma:

```
SELECT *
FROM orders
    pivot(sum(order_amount) for product in ('shirt','shoes'))
GROUP BY order_date
;
```

Embora essa sintaxe seja mais compacta do que a estrutura CASE que vimos anteriormente, as colunas desejadas ainda precisam ser especificadas. Como resultado, `pivot` não resolve o problema de conjuntos de campos recém-chegados ou que são alterados rapidamente que precisam ser transformados em colunas. O Postgres tem uma função `crosstab` semelhante, disponível no módulo `tablefunc`.

O Microsoft SQL Server e o Snowflake também têm funções `unpivot` que funcionam de maneira semelhante às expressões da cláusula *WHERE* e transformam linhas em colunas:

```
SELECT...
FROM...
    unpivot( value_column for label_column in (label_1, label_2,
  ...))
;
```

Por exemplo, os dados de `country_populations` do exemplo anterior poderiam ser remodelados da seguinte forma:

```
SELECT *
FROM country_populations
    unpivot(population for year in (year_1980, year_1990, year_2000,
  year_2010))
;
```

Aqui, novamente, a sintaxe é mais compacta do que a da abordagem de *UNION* ou *UNION ALL* que já examinamos, mas a lista de colunas deve ser especificada na consulta.

O Postgres tem uma função de array `unnest` que pode ser usada para despivotar dados, graças ao seu tipo de dado array. Um array é uma coleção de elementos, e no Postgres podemos listar os elementos de um array em colchetes. A função pode ser usada na cláusula *SELECT* e tem esta forma:

```
unnest(array[element_1, element_2, ...])
```

Voltando ao nosso exemplo anterior de países e populações, essa consulta retorna o mesmo resultado da consulta com as cláusulas *UNION ALL* repetidas:

```
SELECT
country
,unnest(array['1980', '1990', '2000', '2010']) as year
,unnest(array[year_1980, year_1990, year_2000, year_2010]) as pop
FROM country_populations
;
country  year  pop
-------  ----  -----
Canada   1980  24593
Canada   1990  27791
Canada   2000  31100
...      ...   ...
```

Os conjuntos de dados chegam em diferentes formatos, e nem sempre estão no formato necessário para nossa saída. Há muitas opções de remodelagem quando pivotamos ou despivotamos dados, seja com instruções CASE ou *UNION*s, ou com funções específicas do banco de dados. Saber como manipular seus dados para modelá-los da maneira que você deseja dará maior flexibilidade para sua análise e para como apresentará seus resultados.

Conclusão

Preparar dados para a análise pode parecer um trabalho que fazemos antes de chegar à tarefa real de análise, mas ele é tão fundamental para o entendimento dos dados que eu sempre o considero tempo bem gasto. Conhecer os diferentes tipos de dados que você pode encontrar é crítico, e é preciso reservar algum tempo para entender os dados de cada tabela envolvida no trabalho. Criar perfis de dados nos ajuda a aprender mais sobre o que há no conjunto de dados e a examinar sua qualidade. Volto com frequência à criação de perfis no decorrer de meus projetos de análise, à medida que aprendo mais sobre os dados e preciso verificar os resultados de minha consulta quando a complexidade aumenta. Provavelmente a qualidade dos dados nunca deixará de ser um problema;

logo, examinamos algumas maneiras de manipular a limpeza e o acréscimo de dados em conjuntos de dados. Para concluir, saber como modelar os dados para criar o formato de saída correto é essencial. Veremos esses tópicos novamente no contexto de várias análises no decorrer do livro. O próximo capítulo, sobre análise de séries temporais, começa nossa jornada em direção a técnicas de análise específicas.

---

1 John W. Tukey, *Exploratory Data Analysis* (Reading, MA: Addison-Wesley, 1977).

2 N.T.: As propriedades na internet são os URLS, os nomes de domínio registrados e outros nomes associados a endereços e sites na internet.

3 Hadley Wickham, "Tidy Data", *Journal of Statistical Software* 59, n. 10 (2014): 1 – 23, *https://doi.org/10.18637/jss.v059.i10*.

4 US Census Bureau, "International Data Base (IDB)", atualizado pela última vez em dezembro de 2020, *https://www.census.gov/data-tools/demo/idb*.

CAPÍTULO 3

Análise de séries temporais

Agora que abordamos o SQL, os bancos de dados e as principais etapas da preparação de dados para a análise, é hora de examinarmos os tipos específicos de análises que podem ser executados com SQL. Aparentemente há um número interminável de conjuntos de dados em todo o mundo e maneiras igualmente infinitas pelas quais eles podem ser analisados. Neste capítulo e nos próximos, organizei os tipos de análises em temas que espero serem úteis à medida que você for construindo suas habilidades em análise e SQL. Muitas das técnicas que serão discutidas se baseiam nas mostradas no Capítulo 2 e depois nos capítulos anteriores conforme o livro avançar. As séries de dados temporais são tão predominantes e importantes que começarei a sequência de temas de análise com elas.

A análise de séries temporais é um dos tipos mais comuns de análises feitos com SQL. Uma *série temporal* é uma sequência de medidas ou pontos de dados registrados em ordem temporal, geralmente em intervalos regularmente espaçados. Há muitos exemplos de dados de séries temporais no dia a dia, como a temperatura máxima diária, o valor de fechamento do índice S&P 500 da bolsa de valores ou o número de passos diários registrados por um rastreador de fitness. A análise de séries temporais é usada em uma ampla variedade de segmentos industriais e disciplinas, da estatística e a engenharia à previsão do tempo e ao planejamento empresarial. É uma maneira de entender e quantificar como as coisas mudam com o tempo.

A previsão é um objetivo comum da análise de séries temporais. Já que o tempo só caminha para a frente, valores futuros podem ser expressos como uma função de valores passados, mas o oposto não ocorre. No entanto, é importante notar que o passado não prevê o futuro com perfeição. Qualquer número de alterações em condições mais amplas de mercado, tendências populares, lançamentos de produtos ou outras

grandes alterações dificultam a realização de previsões. Mesmo assim, examinar dados históricos pode levar a insights e ao desenvolvimento de um conjunto de resultados possíveis útil para o planejamento. Enquanto escrevo este texto, o planeta está no meio de uma pandemia global de COVID-19, algo diferente de qualquer coisa vista nos últimos 100 anos – anos antes das histórias das organizações começarem, exceto as de mais longa data. Logo, muitas organizações atuais ainda não haviam visto esse evento específico, mas sobreviveram a outras crises econômicas, como as posteriores à explosão da bolha da internet e aos ataques de 11/9 em 2001, assim como à crise financeira global de 2007–2008. Com uma análise cuidadosa e a compreensão do contexto, geralmente podemos extrair insights úteis.

Neste capítulo, primeiro abordaremos os elementos constitutivos do SQL para a análise de séries temporais: a sintaxe e as funções para o trabalho com datas, timestamps e horas. Em seguida, introduzirei o conjunto de dados de vendas no varejo usado nos exemplos do decorrer do capítulo. Uma discussão dos métodos de análise de tendências vem a seguir e depois abordarei o cálculo de janelas de tempo contínuas. Veremos então os cálculos de período a período para analisar dados com componentes de sazonalidade. Para concluir, fornecerei algumas técnicas adicionais úteis para a análise de séries temporais.

Manipulações de data, data/hora e hora

As datas e as horas podem ter vários formatos, dependendo da origem dos dados. Podemos precisar ou querer transformar o formato dos dados brutos da saída ou fazer cálculos para chegar a novas datas ou partes de datas. Por exemplo, o conjunto de dados poderia conter timestamps das transações, enquanto o objetivo da análise seria encontrar a tendência das vendas mensais. Também poderíamos querer saber quantos dias ou meses se passaram desde a ocorrência de um evento específico. Felizmente, o SQL tem funções e recursos de formatação poderosos que podem transformar quase qualquer entrada bruta em praticamente qualquer saída da qual possamos precisar para a análise.

Nesta seção, mostrarei como fazer a conversão entre fusos horários e

depois darei detalhes sobre a formatação de datas e datas/horas. Em seguida, explorarei a matemática de datas e as manipulações de horas, incluindo as que fazem uso de intervalos. Um intervalo é um tipo dado que contém um período de tempo, como um número de meses, dias ou horas. Embora os dados possam ser armazenados em uma tabela de banco de dados com o tipo intervalo, raramente vejo isso ocorrer na prática; logo, falarei sobre intervalos junto com as funções de data e hora com as quais eles podem ser usados. Por fim, discutirei algumas considerações especiais referentes à junção ou à combinação de dados de diferentes fontes.

Conversões de fuso horário

Conhecer o fuso horário padrão usado em um conjunto de dados pode evitar compreensões incorretas e erros posteriormente no processo de análise. Os fusos horários dividem o mundo nas regiões norte–sul, que vivenciam o mesmo horário. Eles permitem que diferentes partes do mundo tenham horários semelhantes para os períodos diário e noturno – assim, por exemplo, o sol estará a pino ao meio-dia onde quer que você esteja. Os fusos seguem limites irregulares que são tanto políticos quanto geográficos. A maioria tem uma hora de diferença, mas alguns têm um deslocamento de somente 30 ou 45 minutos e, portanto, mais de 30 fusos horários abrangem o globo terrestre. Muitos países que estão distantes do equador também se beneficiam do horário de verão durante partes do ano, mas há exceções, como nos Estados Unidos e na Austrália, onde alguns estados usam o horário de verão e outros não. Cada fuso horário tem uma abreviação padrão, como PST para Pacific Standard Time (Horário Padrão do Pacífico) e PDT para Pacific Daylight Time (Horário de Verão do Pacífico).

Muitos bancos de dados são configurados com o *UTC (Coordinated Universal Time, Tempo Universal Coordenado)*, o padrão global usado para regular relógios e registrar eventos nesse fuso horário. Ele substituiu o *GMT (Greenwich Mean Time, Tempo Médio de Greenwich)*, que você ainda poderá ver se seus dados vierem de um banco de dados mais antigo. O UTC não tem horário de verão, logo permanece consistente o ano todo. Isso acaba sendo muito útil para a análise. Lembro-me de uma vez em que um gerente de

produto em pânico me pediu para descobrir por que as vendas de um domingo específico caíram tanto em comparação com o domingo anterior. Passei horas escrevendo consultas e investigando possíveis causas antes de detectar que nossos dados tinham sido registrados no PT (Pacific Time, Fuso Horário do Pacífico). O horário de verão começou cedo na manhã de domingo, o relógio do banco de dados se adiantou 1 hora, o dia teve apenas 23 horas em vez de 24 e, portanto, as vendas pareceram cair. Meio ano depois tivemos um dia correspondente de 25 horas, quando as vendas pareceram excepcionalmente altas.

Geralmente os timestamps do banco de dados não são codificados com o fuso horário, e você terá de consultar a origem ou o desenvolvedor para descobrir como seus dados foram armazenados. O UTC tem sido mais comum nos conjuntos de dados que costumo ver, mas é claro que isso não é universal.

Uma desvantagem do UTC, ou na verdade de qualquer logging de horas de máquina, é que perdemos informações sobre o fuso horário local do humano que está executando as ações que geraram o evento registrado no banco de dados. Eu poderia querer saber se as pessoas tendem a usar mais minha aplicação móvel durante o dia de trabalho ou durante as noites e fins de semana. Se meu público-alvo estiver agrupado em um único fuso horário, não será difícil descobrir isso. No entanto, se ele se estender por vários fusos horários ou for internacional, será necessário calcular a conversão de cada hora registrada para seu fuso horário local.

Todos os fusos horários locais têm um deslocamento UTC. Por exemplo, o deslocamento para o PDT é UTC – 7 horas, enquanto o deslocamento para o PST é UTC – 8 horas. Os timestamps dos bancos de dados são armazenados no formato AAAA-MM-DD hh:mi:ss (que significa anos-meses-dias horas:minutos:segundos). Timestamps com fuso horário têm uma informação adicional para o deslocamento UTC, expresso como um número positivo ou negativo. A conversão de um fuso horário para outro pode ser feita com a função `at time zone` seguida da abreviação do fuso horário de destino. Por exemplo, podemos converter um timestamp do UTC (deslocamento – 0) para o PST:

```
SELECT '2020-09-01 00:00:00 -0' at time zone 'pst';
timezone
-------------------
```

```
2020-08-31 16:00:00
```

O nome do fuso horário de destino pode ser uma constante, ou um campo do banco de dados, permitindo que essa conversão seja dinâmica para o conjunto de dados. Alguns bancos de dados têm uma função `convert_timezone` ou `convert_tz` que opera de maneira semelhante. Um argumento é o fuso horário do resultado e o outro é o fuso horário para o qual será feita a conversão:

```
SELECT convert_timezone('pst','2020-09-01 00:00:00 -0');
timezone
-------------------
2020-08-31 16:00:00
```

Verifique a documentação do seu banco de dados para saber o nome e a ordem exatos dos argumentos de fuso horário de destino e de timestamp da origem. Muitos bancos de dados contêm uma lista com os fusos horários e suas abreviações em uma tabela do sistema. Algumas comuns podem ser vistas na Tabela 3.1. Elas podem ser consultadas com *SELECT * FROM* e o nome da tabela. A Wikipédia também tem uma lista útil de abreviações de fusos horários padrão e seus deslocamentos UTC (*https://oreil.ly/im0wi*).

Tabela 3.1: Tabelas do sistema com informações de fusos horários em bancos de dados comuns

Postgres	`pg_timezone_names`
MySQL	`mysql.time_zone_names`
SQL Server	`sys.time_zone_info`
Redshift	`pg_timezone_names`

Os fusos horários são uma parte natural do trabalho com timestamps. Com as funções de conversão de fuso horário, é possível mover-se entre o fuso horário no qual os dados foram registrados e qualquer outro fuso horário mundial. A seguir, mostrarei várias técnicas para a manipulação de datas e timestamps com SQL.

Conversões de formato de data e timestamp

As datas e os timestamps são essenciais para a análise de séries temporais.

Devido às diversas maneiras pelas quais as datas e horas podem ser representadas nos dados de origem, é quase inevitável que você precise converter formatos de data em algum momento. Nesta seção, abordarei várias conversões comuns e como executá-las com SQL: alterando o tipo de dado, extraindo partes de uma data ou de um timestamp, e criando uma data ou um timestamp com base nas partes. Começarei introduzindo algumas funções úteis que retornam a data e/ou hora atuais.

Retornar a data ou a hora atual é uma tarefa de análise comum – por exemplo, para incluir um timestamp no conjunto de resultados ou usar na matemática de datas, abordada na próxima seção. A data e a hora atuais são chamadas de *tempo do sistema (system time)*, e, embora seja fácil retorná-lo com SQL, há algumas diferenças de sintaxe entre os bancos de dados.

Para retornar a data atual alguns bancos de dados têm uma função `current_date`, sem parênteses:

```
SELECT current_date;
```

Há uma variedade maior de funções para o retorno da data e hora atuais. Verifique a documentação do seu banco de dados ou apenas faça testes digitando em uma janela SQL para ver se uma função retorna um valor ou um erro. As funções com parênteses não recebem argumentos, mas é importante incluir os parênteses:

```
current_timestamp
localtimestamp
get_date()
now()
```

Para concluir, há funções que retornam apenas a parte do tempo do sistema atual referente ao timestamp. Novamente, consulte a documentação ou faça testes para descobrir que função(ões) você deve usar com seu banco de dados:

```
current_time
localtime
timeofday()
```

O SQL tem várias funções para a alteração do formato de datas e horas. Para reduzir a granularidade de um timestamp, use a função `date_trunc`. O primeiro argumento é um valor textual indicando o período de tempo

para o qual o timestamp, que é o segundo argumento, deve ser truncado. O resultado é um valor de timestamp:

```
date_trunc (text, timestamp)
SELECT date_trunc('month','2020-10-04 12:33:35'::timestamp);
date_trunc
-------------------
2020-10-01 00:00:00
```

Os argumentos padrão que podem ser usados estão listados na Tabela 3.2. Eles se estendem de microssegundos a milênios, fornecendo muita flexibilidade. Bancos de dados que não suportam `date_trunc`, como o MySQL, têm uma função alternativa chamada `date_format` que pode ser usada de maneira semelhante:

```
SELECT date_format('2020-10-04 12:33:35','%Y-%m-01') as date_trunc;
date_trunc
-------------------
2020-10-01 00:00:00
```

Tabela 3.2: Argumentos padrão de período de tempo

Argumentos de período de tempo	
`microsecond` (microssegundo)	`month` (mês)
`millisecond` (milissegundo)	`quarter` (trimestre)
`second` (segundo)	`year` (ano)
`minute` (minuto)	`decade` (década)
`hour` (hora)	`century` (século)
`day` (dia)	`millennium` (milênio)
`week` (semana)	

Em vez de retornar datas ou timestamps, nossa análise pode precisar de partes de datas ou horas. Por exemplo, poderíamos querer agrupar as vendas por mês, por dia da semana ou por hora do dia.

O SQL fornece algumas funções para o retorno apenas da parte requerida da data ou do timestamp. Geralmente as datas e os timestamps são intercambiáveis, exceto quando a solicitação é para retornar uma parte da hora. Nesses casos, é claro que a hora é necessária.

A função `date_part` recebe um valor textual para a parte a ser retornada e

um valor de data ou timestamp. O valor retornado é um FLOAT, que é um valor numérico com uma parte decimal; dependendo das suas necessidades, você pode converter o valor para um tipo de dado inteiro:

```
SELECT date_part('day',current_timestamp);
SELECT date_part('month',current_timestamp);
SELECT date_part('hour',current_timestamp);
```

Outra função que opera de maneira semelhante é `extract`, que recebe o nome da parte desejada e um valor de data ou timestamp e retorna um valor FLOAT:

```
SELECT extract('day' from current_timestamp);
date_part
---------
27.0
SELECT extract('month' from current_timestamp);
date_part
---------
5.0
SELECT extract('hour' from current_timestamp);
date_part
---------
14.0
```

As funções `date_part` e `extract` podem ser usadas com intervalos, mas lembre-se de que a parte solicitada deve estar na mesma unidade do intervalo. Logo, por exemplo, solicitar dias de um intervalo definido em dias retorna o valor esperado igual a 30:

```
SELECT date_part('day',interval '30 days');
SELECT extract('day' from interval '30 days');
date_part
---------
30.0
```

No entanto, solicitar dias de um intervalo definido em meses retorna o valor 0.0:

```
SELECT extract('day' from interval '3 months');
date_part
---------
0.0
```

 Uma lista completa das partes de data pode ser encontrada na documentação de seu banco de dados ou com uma busca online, porém as mais comuns são "day" (dia), "month" (mês) e "year" (ano) para datas e "second" (segundo) "minute" (minuto) e "hour" (hora) para timestamps.

Para retornar os valores textuais das partes de data, use a função `to_char`, que recebe o valor da entrada e o formato da saída como argumentos:

```
SELECT to_char(current_timestamp,'Day');
SELECT to_char(current_timestamp,'Month');
```

 Se você encontrar timestamps armazenados como Era Unix (quantos segundos se passaram desde 1º de janeiro de 1970, à 00:00:00 UTC), poderá convertê-los para timestamps usando a função `to_timestamp`.

A análise pode requerer a criação de uma data baseada em partes provenientes de diferentes fontes. Isso pode ocorrer quando os valores de ano, mês e dia estiverem armazenados em diferentes colunas do banco de dados. Também pode ser necessário quando as partes tiverem sido extraídas do texto por parsing, um tópico que abordarei com mais detalhes no Capítulo 5.

Uma maneira simples de criar um timestamp a partir de componentes separados de data e hora é concatenando-os com um sinal de adição (+):

```
SELECT date '2020-09-01' + time '03:00:00' as timestamp;
timestamp
-------------------
2020-09-01 03:00:00
```

Uma data pode ser montada com o uso da função `make_date`, `makedate`, `date_from_parts` ou `datefromparts`. Essas funções são equivalentes, mas os bancos de dados as nomeiam de maneiras diferentes. A função recebe argumentos para as partes do ano, do mês e do dia e retorna um valor com formato de data:

```
SELECT make_date(2020,09,01);
make_date
----------
2020-09-01
```

Os argumentos podem ser constantes ou referenciar nomes de campos e devem ser inteiros. Outra maneira de montar uma data ou um timestamp

é concatenando os valores e convertendo o resultado em um formato de data usando uma das sintaxes de conversão ou a função `to_date`:

```
SELECT to_date(concat(2020,'-',09,'-',01), 'yyyy-mm-dd');
to_date
----------
2020-09-01
SELECT cast(concat(2020,'-',09,'-',01) as date);
to_date
----------
2020-09-01
```

O SQL tem várias maneiras de formatar e converter datas e timestamps e recuperar datas e horas do sistema. Na próxima seção, começarei a usá-las na matemática de datas.

Matemática de datas

O SQL permite executar várias operações matemáticas com datas. Isso pode surpreender já que, a rigor, as datas não são tipos de dados numéricos, mas o conceito deve ser familiar se você já tentou descobrir que dia será daqui a quatro semanas. A matemática de datas é útil para várias tarefas de análise. Por exemplo, podemos usá-la para ter informações sobre a idade ou a permanência de um cliente, saber quanto tempo se passou entre dois eventos e conhecer quantos eventos ocorreram dentro de uma janela de tempo.

A matemática de datas envolve dois tipos de dados: as datas propriamente ditas e os intervalos. Precisamos do conceito de intervalos porque os componentes de data e hora não se comportam como os inteiros. Um décimo de 100 é 10; um décimo de um ano é 36,5 dias. Metade de 100 é 50; metade de um dia é 12 horas. Os intervalos permitem nos movermos facilmente entre unidades de tempo. Há dois tipos de intervalos: ano-mês e dia-hora. Começaremos com algumas operações que retornam valores inteiros e depois passaremos para as funções que operam com ou retornam intervalos.

Primeiro, veremos quantos dias se passaram entre duas datas. Há várias maneiras de fazer isso em SQL. A primeira é usando um operador matemático, o sinal de subtração (-):

```
SELECT date('2020-06-30') - date('2020-05-31') as days;
days
----
30
```

Esse código retorna quantos dias existem entre essas duas datas. Observe que a resposta é 30 dias, e não 31. O número de dias inclui apenas uma das extremidades. Subtrair as datas na ordem inversa também funciona e retorna um intervalo de -30 dias:

```
SELECT date('2020-05-31') - date('2020-06-30') as days;
days
----
-30
```

Também podemos encontrar a diferença entre duas datas com a função `datediff`. O Postgres não dá suporte a essa função, mas outros bancos de dados populares dão, incluindo o SQL Server, o Redshift e o Snowflake, e ela é muito útil, principalmente quando o objetivo é retornar um intervalo que não seja o número de dias. A função recebe três argumentos – as unidades de período de tempo que você deseja retornar, um timestamp ou data inicial e um timestamp ou data final:

```
datediff(interval_name, start_timestamp, end_timestamp)
```

Logo, nosso exemplo anterior ficaria assim:

```
SELECT datediff('day',date('2020-05-31'), date('2020-06-30')) as
  days;
days
----
30
```

Também podemos encontrar quantos meses existem entre duas datas, e o banco de dados fará o cálculo correto ainda que os meses tenham tamanhos diferentes ao longo do ano:

```
SELECT datediff('month'
            ,date('2020-01-01')
            ,date('2020-06-30')
            ) as months;
months
------
```

```
5
```

No Postgres, isso pode ser feito com o uso da função `age`, que calcula o intervalo entre duas datas:

```
SELECT age(date('2020-06-30'),date('2020-01-01'));
age
--------------
5 mons 29 days
```

Também podemos encontrar o número de componentes mensais do intervalo com a função `date_part()`:

```
SELECT date_part('month',age('2020-06-30','2020-01-01')) as months;
months
------
5.0
```

Subtrair datas para saber quanto tempo se passou entre elas é um recurso poderoso. A soma das datas não funciona da mesma forma. Para fazer a soma com datas, temos de usar intervalos ou funções especiais. Por exemplo, podemos adicionar sete dias a uma data acrescentando o intervalo `'7 days'`:

```
SELECT date('2020-06-01') + interval '7 days' as new_date;
new_date
-------------------
2020-06-08 00:00:00
```

Alguns bancos de dados não requerem o uso da sintaxe de intervalo e, em vez disso, convertem automaticamente o número fornecido para dias, embora geralmente seja boa prática usar a notação de intervalo, tanto para estimular a compatibilidade entre os bancos de dados quanto para tornar o código mais fácil de ler:

```
SELECT date('2020-06-01') + 7 as new_date;
new_date
-------------------
2020-06-08 00:00:00
```

Se quiser adicionar uma unidade de tempo diferente, use a notação de intervalo com meses, anos, horas ou outro período de data ou hora. Observe que esse recurso também pode ser usado para fazer a subtração

entre intervalos e datas com o uso de "-" em vez de "+". Muitos bancos de dados, mas não todos, têm uma função `date_add` ou `dateadd` que recebe o intervalo desejado, um valor e a data inicial e faz o cálculo:

```
SELECT date_add('month',1,'2020-06-01') as new_date;
new_date
----------
2020-07-01
```


Consulte a documentação do seu banco de dados, ou apenas faça testes com consultas, para saber que sintaxe e funções estão disponíveis e são apropriadas para seu projeto.

Além de na cláusula *SELECT*, essas fórmulas também podem ser usadas na cláusula *WHERE*. Por exemplo, podemos fazer uma filtragem para obter os registros que ocorreram há pelo menos três meses:

```
WHERE event_date < current_date - interval '3 months'
```

Elas também podem ser usadas em condições de *JOIN*, mas lembre-se de que geralmente o desempenho do banco de dados fica mais lento quando a condição de *JOIN* contém um cálculo em vez de uma igualdade ou diferença entre datas.

O uso da matemática de datas é comum na análise com SQL, tanto para encontrar o tempo passado entre datas ou timestamps quanto para calcular novas datas com base em um intervalo proveniente de uma data conhecida. Há várias maneiras de encontrar o tempo que se passou entre datas, adicionar intervalos a datas e subtrair intervalos de datas. A seguir, veremos as manipulações de horas, que são semelhantes.

Matemática de horas

A matemática de horas é menos comum em muitas áreas da análise, mas pode ser útil em algumas situações. Por exemplo, poderíamos querer saber quanto tempo leva para um representante do atendimento ao cliente responder a uma chamada telefônica em um call center ou para responder a um email solicitando ajuda. Sempre que o tempo passado entre dois eventos for menor do que um dia, ou quando o arredondamento do resultado para um número de dias não fornecer informações suficientes, a manipulação de horas poderá ser usada. A matemática de horas funciona

de maneira semelhante à matemática de datas, com o uso de intervalos. Podemos adicionar intervalos de tempo às horas:

```
SELECT time '05:00' + interval '3 hours' as new_time;
new_time
--------
08:00:00
```

Podemos subtrair intervalos de horas:

```
SELECT time '05:00' - interval '3 hours' as new_time;
new_time
--------
02:00:00
```

Também podemos subtrair horas para obter um intervalo:

```
SELECT time '05:00' - time '03:00' as time_diff;
time_diff
---------
02:00:00
```

As horas, ao contrário das datas, podem ser multiplicadas:

```
SELECT time '05:00' * 2 as time_multiplied;
time_multiplied
---------------
10:00:00
```

Os intervalos também podem ser multiplicados, resultando em um valor temporal:

```
SELECT interval '1 second' * 2000 as interval_multiplied;
interval_multiplied
-------------------
00:33:20
SELECT interval '1 day' * 45 as interval_multiplied;
interval_multiplied
-------------------
45 days
```

Esses exemplos estão usando valores constantes, mas você também pode incluir nomes de campos do banco de dados ou cálculos na consulta SQL para tornar os cálculos dinâmicos. Agora discutiremos considerações de

data especiais das quais devemos nos lembrar ao combinar conjuntos de dados de diferentes sistemas de origem.

Unindo dados de diferentes origens

Combinar dados de diferentes origens é um dos casos de uso mais úteis para um data warehouse. No entanto, diferentes sistemas de origem podem registrar datas e horas em formatos ou fusos horários distintos ou com algum deslocamento temporal devido a problemas com a hora do relógio interno do servidor. Até mesmo tabelas da mesma fonte de dados podem ter diferenças, embora seja menos comum. Conciliar e padronizar datas e timestamps são etapas importantes antes de podermos avançar na análise.

Datas e timestamps que estiverem em formatos diferentes podem ser padronizados com SQL. Realizar junções com datas ou incluir campos de dados em *UNION*s geralmente requer que as datas ou timestamps estejam no mesmo formato. Anteriormente no capítulo 1, mostrei técnicas para a formatação de datas e timestamps que ajudarão nesses problemas. Tome cuidado com fusos horários quando combinar dados de diferentes origens. Por exemplo, um banco de dados interno poderia estar usando o horário UTC, mas os dados de terceiros estarem em um fuso horário local. Vi dados originários de SaaS (software as a service, software como serviço) que tinham sido registrados em diferentes fusos horários locais. Repare que os valores dos timestamps também não têm necessariamente o fuso horário embutido. Você pode ter de consultar a documentação do fornecedor e converter os dados para UTC se o restante de seus dados tiver sido armazenado dessa forma. Outra opção é armazenar o fuso horário em um campo para que o valor do timestamp possa ser convertido quando necessário.

Outro item com o qual devemos tomar cuidado ao trabalhar com dados de diferentes origens são timestamps que estejam um pouco fora de sincronia. Isso pode ocorrer quando os timestamps forem registrados a partir de dispositivos clientes – por exemplo, a partir de um laptop ou celular em uma fonte de dados e a partir de um servidor na outra fonte de dados. Uma vez vi uma série de resultados de um experimento serem calculados erroneamente porque o dispositivo móvel cliente que

registrava a ação de um usuário tinha uma diferença de alguns minutos em relação ao servidor que registrou o grupo de tratamento ao qual o usuário foi atribuído. Os dados dos clientes móveis pareciam chegar antes do timestamp do grupo de tratamento; logo, alguns eventos foram inadvertidamente excluídos. Uma correção para algo assim é relativamente fácil de obter: em vez de fazer a filtragem por timestamps de ação maiores do que o timestamp do grupo de tratamento, permita que eventos dentro de um intervalo ou janela de tempo curto anterior ao timestamp do grupo de tratamento sejam incluídos nos resultados. Isso pode ser feito com uma cláusula *BETWEEN* e a matemática de datas, como visto na última seção.

Ao trabalhar com dados de aplicações móveis, preste bastante atenção em se os timestamps representam quando a ação ocorreu no dispositivo *ou* quando o evento chegou no banco de dados. A diferença pode variar de insignificante a um período de dias, dependendo de se a aplicação móvel permite o uso offline e de como ela manipula o envio de dados durante períodos de baixa intensidade do sinal. Os dados de aplicações móveis podem chegar com algum atraso ou podem percorrer seu caminho até o banco de dados dias após terem ocorrido no dispositivo. As datas e os timestamps também podem ser adulterados no trajeto e você pode encontrar ocorrências que estejam incrivelmente distantes no passado ou no futuro como resultado.

Agora que mostrei como manipular datas, datas/horas e horas alterando os formatos, convertendo fusos horários, usando a matemática de datas e trabalhando com conjuntos de dados de diferentes origens, estamos prontos para examinar alguns exemplos de séries temporais. Primeiro, introduzirei o conjunto de dados dos exemplos do restante do capítulo.

Conjunto de dados de vendas do varejo

Os exemplos do restante deste capítulo usarão um conjunto de dados de vendas mensais do varejo dos EUA extraído do *Monthly Retail Trade Report: Retail and Food Services Sales: Excel (1992-dias atuais)* (*https://www.census.gov/retail/index.html#mrts*) e disponível no site *Census.gov* (*http://Census.gov*). Os dados desse relatório são usados como indicador

econômico para o entendimento das tendências mostradas nos padrões de gastos dos consumidores dos EUA. Embora os valores do GDP (produto interno bruto) sejam publicados trimestralmente, os dados de vendas no varejo são publicados mensalmente, logo também são utilizados para ajudar a prever o GDP. Por essas duas razões, geralmente os valores mais recentes são abordados na imprensa empresarial quando são divulgados.

Os dados se estendem de 1992 a 2020 e incluem tanto os totais de vendas quanto detalhes de subcategorias de vendas no varejo. Eles contêm valores não ajustados e ajustados sazonalmente. Este capítulo usará os valores não ajustados, já que um dos objetivos é analisar a sazonalidade. Os valores das vendas estão em milhões de dólares americanos. O formato original do arquivo é o de um arquivo do Excel, com uma aba para cada ano e com meses como colunas. O site do GitHub referente a este livro (*https://oreil.ly/LMiHw*) apresenta os dados em um formato que é mais fácil de importar para um banco de dados, junto com um código especificamente para a importação para o Postgres. A Figura 3.1 exibe uma amostra da tabela `retail_sales`.

*	sales_month	naics_code	kind_of_business	reason_for_null	sales
1	2020-01-01	441	Motor vehicle and parts dealers	(null)	93268
2	2020-01-01	4411	Automobile dealers	(null)	80728
3	2020-01-01	4411, 4412	Automobile and other motor vehicle dealers	(null)	85823
4	2020-01-01	44111	New car dealers	(null)	71757
5	2020-01-01	44112	Used car dealers	(null)	8971
6	2020-01-01	4413	Automotive parts, acc., and tire stores	(null)	7445
7	2020-01-01	442	Furniture and home furnishings stores	(null)	9257
8	2020-01-01	442, 443	Furniture, home furn, electronics, and appliance stores	(null)	16993
9	2020-01-01	4421	Furniture stores	(null)	4904
10	2020-01-01	4422	Home furnishings stores	(null)	4353
11	2020-01-01	44221	Floor covering stores	Supressed	(null)
12	2020-01-01	442299	All other home furnishings stores	(null)	2408
13	2020-01-01	443	Electronics and appliance stores	(null)	7736
14	2020-01-01	443141	Household appliance stores	(null)	1197
15	2020-01-01	443142	Electronics stores	(null)	6539
16	2020-01-01	444	Building mat. and garden equip. and supplies dealers	(null)	27887
17	2020-01-01	4441	Building mat. and supplies dealers	(null)	24555
18	2020-01-01	44412	Paint and wallpaper stores	(null)	903
19	2020-01-01	44413	Hardware stores	(null)	1902
20	2020-01-01	445	Food and beverage stores	(null)	63590
21	2020-01-01	4451	Grocery stores	(null)	57667
22	2020-01-01	44511	Supermarkets and other grocery (except convenience) stores	(null)	55178
23	2020-01-01	4453	Beer, wine, and liquor stores	(null)	4388
24	2020-01-01	446	Health and personal care stores	(null)	30047
25	2020-01-01	44611	Pharmacies and drug stores	(null)	25209

Figura 3.1: Visualização do conjunto de dados de vendas do varejo dos EUA.

Encontrando tendências nos dados

Quando usamos dados de série temporal, geralmente queremos procurar tendências neles. Uma tendência é simplesmente a direção para a qual os dados estão se movendo. Eles podem estar se movendo para cima ou aumentando com o tempo, ou podem estar se movendo para baixo ou diminuindo com o tempo. Podem permanecer mais ou menos nivelados ou pode haver muito ruído, ou movimentos para cima e para baixo, e isso dificulte determinar uma tendência. Esta seção abordará várias técnicas para a busca de tendências em dados de série temporal, desde tendências simples para a criação de gráficos à comparação dos componentes de uma tendência, o uso do percentual em cálculos de totais para comparar as partes com o todo e, para concluir, a indexação para detectar a alteração percentual a partir de um período de referência.

Tendências simples

Criar uma tendência pode ser uma etapa da criação de perfis e compreensão dos dados ou pode ser a saída final. O conjunto de resultados é uma série de datas ou timestamps e um valor numérico. Quando uma série temporal é representada em gráfico, as datas e os timestamps ficam no eixo x e o valor numérico fica no eixo y. Por exemplo, poderíamos verificar a tendência do total de vendas do *varejo* e do setor de *serviços* alimentícios nos EUA:

```
SELECT sales_month
,sales
FROM retail_sales
WHERE kind_of_business = 'Retail and food services sales, total'
;
sales_month  sales
-----------  ------
1992-01-01   146376
1992-02-01   147079
1992-03-01   159336
...          ...
```

Os resultados estão no gráfico da Figura 3.2.

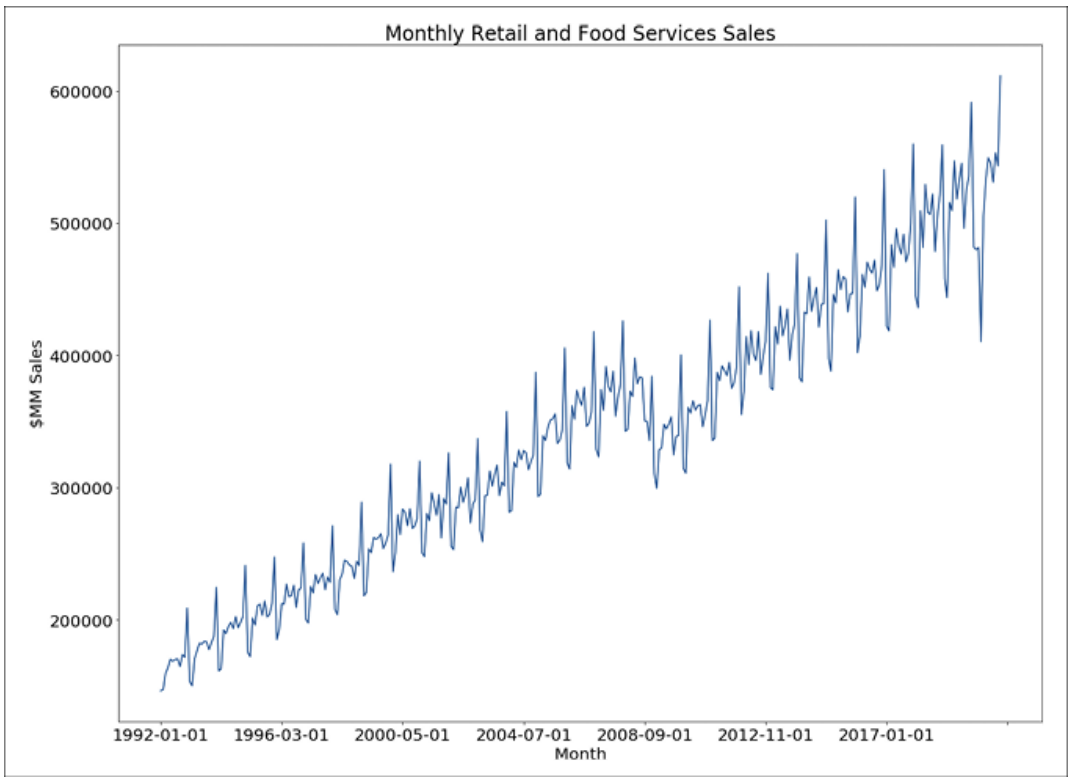


Figura 3.2: Tendências das vendas mensais do varejo e do setor de serviços alimentícios.

Claramente esses dados têm alguns padrões, mas eles também apresentam ruído. Transformar os dados e agregá-los em nível anual pode nos ajudar a ter uma melhor compreensão. Primeiro, usaremos a função `date_part` para retornar apenas o ano a partir do campo `sales_month` e depois somaremos as vendas. Os resultados serão filtrados de acordo com o valor "Retail and food services sales, total" (total de vendas do *varejo* e do setor de *serviços* alimentícios) do atributo `kind_of_business` (tipo de negócio) na cláusula *WHERE*:

```
SELECT date_part('year',sales_month) as sales_year
,sum(sales) as sales
FROM retail_sales
WHERE kind_of_business = 'Retail and food services sales, total'
GROUP BY 1
;
sales_year  sales
----------  -------
1992.0      2014102
1993.0      2153095
```

```
1994.0      2330235
...         ...
```

Ao representar esses dados em gráfico, como na Figura 3.3, vemos que agora temos uma série temporal mais branda que quase sempre aumenta com o tempo, como era de se esperar, mesmo que os valores das vendas não tenham sido ajustados conforme a inflação. As vendas de todo o varejo e do setor de serviços alimentícios caíram em 2009, durante a crise financeira global. Após crescer em todos os anos no decorrer da década de 2010, elas ficaram uniformes em 2020 em comparação com 2019, devido ao impacto da pandemia de COVID-19.

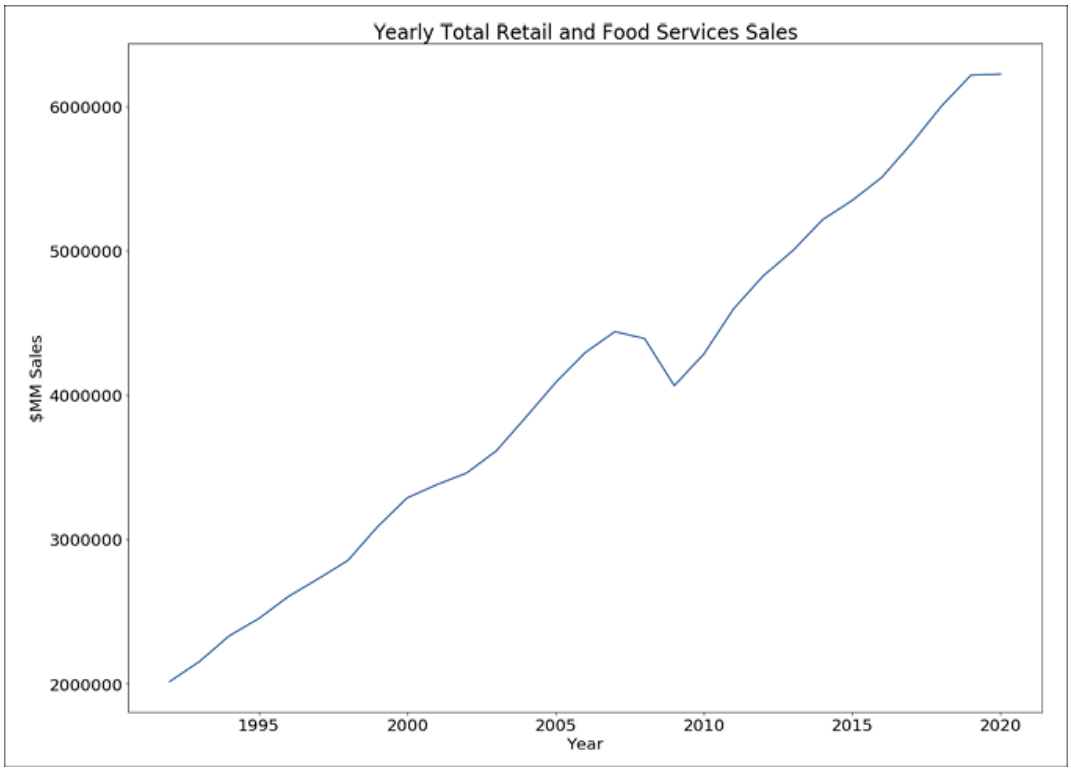


Figura 3.3: Tendência do total anual de vendas do varejo e do setor de serviços alimentícios.

Representar os dados de uma série temporal em gráfico com diferentes níveis de agregação, como semanal, mensal ou anual, é uma boa maneira de entender as tendências. Essa etapa pode ser usada simplesmente para criar um perfil para os dados, mas também pode ser a saída final, dependendo dos objetivos da análise. A seguir, examinaremos o uso de SQL para comparar os componentes de uma série temporal.

Comparando componentes

Geralmente os conjuntos de dados não contêm apenas uma série temporal; na verdade, eles contêm várias fatias ou componentes que perfazem um total ao longo do mesmo intervalo de tempo. Comparar essas fatias costuma revelar padrões interessantes. No conjunto de dados de vendas do varejo, há valores para o total de vendas, mas também há várias subcategorias. Compararemos a tendência das vendas anuais para algumas categorias que estão associadas a atividades de lazer: livrarias (book stores) , lojas de artigos esportivos (sporting goods stores) e lojas direcionadas a passatempos (hobby stores). Essa consulta adiciona `kind_of_business` à cláusula *SELECT* e, já que se trata de outro atributo em vez de uma agregação, ela também o adiciona à cláusula *GROUP BY*:

```
SELECT date_part('year',sales_month) as sales_year
,kind_of_business
,sum(sales) as sales
FROM retail_sales
WHERE kind_of_business in ('Book stores'
,'Sporting goods stores','Hobby, toy, and game stores')
GROUP BY 1,2
;
sales_year  kind_of_business             sales
----------  --------------------------  -----
1992.0      Book stores                 8327
1992.0      Hobby, toy, and game stores 11251
1992.0      Sporting goods stores       15583
...         ...                         ...
```

Os resultados estão representados no gráfico da Figura 3.4. As vendas dos varejistas de artigos esportivos começaram como as mais altas entre as três categorias, cresceram mais rapidamente durante o período e no final da série temporal eram significativamente mais altas. Elas começaram a cair em 2017, mas tiveram uma grande recuperação em 2020. As vendas nas lojas de passatempos, brinquedos e jogos permaneceram relativamente uniformes nesse período de tempo, com uma pequena queda no meio dos anos 2000 e outro pequeno declínio antes de uma recuperação em 2020. As vendas nas livrarias cresceram até o meio dos anos 2000 e vêm caindo

desde então. Todas essas categorias foram afetadas pelo crescimento dos varejistas online, mas o timing e a magnitude parecem diferir.

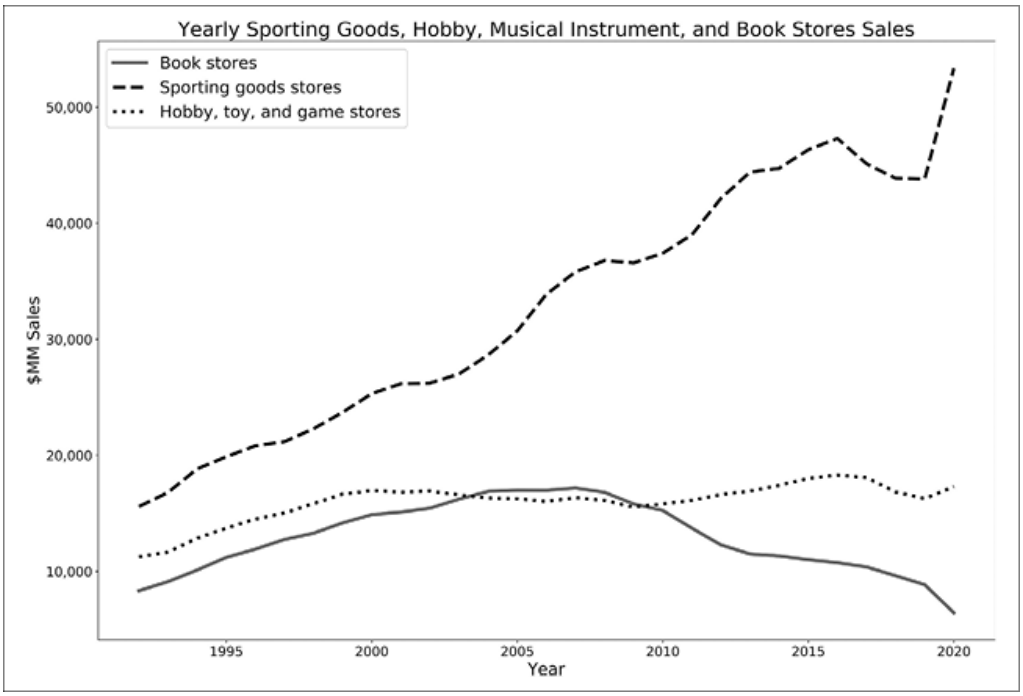


Figura 3.4: Tendência das vendas anuais do varejo nas lojas de artigos esportivos; nas lojas de passatempos, brinquedos e jogos: e nas livrarias.

Além de examinar tendências simples, poderíamos fazer comparações mais complexas entre partes da série temporal. Nos próximos exemplos, verificaremos as vendas nas lojas de roupas femininas e nas lojas de roupas masculinas. Repare que, como os nomes contêm apóstrofos, o caractere que indica o começo e o fim das strings, temos de escapá-los com um apóstrofo adicional. Isso permitirá que o banco de dados saiba que o apóstrofo faz parte da string em vez de indicar o fim dela. Embora pudéssemos considerar o acréscimo de uma etapa em um pipeline de carregamento de dados que removesse os apóstrofos adicionais dos nomes, deixei-os como demonstração dos tipos de ajustes no código que costumam ser necessários no mundo real. Primeiro, encontraremos a tendência nos dados de cada tipo de loja por mês:

```
SELECT sales_month
,kind_of_business
,sales
FROM retail_sales
```

```
WHERE kind_of_business in ('Men''s clothing stores'
,'Women''s clothing stores')
;
sales_month  kind_of_business         sales
-----------  ----------------------  -----
1992-01-01   Men's clothing stores    701
1992-01-01   Women's clothing stores  1873
1992-02-01   Women's clothing stores  1991
...          ...                     ...
```

Os resultados estão representados no gráfico da Figura 3.5. As vendas nos varejistas de roupas femininas são maiores do que nos varejistas de roupas masculinas. Os dois tipos de lojas exibem sazonalidade, um tópico que abordarei com detalhes em "Analisando com sazonalidade", na página 128. Ambos vivenciaram quedas significativas em 2020 devido ao fechamento das lojas e a uma redução nas compras por causa da pandemia de COVID-19.

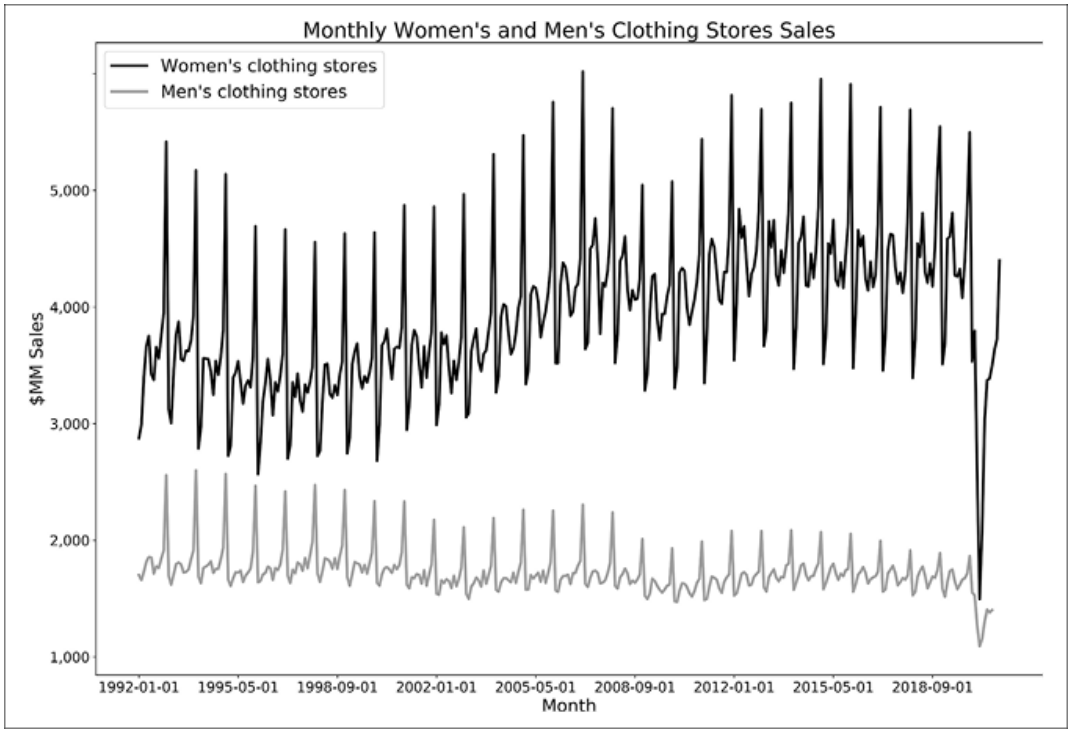


Figura 3.5: Tendência das vendas mensais nas lojas de roupas femininas e masculinas.

Os dados mensais têm padrões intrigantes, mas há ruído; logo, usaremos agregações anuais nos próximos exemplos. Vimos esse formato de

consulta anteriormente ao demonstrar as vendas e o valor total nas categorias de lazer:

```
SELECT date_part('year',sales_month) as sales_year
,kind_of_business
,sum(sales) as sales
FROM retail_sales
WHERE kind_of_business in ('Men''s clothing stores'
,'Women''s clothing stores')
GROUP BY 1,2
;
```

As vendas nas lojas de roupas femininas são uniformemente mais altas do que as das lojas de roupas masculinas? Na tendência anual mostrada na Figura 3.6, a lacuna entre as vendas de roupas masculinas e femininas não parece constante e, em vez disso, estava aumentando do início ao meio dos anos 2000. As vendas de roupas femininas em particular caíram durante a crise financeira global de 2008-2009 e as vendas das duas categorias caíram muito durante a pandemia em 2020.

Figura 3.6: Tendência das vendas anuais nas lojas de roupas femininas e masculinas.

No entanto, não precisamos depender da estimativa visual. Para aumentar

a precisão, podemos calcular a lacuna entre as duas categorias, a proporção, e a diferença percentual entre elas. Para fazê-lo, a primeira etapa é organizar os dados de modo que haja uma única linha para cada mês, com uma coluna para cada categoria. Pivotar os dados com funções de agregação combinadas com instruções CASE atinge esse objetivo:

```
SELECT date_part('year',sales_month) as sales_year
,sum(case when kind_of_business = 'Women''s clothing stores'
      then sales
      end) as womens_sales
,sum(case when kind_of_business = 'Men''s clothing stores'
      then sales
      end) as mens_sales
FROM retail_sales
WHERE kind_of_business in ('Men''s clothing stores'
,'Women''s clothing stores')
GROUP BY 1
;
sales_year  womens_sales  mens_sales
----------  ------------  ----------
1992.0      31815         10179
1993.0      32350         9962
1994.0      30585         10032
...         ...           ...
```

Com esse cálculo dos elementos constitutivos, podemos encontrar a diferença, a proporção e a diferença percentual entre as séries temporais do conjunto de dados. A diferença pode ser calculada pela subtração dos dois valores com o uso do operador matemático "–". Dependendo dos objetivos da análise, pode ser apropriado encontrar a diferença a partir das vendas de roupas masculinas ou a partir das vendas de roupas femininas. As duas situações são mostradas aqui e são equivalentes exceto pelo sinal:

```
SELECT sales_year
,womens_sales - mens_sales as womens_minus_mens
,mens_sales - womens_sales as mens_minus_womens
FROM
(
```

```
    SELECT date_part('year',sales_month) as sales_year
    ,sum(case when kind_of_business = 'Women''s clothing stores'
              then sales
              end) as womens_sales
    ,sum(case when kind_of_business = 'Men''s clothing stores'
              then sales
              end) as mens_sales
    FROM retail_sales
    WHERE kind_of_business in ('Men''s clothing stores'
     ,'Women''s clothing stores')
    and sales_month <= '2019-12-01'
    GROUP BY 1
) a
;
sales_year  womens_minus_mens  mens_minus_womens
----------  -----------------  -----------------
1992.0      21636              -21636
1993.0      22388              -22388
1994.0      20553              -20553
...         ...                ...
```

A subconsulta não é necessária do ponto de vista de execução da consulta, já que as agregações podem ser adicionadas ou subtraídas entre si. Geralmente uma subconsulta é mais legível, porém adiciona mais linhas ao código. Dependendo de quanto o restante de sua consulta SQL for longo ou complexo, pode ser preferível inserir o cálculo intermediário em uma subconsulta, ou executá-lo na consulta principal. Aqui está um exemplo sem a subconsulta, subtraindo as vendas de roupas masculinas das vendas de roupas femininas, com um filtro de cláusula *WHERE* adicionado para remover 2020, já que alguns meses têm valores nulos:[1]

```
SELECT date_part('year',sales_month) as sales_year
,sum(case when kind_of_business = 'Women''s clothing stores'
          then sales end)
-
sum(case when kind_of_business = 'Men''s clothing stores'
          then sales end)
as womens_minus_mens
```

```
FROM retail_sales
WHERE kind_of_business in ('Men''s clothing stores'
,'Women''s clothing stores')
and sales_month <= '2019-12-01'
GROUP BY 1
;
sales_year  womens_minus_mens
----------  -----------------
1992.0      21636
1993.0      22388
1994.0      20553
...         ...
```

A Figura 3.7 mostra que a lacuna diminuiu entre 1992 e 1997, teve um longo aumento até 2011 (com uma pequena queda em 2007) e ficou mais ou menos nivelada até 2019.

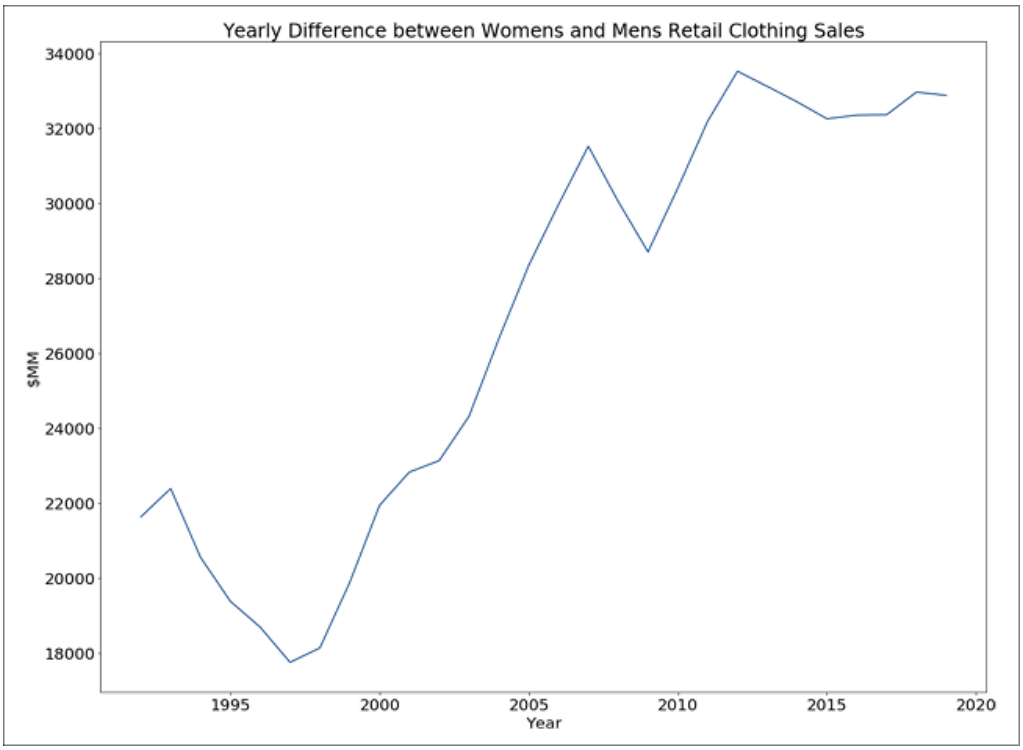


Figura 3.7: Diferença anual entre as vendas em lojas de roupas femininas e masculinas.

Continuaremos nossa investigação e examinaremos a proporção entre essas categorias. Usaremos as vendas de roupas masculinas como linha de base ou denominador, mas lembre-se de que poderíamos igualmente usar

as vendas das lojas de roupas femininas:

```
SELECT sales_year
,womens_sales / mens_sales as womens_times_of_mens
FROM
(
    SELECT date_part('year',sales_month) as sales_year
    ,sum(case when kind_of_business = 'Women''s clothing stores'
              then sales
              end) as womens_sales
    ,sum(case when kind_of_business = 'Men''s clothing stores'
              then sales
              end) as mens_sales
    FROM retail_sales
    WHERE kind_of_business in ('Men''s clothing stores'
     ,'Women''s clothing stores')
    and sales_month <= '2019-12-01'
    GROUP BY 1
) a
;
sales_year  womens_times_of_mens
----------  --------------------
1992.0      3.1255526083112290
1993.0      3.2473398915880345
1994.0      3.0487440191387560
...         ...
```

O SQL retorna muitos dígitos decimais ao executar divisões. Você deve considerar o arredondamento do resultado antes de apresentar a análise. Use o nível de precisão (número de casas decimais) que for mais ilustrativo.

A representação gráfica do resultado, mostrada na Figura 3.8, revela que a tendência é semelhante à da diferença, mas, embora haja uma queda na diferença em 2009, a proporção aumentou.

Figura 3.8: Proporção anual entre as vendas de roupas femininas e masculinas.

A seguir, calcularemos a diferença percentual entre as vendas em lojas de roupas femininas e masculinas:

```
SELECT sales_year
,(womens_sales / mens_sales - 1) * 100 as womens_pct_of_mens
FROM
(
    SELECT date_part('year',sales_month) as sales_year
    ,sum(case when kind_of_business = 'Women''s clothing stores'
              then sales
              end) as womens_sales
    ,sum(case when kind_of_business = 'Men''s clothing stores'
              then sales
              end) as mens_sales
    FROM retail_sales
    WHERE kind_of_business in ('Men''s clothing stores'
     ,'Women''s clothing stores')
    and sales_month <= '2019-12-01'
    GROUP BY 1
```

```
) a
;
sales_year  womens_pct_of_mens
----------  --------------------
1992.0      212.5552608311229000
1993.0      224.7339891588034500
1994.0      204.8744019138756000
...         ...
```

Embora as unidades dessa saída sejam diferentes das do exemplo anterior, a forma desse gráfico é a mesma da do gráfico da proporção. A escolha de qual usar vai depender de seu público-alvo e das normas de sua área. Todas essas afirmações são precisas: em 2009, as vendas nas lojas de roupas femininas foram 28,7 bilhões de dólares mais altas do que as vendas nas lojas masculinas; em 2009, as vendas nas lojas de roupas femininas foram 4,9 vezes mais altas do que as vendas nas lojas masculinas; em 2009, as vendas nas lojas de roupas femininas foram 390% mais altas do que as vendas nas lojas masculinas. Que versão selecionar vai depender da história que você deseja contar com a análise.

As transformações que vimos nesta seção nos permitem analisar séries temporais pela comparação de partes relacionadas. A próxima seção continuará o tema da comparação de séries temporais mostrando maneiras de analisar séries que representam partes de um todo.

Percentual em cálculos de totais

No trabalho com dados de séries temporais que tenham várias partes ou atributos que componham um todo, costuma ser útil analisar a contribuição de cada parte para o todo e se isso mudou com o tempo. A menos que os dados já contenham uma série temporal dos valores totais, precisaremos calcular o total geral para achar o percentual do total para cada linha. Isso pode ser feito com uma self-*JOIN*, ou uma função de janela, que como vimos no Capítulo 2 é um tipo especial de função SQL que pode referenciar qualquer linha existente dentro de uma partição especificada da tabela.

Primeiro mostrarei o método self-*JOIN*. Uma self-*JOIN* ocorre quando fazemos a junção de uma tabela com ela própria. Contanto que cada

instância da tabela na consulta receba um alias diferente, o banco de dados as tratará como tabelas distintas. Por exemplo, para encontrar o percentual das vendas de roupas masculinas e femininas combinadas que cada série representa, podemos fazer a junção (com *JOIN*) de `retail_sales`, cujo alias é `a`, com `retail_sales`, de alias `b`, no campo `sales_month`. Em seguida, selecionamos (com *SELECT*) o nome individual da série (`kind_of_business`) e os valores de `sales` a partir do alias `a`. Agora, a partir do alias `b`, somaremos (com `sum`) as vendas das duas categorias e chamaremos o resultado de `total_sales`. Observe que a *JOIN* entre as tabelas no campo `sales_month` cria uma *JOIN* Cartesiana parcial, que resulta em duas linhas do alias `b` para cada linha do alias `a`. No entanto, o agrupamento por `a.sales_month`, `a.kind_of_business` e `a.sales` e a agregação de `b.sales` retorna os resultados necessários. Na consulta externa, o percentual do total para cada linha é calculado pela divisão de `sales` por `total_sales`:

```
SELECT sales_month
,kind_of_business
,sales * 100 / total_sales as pct_total_sales
FROM
(
    SELECT a.sales_month, a.kind_of_business, a.sales
    ,sum(b.sales) as total_sales
    FROM retail_sales a
    JOIN retail_sales b on a.sales_month = b.sales_month
    and b.kind_of_business in ('Men''s clothing stores'
     ,'Women''s clothing stores')
    WHERE a.kind_of_business in ('Men''s clothing stores'
     ,'Women''s clothing stores')
    GROUP BY 1,2,3
) aa
;
sales_month  kind_of_business         pct_total_sales
-----------  ----------------------  ------------------
1992-01-01   Men's clothing stores    27.2338772338772339
1992-01-01   Women's clothing stores  72.7661227661227661
1992-02-01   Men's clothing stores    24.8395620989052473
```

```
...          ...                       ...
```

A subconsulta não é necessária aqui, já que o mesmo resultado poderia ser obtido sem ela, mas o seu uso torna o código um pouco mais fácil de seguir. Uma segunda maneira de calcular o percentual de vendas totais para cada categoria é usando a função de janela `sum` e empregando *PARTITION BY* com `sales_month`. Lembre-se de que a cláusula *PARTITION BY* indica a seção da tabela dentro da qual a função deve fazer o cálculo. A cláusula *ORDER BY* não é necessária nessa função de janela `sum`, porque a ordem do cálculo não é importante. Além disso, a consulta não precisa de uma cláusula *GROUP BY*, porque as funções de janela envolvem várias linhas, mas não reduzem o número de linhas no conjunto de resultados:

```
SELECT sales_month, kind_of_business, sales
,sum(sales) over (partition by sales_month) as total_sales
,sales * 100 / sum(sales) over (partition by sales_month) as
  pct_total
FROM retail_sales
WHERE kind_of_business in ('Men''s clothing stores'
,'Women''s clothing stores')
;
sales_month  kind_of_business         sales  total_sales  pct_total
-----------  ----------------------   -----  -----------  ---------
1992-01-01   Men's clothing stores    701    2574         27.233877
1992-01-01   Women's clothing stores  1873   2574         72.766122
1992-02-01   Women's clothing stores  1991   2649         75.160437
...          ...                      ...    ...          ...
```

A representação desses dados em gráfico, como na Figura 3.9, revela algumas tendências interessantes. Em primeiro lugar, a partir do fim dos anos 1990, as vendas das lojas de roupas femininas contribuíram com um percentual cada vez maior do total. Em segundo lugar, no início da série um padrão sazonal é evidente, no qual as vendas de roupas masculinas apresentam um pico no percentual de vendas totais em dezembro e janeiro. Na primeira década do século 21, aparecem dois picos sazonais, no verão e no inverno, mas no fim dos anos 2010, os padrões sazonais são reduzidos quase até aleatoriedade. Examinaremos a análise de sazonalidade com mais detalhes posteriormente neste capítulo.

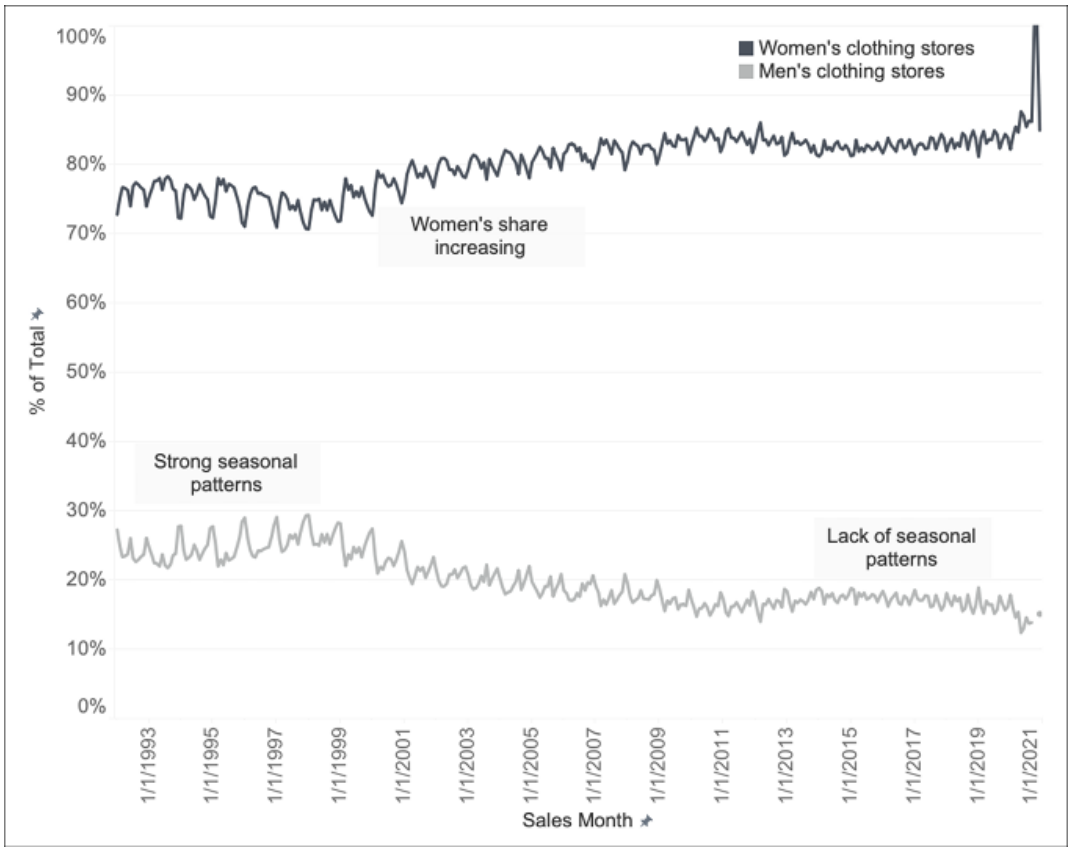


Figura 3.9: Vendas das lojas de roupas masculinas e femininas como um percentual do total mensal.

Outro percentual do total que podemos querer encontrar é o de vendas dentro de um período de tempo mais longo, como o percentual das vendas anuais que cada mês representa. Novamente, uma self-*JOIN* ou uma função de janela pode ajudar. Nesse exemplo, usaremos uma self-*JOIN* na subconsulta:

```
SELECT sales_month
,kind_of_business
,sales * 100 / yearly_sales as pct_yearly
FROM
(
    SELECT a.sales_month, a.kind_of_business, a.sales
    ,sum(b.sales) as yearly_sales
    FROM retail_sales a
    JOIN retail_sales b on
     date_part('year',a.sales_month) =
  date_part('year',b.sales_month)
```

```
     and a.kind_of_business = b.kind_of_business
     and b.kind_of_business in ('Men''s clothing stores'
      ,'Women''s clothing stores')
    WHERE a.kind_of_business in ('Men''s clothing stores'
     ,'Women''s clothing stores')
    GROUP BY 1,2,3
) aa
;
sales_month  kind_of_business       pct_yearly
-----------  --------------------  -----------------
1992-01-01   Men's clothing stores  6.8867275763827488
1992-02-01   Men's clothing stores  6.4642892229099126
1992-03-01   Men's clothing stores  7.1814520090382159
...          ...                   ...
```

Alternativamente, o método de função de janela pode ser usado:

```
SELECT sales_month, kind_of_business, sales
,sum(sales) over (partition by date_part('year',sales_month)
                               ,kind_of_business
                               ) as yearly_sales
,sales * 100 /
sum(sales) over (partition by date_part('year',sales_month)
                               ,kind_of_business
                               ) as pct_yearly
FROM retail_sales
WHERE kind_of_business in ('Men''s clothing stores'
,'Women''s clothing stores')
;
sales_month  kind_of_business       pct_yearly
-----------  --------------------  -----------------
1992-01-01   Men's clothing stores  6.8867275763827488
1992-02-01   Men's clothing stores  6.4642892229099126
1992-03-01   Men's clothing stores  7.1814520090382159
...          ...                   ...
```

Os resultados, com destaque para 2019, são mostrados na Figura 3.10. As duas séries temporais têm um emparelhamento bem próximo, mas as lojas masculinas tiveram em janeiro um percentual maior de vendas do

que as lojas femininas. As lojas masculinas tiveram uma queda nas vendas no verão em julho, enquanto a queda correspondente nas vendas das lojas femininas não ocorreu antes de setembro.

Agora que mostrei como usar SQL para cálculos de percentual do total e os tipos de análise que podem ser feitas, abordarei a indexação e o cálculo de alterações percentuais com o tempo.

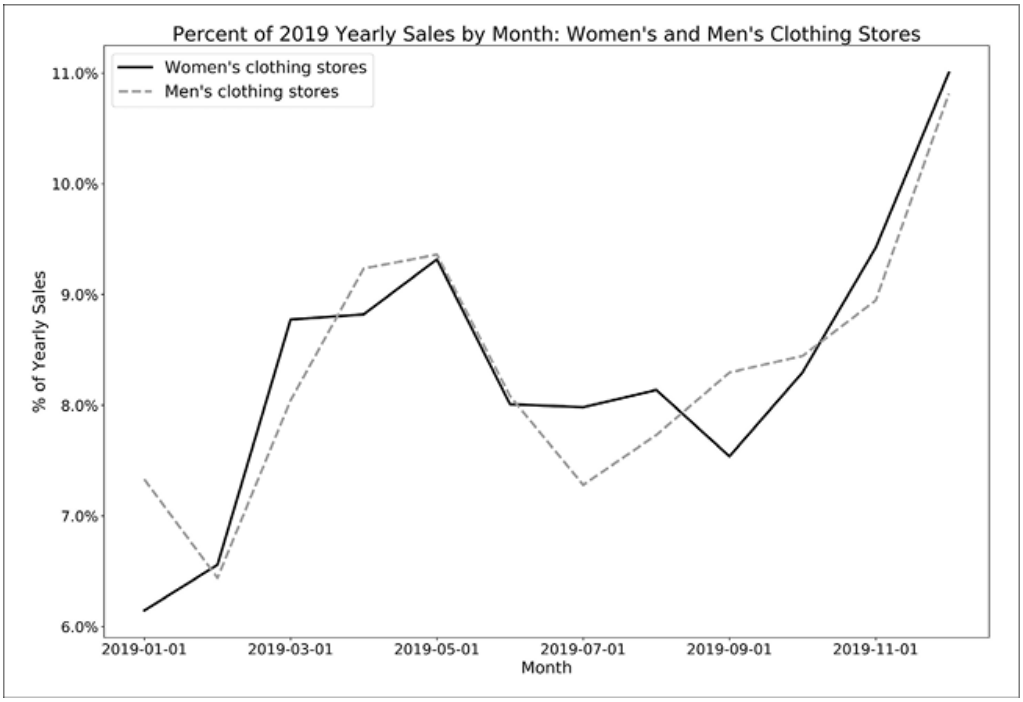


Figura 3.10: Percentual de vendas anuais de 2019, referente às vendas de roupas femininas e masculinas.

Indexação para a detecção de alterações percentuais com o passar do tempo

Geralmente os valores das séries temporais flutuam com o tempo. As vendas aumentam com a crescente popularidade e disponibilidade de um produto, enquanto o tempo de resposta das páginas web diminuem com os esforços dos engenheiros para otimizar o código. A indexação dos dados é uma maneira de entender as alterações ocorridas em uma série temporal em relação a um período base (ponto de partida). Os índices são amplamente usados na economia assim como nas definições empresariais. Um dos índices mais famosos é o IPC (Índice de Preços ao Consumidor), que rastreia a alteração nos preços dos itens que um consumidor típico compra e é usado no acompanhamento da inflação, na tomada de

decisões de aumentos salariais e em muitas outras aplicações. O IPC é uma medida estatística complexa que usa vários pesos e entradas de dados, mas a premissa básica é simples. Escolha um período base e calcule a alteração percentual no valor a partir desse período base para cada período subsequente.

A indexação de dados de séries temporais com SQL pode ser feita com uma combinação de agregações e funções de janela, ou self-*JOIN*s. Como exemplo, indexaremos as vendas das lojas de roupas femininas a partir do primeiro ano da série, 1992. A primeira etapa é agregar as vendas por `sales_year` (ano de vendas) em uma subconsulta, como fizemos anteriormente. Na consulta externa, a função de janela `first_value` encontrará o valor associado à primeira linha na cláusula *PARTITION BY*, de acordo com a classificação feita na cláusula *ORDER BY*. Nesse exemplo, podemos omitir a cláusula *PARTITION BY*, porque queremos retornar o valor das vendas da primeira linha do conjunto de dados inteiro retornado pela subconsulta:

```
SELECT sales_year, sales
,first_value(sales) over (order by sales_year) as index_sales
FROM
(
    SELECT date_part('year',sales_month) as sales_year
    ,sum(sales) as sales
    FROM retail_sales
    WHERE kind_of_business = 'Women''s clothing stores'
    GROUP BY 1
) a
;
sales_year  sales  index_sales
----------  -----  -----------
1992.0      31815  31815
1993.0      32350  31815
1994.0      30585  31815
...         ...    ...
```

Com essa amostra de dados, podemos verificar visualmente se o valor do índice está definido corretamente com o valor de 1992. A seguir, encontraremos a alteração percentual a partir desse ano base para cada

linha:

```
SELECT sales_year, sales
,(sales / first_value(sales) over (order by sales_year) - 1) * 100
as pct_from_index
FROM
(
    SELECT date_part('year',sales_month) as sales_year
    ,sum(sales) as sales
    FROM retail_sales
    WHERE kind_of_business = 'Women''s clothing stores'
    GROUP BY 1
) a
;
sales_year  sales  pct_from_index
----------  -----  --------------
1992.0      31815  0
1993.0      32350  1.681596731101
1994.0      30585  -3.86610089580
...         ...    ...
```

A alteração percentual pode ser positiva ou negativa, e veremos que ela realmente ocorre nessa série temporal. A função de janela `last_value` poderia ser substituída por `first_value` nessa consulta. No entanto, é bem menos comum indexar a partir do último valor de uma série, já que quase sempre as perguntas da análise estão relacionadas à alteração a partir de um ponto inicial em vez de olhar para trás a partir de um ponto final arbitrário; mesmo assim, existe essa opção. Além disso, a ordem de classificação pode ser usada para fazermos a indexação a partir do primeiro ou do último valor com a alternância entre *ASC* e *DESC*:

```
first_value(sales) over (order by sales_year desc)
```

As funções de janela proporcionam muita flexibilidade. A indexação pode ser obtida sem elas por meio de uma série de self-*JOIN*s, embora mais linhas de código sejam necessárias:

```
SELECT sales_year, sales
,(sales / index_sales - 1) * 100 as pct_from_index
FROM
(
```

```
    SELECT date_part('year',aa.sales_month) as sales_year
    ,bb.index_sales
    ,sum(aa.sales) as sales
    FROM retail_sales aa
    JOIN
    (
        SELECT first_year, sum(a.sales) as index_sales
        FROM retail_sales a
        JOIN
        (
            SELECT min(date_part('year',sales_month)) as first_year
            FROM retail_sales
            WHERE kind_of_business = 'Women''s clothing stores'
        ) b on date_part('year',a.sales_month) = b.first_year
        WHERE a.kind_of_business = 'Women''s clothing stores'
        GROUP BY 1
    ) bb on 1 = 1
    WHERE aa.kind_of_business = 'Women''s clothing stores'
    GROUP BY 1,2
) aaa
;
sales_year  sales  pct_from_index
----------  -----  --------------
1992.0      31815  0
1993.0      32350  1.681596731101
1994.0      30585  -3.86610089580
...         ...    ...
```

Observe a incomum cláusula *JOIN* `on 1 = 1` entre o alias `aa` e a subconsulta de `bb`. Já que queremos que o valor de `index_sales` preencha cada linha do conjunto de resultados, não podemos fazer a junção com o ano ou outro valor, porque restringiria os resultados. No entanto, o banco de dados retornará um erro se nenhuma cláusula *JOIN* for especificada. Podemos enganar o banco de dados usando qualquer expressão que seja avaliada como TRUE para criar a *JOIN* Cartesiana desejada. Qualquer outra instrução verdadeira (TRUE), como `on 2 = 2` ou `on 'apples' = 'apples'`, também poderia ser usada.

Cuidado com os zeros no denominador de operações de divisão como a operação `sales / index_sales` do último exemplo. Os bancos de dados retornam um erro quando encontram uma divisão por zero, o que pode ser frustrante. Mesmo quando você achar improvável que haja um zero no campo do denominador, é boa prática evitar que isso ocorra solicitando ao banco de dados que retorne um valor padrão alternativo quando encontrar um zero. Isso pode ser feito com uma instrução CASE. Os exemplos desta seção não têm zeros no denominador, logo, omitirei esse código adicional para melhorar a legibilidade.

Para encerrar esta seção, examinaremos um gráfico da série temporal indexada das lojas de roupas masculinas e femininas, mostrado na Figura 3.11. O código SQL é o descrito a seguir:

```
SELECT sales_year, kind_of_business, sales
,(sales / first_value(sales) over (partition by kind_of_business
  order by sales_year)
- 1) * 100 as pct_from_index
FROM
(
    SELECT date_part('year',sales_month) as sales_year
    ,kind_of_business
    ,sum(sales) as sales
    FROM retail_sales
    WHERE kind_of_business in ('Men''s clothing stores'
     ,'Women''s clothing stores')
    and sales_month <= '2019-12-31'
    GROUP BY 1,2
) a
;
```

Figura 3.11: Vendas das lojas de roupas masculinas e femininas, indexadas a partir das vendas de 1992.

Fica evidente nesse gráfico que 1992 foi o ápice das vendas nas lojas de roupas masculinas. Após 1992, as vendas caíram, depois voltaram brevemente para o mesmo nível em 1998 e têm caído desde então. Isso é surpreendente porque, mesmo que o conjunto de dados não tenha sido ajustado de acordo com a inflação, a tendência dos preços é subir com o tempo. Inicialmente as vendas nas lojas de roupas femininas ficaram em um nível menor do que o de 1992, mas voltaram ao nível de 1992 em 2003. Depois disso, elas aumentaram, a não ser pela queda durante a crise financeira que diminuiu as vendas em 2009 e 2010. Uma explicação para essas tendências seria que os homens simplesmente reduziram seus gastos com roupas com o tempo, talvez por estarem acompanhando menos a moda do que as mulheres. Talvez as roupas masculinas tenham ficado menos caras à medida que as cadeias de suprimento globais diminuíram seus custos. Outra explicação poderia ser que os homens deixaram de comprar roupas em varejistas categorizados como "lojas de roupas masculinas" e passaram a comprar em outros tipos de varejistas, como nas lojas de artigos esportivos ou nos varejistas online.

A indexação de dados de séries temporais é uma técnica de análise

poderosa, que nos permite descobrir vários insights nos dados. O SQL é apropriado para essa tarefa, e mostrei como construir séries temporais indexadas com e sem funções de janela. A seguir, mostrarei como analisar dados usando janelas de tempo contínuas para encontrar padrões em séries temporais com ruído.

Janelas de tempo contínuas

Os dados de séries temporais geralmente têm ruído, um desafio para um de nossos objetivos primários de busca por padrões. Vimos como a agregação de dados, como de mensais para anuais, pode uniformizar os resultados e torná-los mais fáceis de interpretar. Outra técnica para a uniformização dos dados são as *janelas de tempo contínuas*, também conhecidas como cálculos móveis, que levam em consideração vários períodos. As médias móveis talvez sejam as mais comuns, mas com o poder do SQL, qualquer função de agregação está disponível para análise. As janelas de tempo contínuas são usadas em várias áreas de análise, incluindo os mercados de ações, as tendências macroeconômicas e a medição de audiência. Alguns cálculos são tão comumente usados que têm os próprios acrônimos: LTM (last twelve months, últimos doze meses), TTM (trailing twelve monts, primeiros doze meses) e YTD (year-to-date, do início do ano até hoje).

A Figura 3.12 mostra um exemplo de janela de tempo contínua e de um cálculo cumulativo, em relação ao mês de outubro da série temporal.

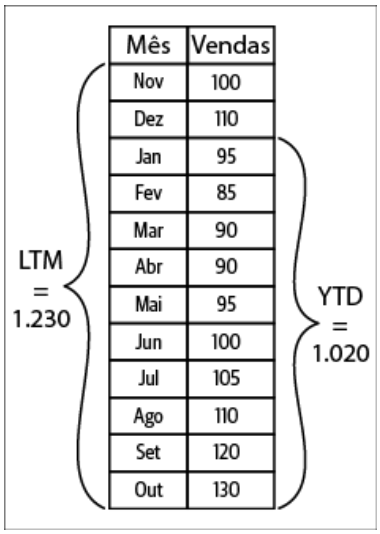


Figura 3.12: Exemplos das somas LTM e YTD das vendas contínuas.

Há várias partes importantes em qualquer cálculo de série temporal contínua. Primeiro vem o tamanho da janela, que é o número de períodos a serem incluídos no cálculo. Janelas maiores com mais períodos de tempo têm um efeito maior de uniformização, mas com o risco de perder a capacidade de detecção de importantes alterações de curto prazo nos dados. Janelas mais curtas com menos períodos de tempo proporcionam menos uniformização e, portanto, são mais sensíveis a alterações de curto prazo, mas com o risco de menor redução do ruído.

A segunda parte dos cálculos de séries temporais é a função de agregação usada. Como mencionado anteriormente, talvez as médias móveis sejam as mais comuns. Assim como as somas móveis, as contagens e a obtenção de valores mínimos e máximos também podem ser calculadas com SQL. As contagens móveis são úteis em métricas de população de usuários (consulte a caixa de texto a seguir). Os cálculos móveis de valores mínimo e máximo podem nos ajudar a entender os extremos dos dados, o que é útil para o planejamento das análises.

A terceira parte dos cálculos de séries temporais é selecionar o particionamento, ou agrupamento, dos dados que serão incluídos na janela. A análise pode precisar de uma redefinição anual da janela. Ou de uma série móvel diferente para cada grupo de componentes ou de usuários. O Capítulo 4 fornecerá mais detalhes sobre a análise de coorte de grupos de usuários, onde consideraremos como a retenção e valores cumulativos, como os gastos, diferem entre as populações com o passar do tempo. O particionamento será controlado por meio do agrupamento e da instrução *PARTITION BY* das funções de janela.

Tendo tomado conhecimento dessas três partes, examinaremos o código SQL e os cálculos de períodos de tempo móveis, continuando com o conjunto de dados de vendas do varejo dos EUA como exemplo.

Medindo "usuários ativos": DAU, WAU e MAU

Muitas aplicações SaaS de consumidores e algumas B2B usam cálculos de usuários ativos como o DAU (daily active users, usuários ativos diários), o WAU (weekly active users, usuários ativos semanais) e o MAU (monthly active users, usuários ativos mensais) para estimar o tamanho de seu público-alvo. Já que são janelas contínuas, são cálculos que podem ser feitos diariamente. Perguntam-me

com frequência qual é a métrica certa ou melhor e minha resposta é sempre "depende".

O DAU ajuda as empresas no planejamento da capacidade, como na estimativa do volume de carga esperada nos servidores. No entanto, dependendo do serviço, dados ainda mais detalhados podem ser necessários, como o pico por hora ou até mesmo informações de usuários simultâneos de minuto a minuto.

Normalmente o MAU é usado para estimar os tamanhos relativos de aplicações ou serviços. Ele é útil para medir populações de usuários razoavelmente estáveis ou crescentes que tenham padrões de uso regulares que não sejam necessariamente diários, como o uso mais alto no fim de semana para produtos de entretenimento ou nos dias úteis para produtos relacionados a trabalho ou estudos. O MAU não é adequado para a detecção de alterações na rotatividade subjacente de usuários que pararam de usar uma aplicação. Já que o cálculo do MAU para um usuário envolve 30 dias, que é a janela mais comum, o usuário pode deixar de usar o produto por 29 dias antes de o MAU detectar uma queda no uso.

O WAU, calculado ao longo de 7 dias, pode ser um intermediário satisfatório entre o DAU e o MAU. Ele é mais sensível a flutuações de curto prazo, alertando as equipes sobre alterações na rotatividade com mais rapidez do que o MAU e abrandando ao mesmo tempo as flutuações nos dias úteis que são rastreadas pelo DAU. Uma desvantagem do WAU é que ele também é sensível a flutuações de curto prazo causadas por eventos como os feriados.

Calculando janelas de tempo contínuas

Agora que sabemos o que são janelas de tempo contínuas, como elas podem ser úteis e quais são seus principais componentes, iremos calculá-las usando o conjunto de dados de vendas do varejo dos EUA. Começaremos com o caso mais simples, no qual o conjunto de dados contém um registro para cada período que deve fazer parte da janela, e na seção seguinte examinaremos o que fazer quando não for isso que ocorrer.

Há dois métodos principais para o cálculo de uma janela de tempo contínua: uma self-*JOIN*, que pode ser usada em qualquer banco de dados, e uma função de janela, que como vimos não está disponível em alguns bancos de dados. Nos dois casos, precisamos do mesmo resultado: uma data e vários pontos de dados correspondentes ao tamanho da janela à qual aplicaremos uma média ou outra função de agregação.

Neste exemplo, usaremos uma janela de 12 meses para obter as vendas

contínuas anuais, já que os dados estão em um nível mensal de granularidade. Em seguida, aplicaremos uma média para obter a média móvel de 12 meses das vendas do varejo. Primeiro, usaremos a intuição para desenvolver o que entrará no cálculo. Nesta consulta, o alias `a` da tabela será nossa tabela "âncora", aquela a partir da qual coletaremos as datas. Para começar, examinaremos um único mês, dezembro de 2019. Do alias `b`, a consulta coletará os 12 meses de vendas individuais que entrarão na média móvel. Isso será feito com a cláusula *JOIN* `b.sales_month between a.sales_month - interval '11 months' and a.sales_month`, que cria uma *JOIN* Cartesiana intencional:

```
SELECT a.sales_month
,a.sales
,b.sales_month as rolling_sales_month
,b.sales as rolling_sales
FROM retail_sales a
JOIN retail_sales b on a.kind_of_business = b.kind_of_business
and b.sales_month between a.sales_month - interval '11 months'
and a.sales_month
and b.kind_of_business = 'Women''s clothing stores'
WHERE a.kind_of_business = 'Women''s clothing stores'
and a.sales_month = '2019-12-01'
;
sales_month  sales  rolling_sales_month  rolling_sales
-----------  -----  -------------------  -------------
2019-12-01   4496   2019-01-01           2511
2019-12-01   4496   2019-02-01           2680
2019-12-01   4496   2019-03-01           3585
2019-12-01   4496   2019-04-01           3604
2019-12-01   4496   2019-05-01           3807
2019-12-01   4496   2019-06-01           3272
2019-12-01   4496   2019-07-01           3261
2019-12-01   4496   2019-08-01           3325
2019-12-01   4496   2019-09-01           3080
2019-12-01   4496   2019-10-01           3390
2019-12-01   4496   2019-11-01           3850
2019-12-01   4496   2019-12-01           4496
```

Observe que os valores de `sales_month` e `sales` provenientes do alias `a` são

repetidos para cada linha dos 12 meses da janela.


Lembre-se de que as datas de uma cláusula *BETWEEN* são inclusivas (as duas serão retornadas no conjunto de resultados). É um erro comum o uso de 12 em vez de 11 como na consulta anterior. Quando em dúvida, verifique os resultados de consulta intermediários como fiz aqui para se certificar se o número desejado de períodos está entrando no cálculo da janela.

A próxima etapa é aplicar a agregação – nesse caso, `avg`, já que queremos uma média móvel. A contagem de registros retornada de alias `b` está sendo incluída para confirmarmos se cada linha está envolvendo 12 pontos de dados na média, uma verificação útil de qualidade de dados. O alias `a` também tem um filtro em `sales_month`. Como esse conjunto de dados começa em 1992, os meses desse ano, exceto dezembro, têm menos de 12 registros históricos:

```
SELECT a.sales_month
,a.sales
,avg(b.sales) as moving_avg
,count(b.sales) as records_count
FROM retail_sales a
JOIN retail_sales b on a.kind_of_business = b.kind_of_business
and b.sales_month between a.sales_month - interval '11 months'
and a.sales_month
and b.kind_of_business = 'Women''s clothing stores'
WHERE a.kind_of_business = 'Women''s clothing stores'
and a.sales_month >= '1993-01-01'
GROUP BY 1,2
;
sales_month  sales  moving_avg  records_count
-----------  -----  ----------  -------------
1993-01-01   2123   2672.08     12
1993-02-01   2005   2673.25     12
1993-03-01   2442   2676.50     12
...          ...    ...         ...
```

Os resultados estão representados em gráfico na Figura 3.13. Embora a tendência mensal apresente ruído, a tendência abrandada pela média móvel facilita a detecção de alterações como o aumento de 2003 a 2007 e

a queda subsequente até 2011. Observe que a queda acentuada no início de 2020 puxa a média móvel para baixo mesmo após as vendas começarem a se recuperar posteriormente nesse ano.

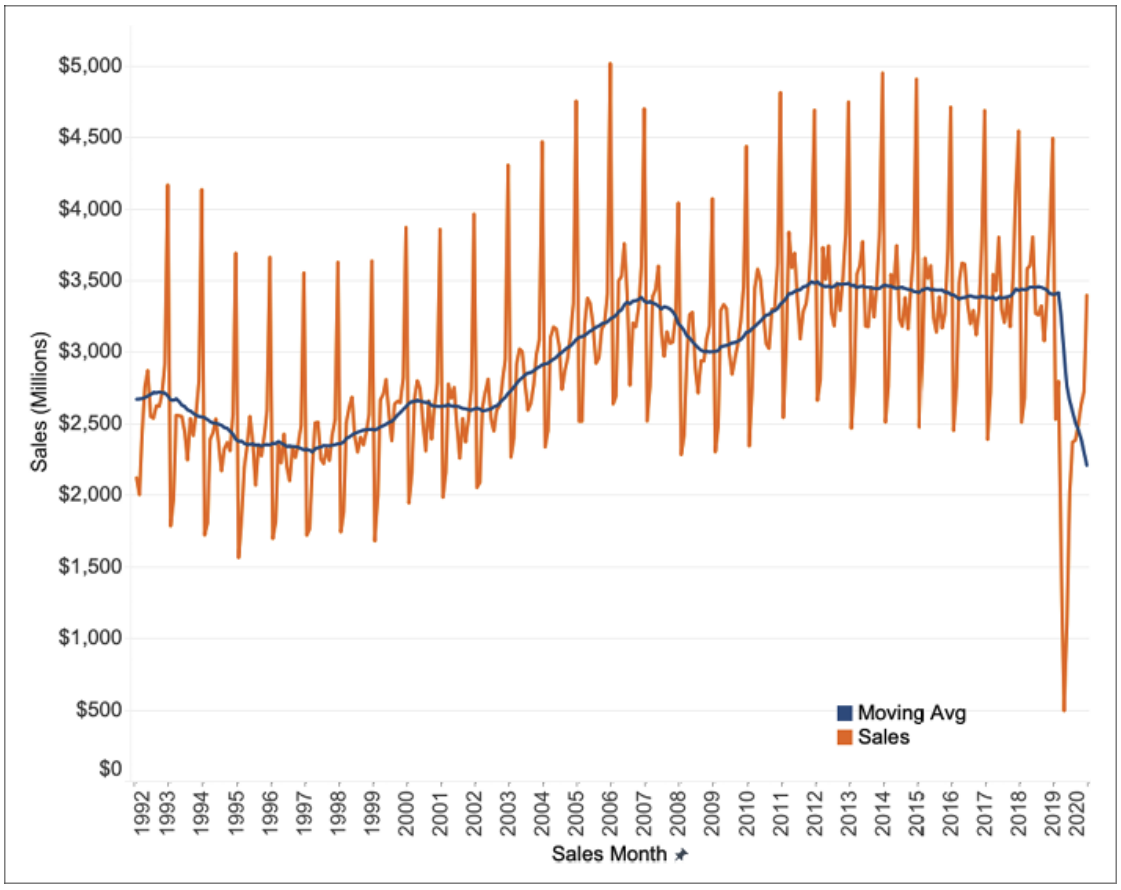


Figura 3.13: Vendas mensais e vendas na média móvel de 12 meses das lojas de roupas femininas.

A inclusão do filtro `kind_of_business = 'Women''s clothing stores'` em cada alias não é obrigatória. Já que a consulta usa uma *INNER JOIN*, fazer a filtragem em uma tabela fará com que ela seja executada automaticamente na outra tabela. No entanto, geralmente executar a filtragem nas duas tabelas faz as consultas ocorrerem com mais rapidez, principalmente quando as tabelas são grandes.

As funções de janela são outra maneira de calcular janelas de tempo contínuas. Para criar uma janela contínua, precisamos usar outra parte, opcional, de um cálculo de janela: a *cláusula de frame*. A cláusula de frame permite especificar que registros serão incluídos na janela. Por padrão, todos os registros da partição são incluídos, e em muitos casos isso

funciona bem. No entanto, controlar os registros incluídos em um nível de maior granularidade é útil para casos como os de cálculos de janela móvel. A sintaxe é simples, mas mesmo assim pode parecer confusa quando vista pela primeira vez. A cláusula de frame pode ser especificada como:

```
{ RANGE | ROWS | GROUPS } BETWEEN frame_start AND frame_end
```

Dentro dos parênteses há três opções para o tipo de frame: range (intervalo), rows (linhas) e groups (grupos). Essas são as maneiras de especificar que registros serão incluídos no resultado, em relação à linha atual. Os registros são sempre selecionados na partição atual e seguem a cláusula *ORDER BY* especificada. A classificação padrão é a ascendente (*ASC*), mas ela pode ser alterada para descendente (*DESC*). *Rows* é a opção mais simples e permite especificar o número exato de linhas que deve ser retornado. *Range* incluirá os registros que estejam dentro de algum limite de valores em relação à linha atual. *Groups* pode ser usada quando houver vários registros com o mesmo valor para *ORDER BY*, como quando um conjunto de dados inclui múltiplas linhas por vendas mensais, uma para cada cliente.

frame_start e *frame_end* podem ter qualquer um dos valores a seguir:

```
UNBOUNDED PRECEDING
offset PRECEDING
CURRENT ROW
offset FOLLOWING
UNBOUNDED FOLLOWING
```

Preceding significa incluir linhas anteriores à linha atual, de acordo com a classificação de *ORDER BY*. *Current row* é exatamente isso, a linha atual, e *following* significa incluir linhas que ocorram após a linha atual de acordo com a classificação de *ORDER BY*. A palavra-chave *UNBOUNDED* significa incluir todos os registros da partição anteriores ou posteriores à linha atual. O *offset* (deslocamento) é o número de registros, com frequência apenas uma constante inteira, embora um campo ou uma expressão que retorne um inteiro também possa ser usado. As cláusulas de frame também têm uma opção facultativa *frame_exclusion*, que não faz parte dessa discussão. A Figura 3.14 mostra um exemplo das linhas que cada uma das opções de frame de janela selecionará.

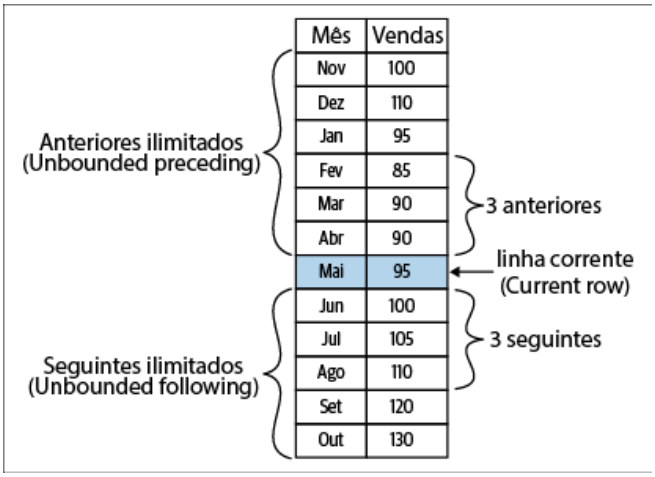


Figura 3.14: Cláusulas de frame de janela e as linhas que elas incluirão.

Do particionamento à classificação e aos frames de janelas, as funções de janela têm várias opções que controlam os cálculos, o que as torna incrivelmente poderosas e adequadas para a execução de cálculos complexos com uma sintaxe relativamente simples. Voltando ao nosso exemplo das vendas do varejo, a média móvel que calculamos usando uma self-*JOIN* pode ser calculada com funções de janela com menos linhas de código:

```
SELECT sales_month
,avg(sales) over (order by sales_month
                  rows between 11 preceding and current row
                  ) as moving_avg
,count(sales) over (order by sales_month
                  rows between 11 preceding and current row
                  ) as records_count
FROM retail_sales
WHERE kind_of_business = 'Women''s clothing stores'
;
sales_month  moving_avg  records_count
-----------  ----------  -------------
1992-01-01   1873.00     1
1992-02-01   1932.00     2
1992-03-01   2089.00     3
...          ...         ...
1993-01-01   2672.08     12
1993-02-01   2673.25     12
```

```
1993-03-01   2676.50     12
...          ...          ...
```

Nessa consulta, a janela ordena as vendas por mês (ascendente) para assegurar que os registros estejam em ordem cronológica. A cláusula de frame é `rows between 11 preceding and current row`, já que sei que tenho um registro para cada mês e quero os 11 meses anteriores e os meses da linha atual incluídos nos cálculos de média e contagem. A consulta retorna todos os meses, incluindo os que não têm 11 meses anteriores, e, se quisermos filtrar esses meses e excluí-los, podemos fazer isso inserindo essa consulta em uma subconsulta e fazendo a filtragem por mês ou número de registros na consulta externa.

Embora calcular médias móveis de períodos de tempo anteriores seja comum em muitos contextos empresariais, as funções de janela do SQL são suficientemente flexíveis para também incluir períodos de tempo futuros. Elas também podem ser usadas em qualquer cenário em que os dados tenham alguma ordem, e não apenas na análise de séries temporais.

O cálculo de médias móveis ou de outras agregações móveis pode ser executado com self-*JOIN*s ou funções de janela quando existirem registros no conjunto de dados para cada período de tempo da janela. Pode haver diferenças de desempenho entre os dois métodos, dependendo do tipo de banco de dados e do tamanho do conjunto de dados. Infelizmente, é difícil prever qual terá um desempenho adequado ou fazer recomendações sobre qual usar. É útil testar os dois métodos e prestar atenção em quanto tempo eles demoram para retornar os resultados da consulta; em seguida, use o que executar com mais rapidez sua opção padrão. Agora que vimos como calcular janelas de tempo contínuas, mostrarei como calcular janelas contínuas com conjuntos de dados esparsos.

Janelas de tempo contínuas com dados esparsos

Os conjuntos de dados do mundo real podem não conter um registro para cada período de tempo que pertence à janela. A medição de interesse pode ser sazonal ou intermitente por natureza. Por exemplo, clientes podem voltar a fazer compras em um site em intervalos regulares, ou um produto específico pode existir ou não em estoque. Isso resulta em dados

esparsos.

Na última seção, mostrei como calcular uma janela contínua com uma self-*JOIN* e um intervalo de datas da cláusula *JOIN*. Você deve estar pensando que esse cálculo incluirá qualquer registro que esteja dentro da janela de tempo de 12 meses, estejam eles ou não no conjunto de dados, e está correto. O problema dessa abordagem surge quando não há registro para o mês (ou para o dia ou o ano). Por exemplo, suponhamos que quiséssemos calcular as vendas contínuas de 12 meses para cada modelo de sapato que minha loja tem em estoque desde dezembro de 2019. Contudo, alguns dos sapatos não têm estoque antes de dezembro e, portanto, não têm registros de vendas nesse mês. O uso de uma self-*JOIN* ou de uma função de janela retornará um conjunto de dados de vendas contínuas para todos os sapatos que foram vendidos em dezembro, mas os dados não incluirão os sapatos para os quais não havia estoque. Felizmente, temos uma maneira de resolver esse problema: usando uma dimensão de data.

A *dimensão de data*, uma tabela estática que contém uma linha para cada data do calendário, foi introduzida no Capítulo 2. Com essa tabela podemos assegurar que uma consulta retorne um resultado para cada data de interesse, independentemente de haver ou não um ponto de dados para essa data no conjunto de dados subjacente. Já que os dados de `retail_sales` não incluem linhas para todos os meses, simulei um conjunto de dados esparsos adicionando uma subconsulta para filtrar a tabela e incluir apenas vendas mensais de janeiro e julho (1 e 7). Examinaremos os resultados quando unidos (com *JOIN*) a `date_dim`, mas antes da agregação, para ter um insight sobre os dados antes da aplicação de cálculos:

```
SELECT a.date, b.sales_month, b.sales
FROM date_dim a
JOIN
(
    SELECT sales_month, sales
    FROM retail_sales
    WHERE kind_of_business = 'Women''s clothing stores'
     and date_part('month',sales_month) in (1,7)
```

```
) b on b.sales_month between a.date - interval '11 months' and
  a.date
WHERE a.date = a.first_day_of_month
and a.date between '1993-01-01' and '2020-12-01'
;
date        sales_month  sales
----------  -----------  -----
1993-01-01  1992-07-01   2373
1993-01-01  1993-01-01   2123
1993-02-01  1992-07-01   2373
1993-02-01  1993-01-01   2123
1993-03-01  1992-07-01   2373
...         ...          ...
```

Observe que a consulta retorna resultados de datas de fevereiro e março além de janeiro, ainda que não haja vendas para esses meses nos resultados da subconsulta. Isso é possível porque a dimensão de data contém registros para todos os meses. O filtro `a.date = a.first_day_of_month` restringe o conjunto de resultados a um valor por mês, em vez das 28 a 31 linhas por mês que resultariam da junção com cada data. A construção dessa consulta seria então muito semelhante à consulta com self-*JOIN* da última seção, com a cláusula *JOIN* `on b.sales_month between a.date - interval '11 months' and a.date` tendo o mesmo formato da cláusula *JOIN* de self-*JOIN*. Agora que sabemos o que a consulta retornará, podemos prosseguir e aplicar a agregação `avg` para obter a média móvel:

```
SELECT a.date
,avg(b.sales) as moving_avg
,count(b.sales) as records
FROM date_dim a
JOIN
(
    SELECT sales_month, sales
    FROM retail_sales
    WHERE kind_of_business = 'Women''s clothing stores'
     and date_part('month',sales_month) in (1,7)
) b on b.sales_month between a.date - interval '11 months' and
```

```
  a.date
WHERE a.date = a.first_day_of_month
and a.date between '1993-01-01' and '2020-12-01'
GROUP BY 1
;
date        moving_avg  records
----------  ----------  -------
1993-01-01  2248.00     2
1993-02-01  2248.00     2
1993-03-01  2248.00     2
...         ...         ...
```

Como vimos, o conjunto de resultados inclui uma linha para cada mês; no entanto, a média móvel permanece constante até um novo ponto de dados (nesse caso, janeiro ou julho) ser adicionado. Cada média móvel é composta de dois pontos de dados subjacentes. Em um caso de uso real, o número de pontos de dados subjacentes deve variar. Para retornar o valor do mês atual ao usar uma dimensão de data, podemos utilizar uma agregação com uma instrução CASE – por exemplo:

```
,max(case when a.date = b.sales_month then b.sales end)
as sales_in_month
```

As condições da instrução CASE podem ser alteradas para retornar qualquer um dos registros subjacentes que a análise demanda por meio do uso da igualdade, da diferença ou dos deslocamentos com a matemática de datas. Se não houver uma dimensão de data disponível em seu banco de dados, outra técnica pode ser usada para simular uma. Em uma subconsulta, use *SELECT* para selecionar as diferentes datas necessárias e faça a junção delas com sua tabela da mesma forma que fizemos nos exemplos anteriores:

```
SELECT a.sales_month, avg(b.sales) as moving_avg
FROM
(
    SELECT distinct sales_month
    FROM retail_sales
    WHERE sales_month between '1993-01-01' and '2020-12-01'
) a
JOIN retail_sales b on b.sales_month between
```

```
a.sales_month - interval '11 months' and a.sales_month
and b.kind_of_business = 'Women''s clothing stores'
GROUP BY 1
;
sales_month  moving_avg
-----------  ----------
1993-01-01   2672.08
1993-02-01   2673.25
1993-03-01   2676.50
...          ...
```

Neste exemplo, usei a mesma tabela subjacente porque sei que ela contém todos os meses. No entanto, na prática qualquer tabela de banco de dados que contenha as datas necessárias pode ser usada, estando ou não relacionada com a tabela a partir da qual você deseja calcular a agregação contínua.

O cálculo de janelas de tempo contínuas com dados esparsos ou ausentes pode ser feito em SQL com a aplicação controlada de *JOIN*s Cartesianas. A seguir, examinaremos como calcular os valores cumulativos que são usados com frequência nas análises.

Calculando valores cumulativos

Normalmente os cálculos de janela contínua, como as médias móveis, usam janelas de tamanho fixo, como as de 12 meses que vimos na última seção. Outro tipo de cálculo que costuma ser usado é o de *valor cumulativo*, como o YTD, o QTD (quarter-to-date, do começo do trimestre até hoje) e o MTD (do começo do mês até hoje). Em vez de uma janela de tamanho fixo, eles usam um ponto de partida comum, com o tamanho da janela crescendo a cada linha.

A maneira mais simples de calcular valores cumulativos é com uma função de janela. Neste exemplo, `sum` é usada para encontrar o YTD das vendas totais para cada mês. Outra análise poderia demandar o YTD da média mensal ou um YTD do valor máximo mensal, que poderiam ser calculados pela troca de `sum` por `avg` ou `max`. A janela é redefinida de acordo com a cláusula *PARTITION BY*, nesse caso com o ano das vendas mensais. Normalmente a cláusula *ORDER BY* inclui um campo de data na

análise de série temporal. Omitir *ORDER BY* pode levar a resultados incorretos devido à maneira como os dados são armazenados na tabela subjacente; logo, é uma boa ideia incluí-la mesmo se você achar que os dados já estão classificados por data:

```
SELECT sales_month, sales
,sum(sales) over (partition by date_part('year',sales_month)
                  order by sales_month
                  ) as sales_ytd
FROM retail_sales
WHERE kind_of_business = 'Women''s clothing stores'
;
sales_month  sales  sales_ytd
-----------  -----  ---------
1992-01-01   1873   1873
1992-02-01   1991   3864
1992-03-01   2403   6267
...          ...    ...
1992-12-01   4416   31815
1993-01-01   2123   2123
1993-02-01   2005   4128
...          ...    ...
```

A consulta retorna um registro para cada `sales_month`, as vendas desse mês e o total acumulado `sales_ytd`. A série começa em 1992 e depois é retomada em janeiro de 1993, como o será para cada ano do conjunto de dados. Os resultados dos anos 2016 a 2020 estão representados em gráfico na Figura 3.15. Os quatro primeiros anos mostram padrões semelhantes no decorrer do ano, mas é claro que 2020 é bem diferente.

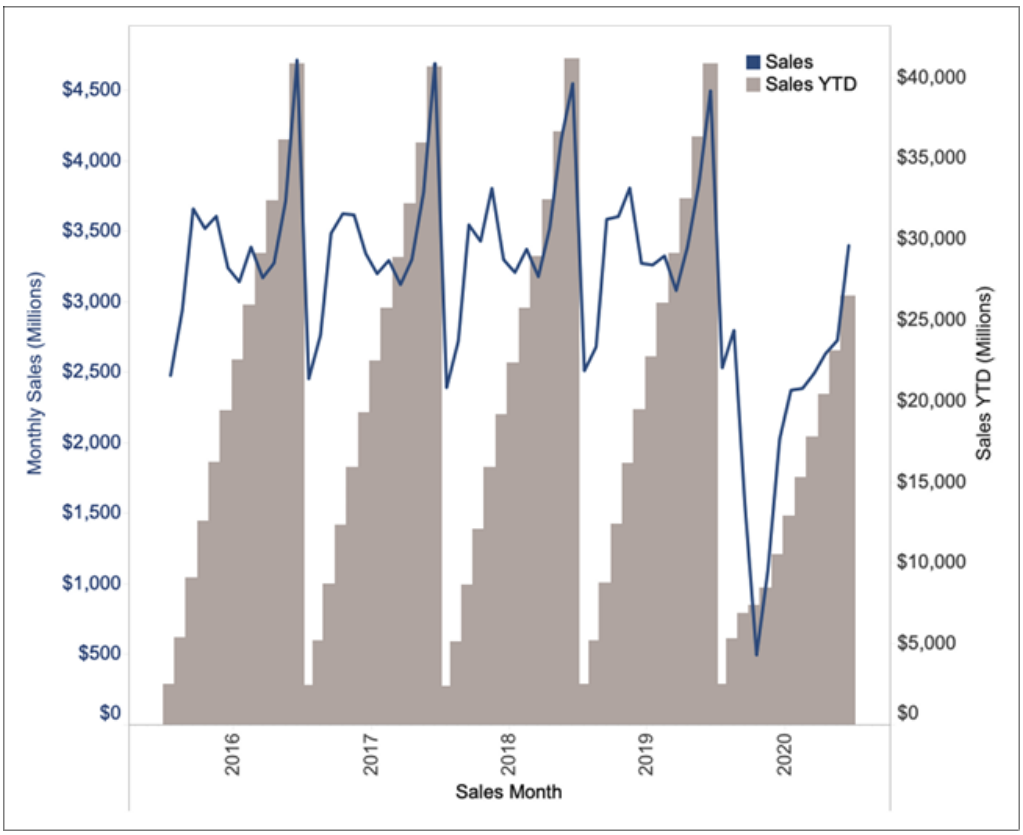


Figura 3.15: Vendas mensais e vendas anuais cumulativas nas lojas de roupas femininas.

Os mesmos resultados podem ser obtidos sem funções de janela, com o uso de uma self-*JOIN* que se beneficie de uma *JOIN* Cartesiana. Neste exemplo, os dois aliases de tabelas são unidos com *JOIN* no ano de `sales_month` para assegurarmos que os valores agregados sejam do mesmo ano, sendo redefinidos a cada ano. A cláusula *JOIN* também especifica que os resultados devem incluir `sales_month`s do alias `b` que sejam menores ou iguais ao `sales_month` do alias `a`. Em janeiro de 1992, só a linha desse mês do alias `b` atende a esse critério; em fevereiro de 1992, tanto janeiro quanto fevereiro de 1992 atendem ao critério e assim por diante:

```
SELECT a.sales_month, a.sales
,sum(b.sales) as sales_ytd
FROM retail_sales a
JOIN retail_sales b on
date_part('year',a.sales_month) = date_part('year',b.sales_month)
and b.sales_month <= a.sales_month
and b.kind_of_business = 'Women''s clothing stores'
WHERE a.kind_of_business = 'Women''s clothing stores'
```

```
GROUP BY 1,2
;
sales_month  sales  sales_ytd
-----------  -----  ---------
1992-01-01   1873   1873
1992-02-01   1991   3864
1992-03-01   2403   6267
...          ...    ...
1992-12-01   4416   31815
1993-01-01   2123   2123
1993-02-01   2005   4128
...          ...    ...
```

As funções de janela requerem menos caracteres de código, e geralmente é mais fácil acompanhar o que elas estão calculando quando estamos familiarizados com a sintaxe. Costuma haver mais de uma maneira de resolver um problema em SQL, e as janelas de tempo contínuas são um bom exemplo disso. Acho útil conhecer várias abordagens porque de vez em quando encontro um problema complexo que é resolvido de uma maneira melhor com o uso de uma abordagem que parece menos eficiente em outros contextos. Agora que examinamos as janelas de tempo contínuas, passaremos para nosso último tópico da análise de séries temporais com SQL: a sazonalidade.

Analisando com sazonalidade

Sazonalidade é qualquer padrão que se repita ao longo de intervalos regulares. Ao contrário de outros ruídos que ocorrem nos dados, a sazonalidade pode ser prevista. A palavra *sazonalidade* nos faz lembrar das quatro estações do ano – primavera, verão, outono, inverno – e alguns conjuntos de dados incluem esses padrões. Os padrões de consumo mudam com as estações, das roupas e alimentos que as pessoas compram ao dinheiro gasto com lazer e viagens. A temporada de compras das férias de inverno pode fortalecer ou arruinar muitos varejistas. Também pode existir sazonalidade em outras escalas de tempo, dos anos aos minutos. As eleições presidenciais dos Estados Unidos ocorrem a cada quatro anos, levando a padrões distintos de cobertura da mídia. A ciclicidade nos dias

úteis é comum, já que o trabalho e os estudos são predominantes de segunda a sexta-feira, enquanto as tarefas domésticas e as atividades de lazer predominam no fim de semana. A hora do dia é outro tipo de sazonalidade que os restaurantes vivenciam, com locais cheios nas horas do almoço e do jantar e vendas mais lentas no período intermediário.

Para saber se existe sazonalidade em uma série temporal, e qual sua escala, é útil representá-la em gráfico e procurar padrões visualmente. Tente encontrar agregações em diferentes níveis, do horário ao diário, semanalmente e mensalmente. Você também deve incorporar informações conhecidas sobre o conjunto de dados. Há padrões que você possa intuir de acordo com o que sabe sobre a entidade ou o processo que está sendo representado? Consulte especialistas no assunto, se houver algum disponível.

Examinaremos alguns padrões sazonais no conjunto de dados de vendas do varejo, mostrado na Figura 3.16. As joalherias têm um padrão altamente sazonal, com picos anuais em dezembro devido aos presentes dados nos feriados. As livrarias têm dois picos todo ano: um é em agosto, correspondente ao período de volta às aulas nos Estados Unidos; o outro pico começa em dezembro e permanece em janeiro, incluindo a época de dar presentes e o período de volta às aulas no semestre da primavera. Um terceiro exemplo seria o das mercearias, que têm muito menos sazonalidade mensal do que as outras duas séries temporais (embora provavelmente tenham sazonalidade no nível dos dias úteis e da hora do dia). Isso não é novidade: as pessoas precisam se alimentar durante o ano todo. As vendas das mercearias aumentam um pouco em dezembro para os feriados, e declinam em fevereiro, já que esse mês tem menos dias.

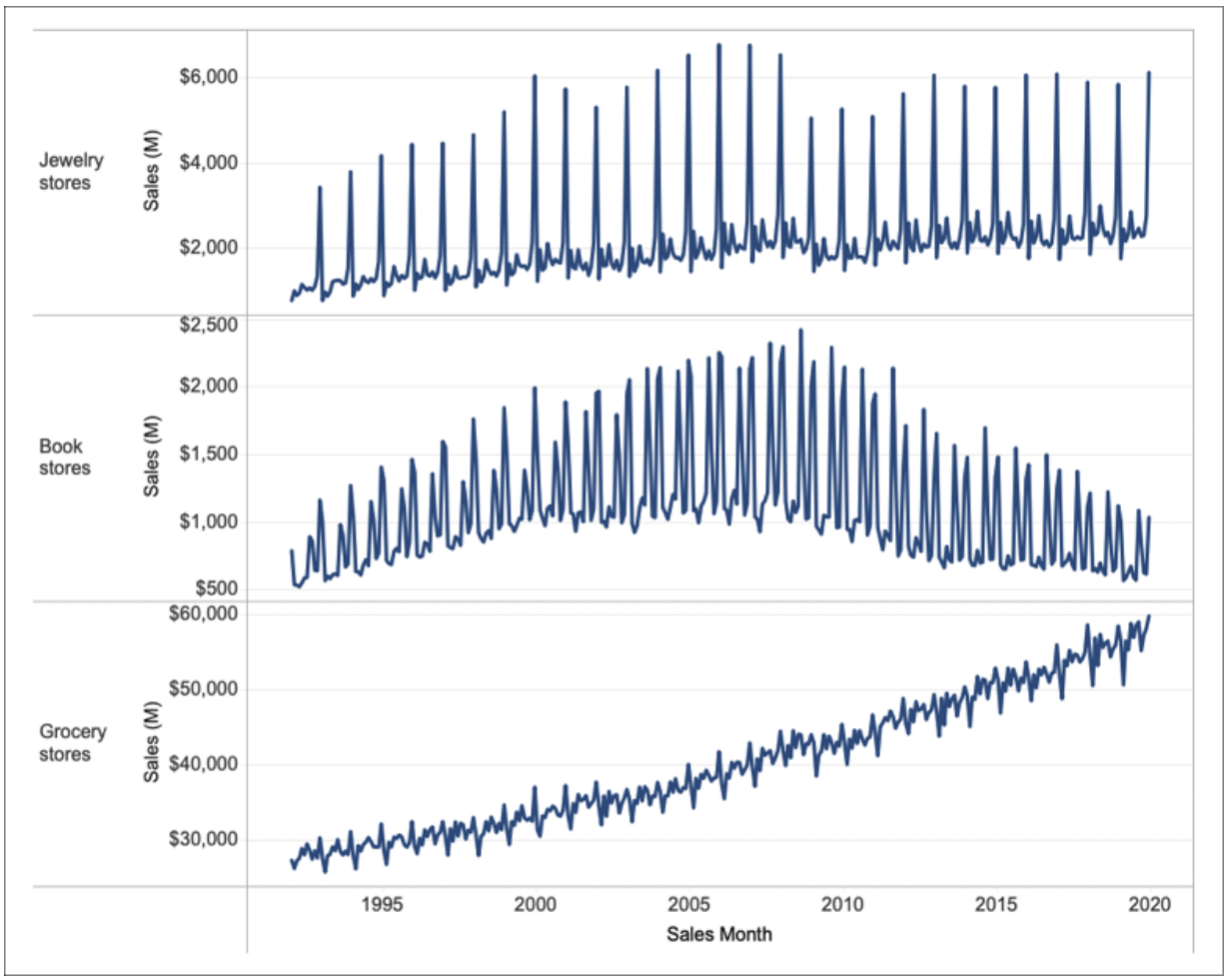


Figura 3.16: Exemplos de padrões de sazonalidade nas vendas das livrarias, mercearias e joalherias.

A sazonalidade pode assumir muitas formas, embora haja algumas abordagens comuns para a analisarmos em que isso não importa. Uma maneira de lidar com a sazonalidade é abrandá-la, agregando os dados em um período de tempo menos granular ou usando janelas contínuas, como vimos anteriormente. Outra maneira de trabalhar com dados sazonais é avaliá-los em relação a períodos de tempo semelhantes e analisar a diferença. Mostrarei vários modos de fazer isso a seguir.

Comparações período a período: YoY e MoM

Existem comparações período a período de vários tipos. A primeira seria comparar um período de tempo com o valor anterior da série, uma prática tão comum em análise que existem acrônimos para as comparações mais usadas. Dependendo do nível de agregação, a comparação pode ser YoY

(year-over-year, de ano a ano), MoM (month-over month, de mês a mês), DoD (day-over-day, de dia a dia) e assim por diante.

Para esses cálculos usarei a função `lag`, mais uma das funções de janela. A função `lag` retorna um valor anterior ou defasado de uma série. Ela tem a forma a seguir:

```
lag(return_value [,offset [,default]])
```

O valor de retorno (`return_value`) será qualquer campo do conjunto de dados e, portanto, pode ter qualquer tipo de dado. O *OFFSET* opcional indica quantas linhas teremos de voltar na partição para obter `return_value`. O padrão é 1, mas qualquer valor inteiro pode ser usado. Opcionalmente você também pode especificar um valor `default` para usar se não houver um registro defasado de onde recuperar um valor. Como outras funções de janela, `lag` também é calculada ao longo de uma partição, com a classificação sendo determinada pela cláusula *ORDER BY*. Se não for especificada uma cláusula *PARTITION BY*, `lag` retrocederá ao longo do conjunto de dados inteiro, e da mesma forma, se não for especificada uma cláusula *ORDER BY*, a ordem do banco de dados será usada. Geralmente é uma boa ideia incluir pelo menos uma cláusula *ORDER BY* em uma função de janela `lag` para controle da saída.

A função de janela `lead` opera da mesma forma que a função `lag`, exceto por retornar um valor subsequente como determinado pelo deslocamento. A alteração de *ORDER BY* de ascendente (*ASC*) para descendente (*DESC*) em uma série temporal produz o efeito de transformar uma instrução `lag` no equivalente da instrução `lead`. Alternativamente, um inteiro negativo pode ser usado como valor de *OFFSET* para retornar um valor de uma linha subsequente.

Aplicaremos isso ao nosso conjunto de dados de vendas do varejo para calcular o crescimento MoM e YoY. Nesta seção, nos concentraremos nas vendas das livrarias, já que sou um aficionado por esses estabelecimentos. Primeiro, examinaremos o que será exibido pela função `lag` ao retornarmos tanto o mês do deslocamento quanto os valores de vendas defasados:

```
SELECT kind_of_business, sales_month, sales
,lag(sales_month) over (partition by kind_of_business
                        order by sales_month
```

```
                        ) as prev_month
,lag(sales) over (partition by kind_of_business
                  order by sales_month
                  ) as prev_month_sales
FROM retail_sales
WHERE kind_of_business = 'Book stores'
;
kind_of_business  sales_month  sales  prev_month  prev_month_sales
----------------  -----------  -----  ----------  ----------------
Book stores       1992-01-01   790    (null)      (null)
Book stores       1992-02-01   539    1992-01-01  790
Book stores       1992-03-01   535    1992-02-01  539
...               ...          ...    ...         ...
```

Para cada linha, o `sales_month` anterior é retornado, assim como as vendas desse mês, e podemos confirmar isso inspecionando as primeiras linhas do conjunto de resultados. A primeira linha tem nulo para `prev_month` e `prev_month_sales` já que não há um registro anterior nesse conjunto de dados. Sabendo os valores que serão retornados pela função `lag`, podemos calcular a alteração percentual em relação ao valor anterior:

```
SELECT kind_of_business, sales_month, sales
,(sales / lag(sales) over (partition by kind_of_business
                           order by sales_month)
- 1) * 100 as pct_growth_from_previous
FROM retail_sales
WHERE kind_of_business = 'Book stores'
;
kind_of_business  sales_month  sales  pct_growth_from_previous
----------------  -----------  -----  ------------------------
Book stores       1992-01-01   790    (null)
Book stores       1992-02-01   539    -31.77
Book stores       1992-03-01   535    -0.74
...               ...          ...    ...
```

As vendas caíram 31,8% de janeiro a fevereiro, devido pelo menos em parte ao declínio sazonal após as férias e ao retorno às aulas do semestre de primavera. Elas só caíram 0,7% de fevereiro a março.

O cálculo da comparação YoY é semelhante, mas primeiro precisamos

agregar as vendas no nível anual. Já que estamos examinando um único `kind_of_business`, removerei esse campo do restante dos exemplos para simplificar o código:

```
SELECT sales_year, yearly_sales
,lag(yearly_sales) over (order by sales_year) as prev_year_sales
,(yearly_sales / lag(yearly_sales) over (order by sales_year)
-1) * 100 as pct_growth_from_previous
FROM
(
    SELECT date_part('year',sales_month) as sales_year
    ,sum(sales) as yearly_sales
    FROM retail_sales
    WHERE kind_of_business = 'Book stores'
    GROUP BY 1
) a
;
sales_year  yearly_sales  prev_year_sales  pct_growth_from_previous
----------  ------------  ---------------  -----------------------
1992.0      8327          (null)           (null)
1993.0      9108          8327             9.37
1994.0      10107         9108             10.96
...         ...           ...              ...
```

As vendas cresceram mais de 9,3% de 1992 a 1993, e quase 11% de 1993 a 1994. Esses cálculos de período a período são úteis, mas não nos permitem analisar a sazonalidade do conjunto de dados. Por exemplo, na Figura 3.17 estão representados os valores do crescimento percentual de MoM, e eles contêm o mesmo nível de sazonalidade da série temporal original.

Para resolver isso, a próxima seção demonstrará como usar SQL para comparar valores atuais com os valores do mesmo mês do ano anterior.

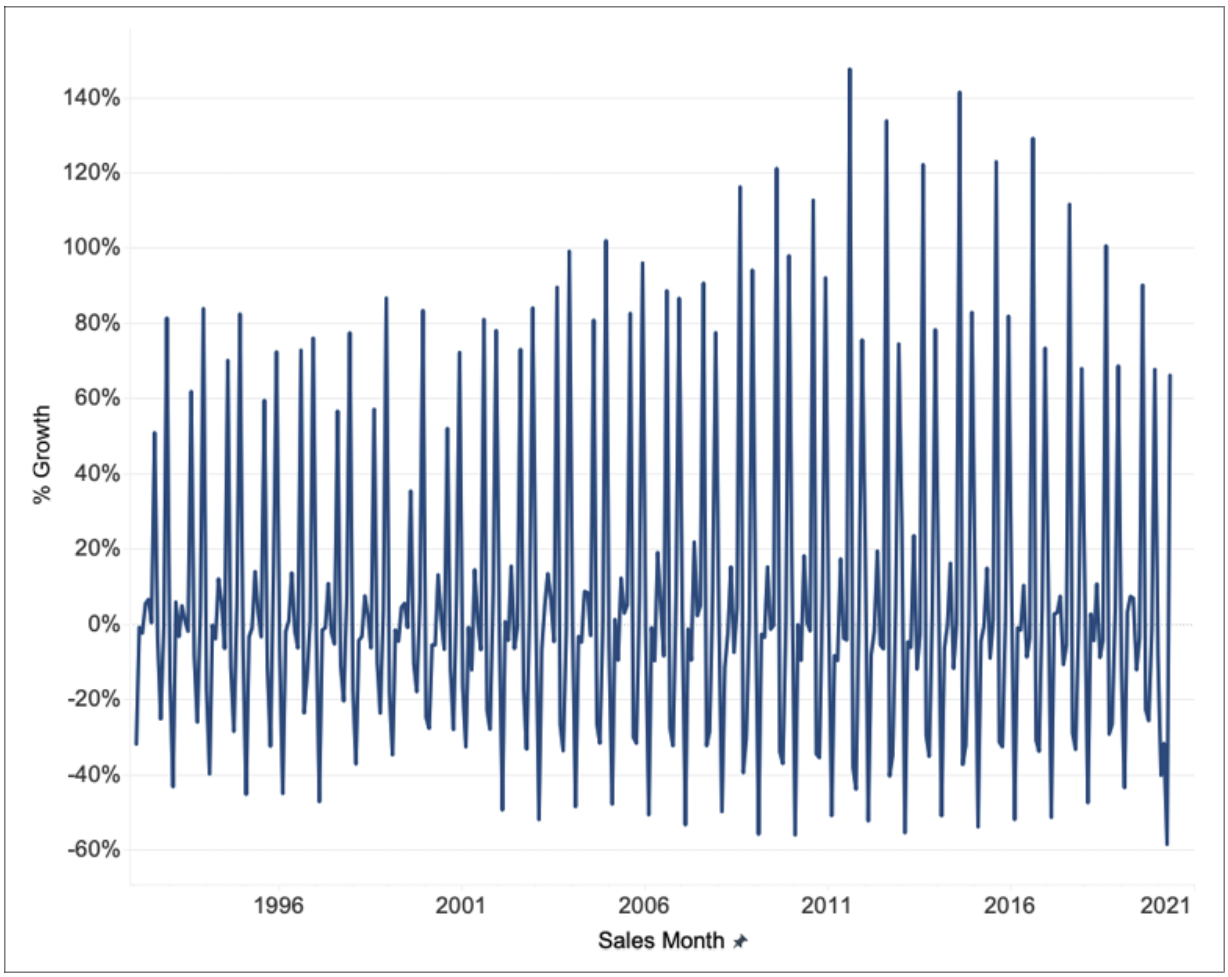


Figura 3.17: Crescimento percentual das vendas das livrarias varejistas dos EUA em relação ao mês anterior.

Comparações período a período: mesmo mês versus ano passado

Comparar dados de um período de tempo com os dados de outro período semelhante anterior pode ser uma maneira útil de acompanhar a sazonalidade. O período de tempo anterior pode ser o mesmo dia da semana anterior, o mesmo mês do ano anterior ou outra variação que faça sentido para o conjunto de dados.

Para fazer essa comparação, podemos usar a função `lag` junto com um particionamento inteligente: o da unidade de tempo com a qual queremos comparar o valor atual. Nesse caso, compararemos as vendas mensais com as vendas do mesmo mês do ano anterior. Por exemplo, as vendas de janeiro serão comparadas com as vendas de janeiro do ano anterior, as vendas de fevereiro serão comparadas com as vendas de fevereiro do ano

anterior e assim por diante.

Primeiro, lembre-se de que a função `date_part` retorna um valor numérico usado com o argumento "month":

```
SELECT sales_month
,date_part('month',sales_month)
FROM retail_sales
WHERE kind_of_business = 'Book stores'
;
sales_month  date_part
-----------  ---------
1992-01-01   1.0
1992-02-01   2.0
1992-03-01   3.0
...          ...
```

Em seguida, incluiremos `date_part` na cláusula *PARTITION BY* para que a função de janela procure o valor do número correspondente ao mês do ano anterior.

Este é um exemplo de como as cláusulas de função de janela podem incluir cálculos além de campos de banco de dados, o que dá a elas ainda mais versatilidade. Acho útil verificar resultados intermediários para ter uma ideia do que a consulta final retornará; logo, a primeira coisa que faremos será confirmar se a função `lag` com `partition by date_part('month', sales_month)` retorna os valores desejados:

```
SELECT sales_month, sales
,lag(sales_month) over (partition by date_part('month',sales_month)
                        order by sales_month
                        ) as prev_year_month
,lag(sales) over (partition by date_part('month',sales_month)
                  order by sales_month
                  ) as prev_year_sales
FROM retail_sales
WHERE kind_of_business = 'Book stores'
;
sales_month  sales  prev_year_month  prev_year_sales
-----------  -----  --------------  --------------
```

```
1992-01-01   790    (null)          (null)
1993-01-01   998    1992-01-01      790
1994-01-01   1053   1993-01-01      998
...          ...    ...             ...
1992-02-01   539    (null)          (null)
1993-02-01   568    1992-02-01      539
1994-02-01   635    1993-02-01      568
...          ...     ...            ...
```

A primeira função `lag` retorna o mesmo mês do ano anterior, o que podemos verificar examinando o valor de `prev_year_month`. A linha de `sales_month` que contém 1993-01-01 retorna 1992-01-01 para `prev_year_month` como esperado, e as vendas do ano anterior (`prev_year_sales`), que são iguais a 790, coincidem com as vendas que vemos na linha 1992-01-01. Observe que `prev_year_month` e `prev_year_sales` são nulos para 1992, já que não há registros anteriores no conjunto de dados.

Agora que temos certeza de que, da forma como foi escrita, a função `lag` retorna os valores corretos, podemos calcular métricas de comparação como a diferença absoluta e a alteração percentual em relação ao ano anterior:

```
SELECT sales_month, sales
,sales - lag(sales) over (partition by
  date_part('month',sales_month)
                          order by sales_month
                          ) as absolute_diff
,(sales / lag(sales) over (partition by
  date_part('month',sales_month)
                          order by sales_month)
- 1) * 100 as pct_diff
FROM retail_sales
WHERE kind_of_business = 'Book stores'
;
sales_month  sales  absolute_diff  pct_diff
-----------  -----  -------------  --------
1992-01-01   790    (null)         (null)
1993-01-01   998    208            26.32
```

```
1994-01-01   1053   55              5.51
...          ...    ...             ...
```

Podemos então representar os resultados no gráfico da Figura 3.18 e ver mais facilmente os meses em que o crescimento foi excepcionalmente alto, como em janeiro de 2002, ou excepcionalmente baixo, como em dezembro de 2001.

Outra ferramenta de análise útil seria criar um gráfico que fizesse o alinhamento entre o mesmo período de tempo – nesse caso, meses – e cada série temporal – em anos. Para fazê-lo, criaremos um conjunto de resultados com uma linha para cada número ou nome referente ao mês e uma coluna para cada um dos anos a serem considerados. Para obter o mês, podemos usar a função `date_part` ou a função `to_char`, de acordo com se serão usados valores numéricos ou textuais para os meses. Em seguida, pivotaremos os dados usando uma função de agregação.

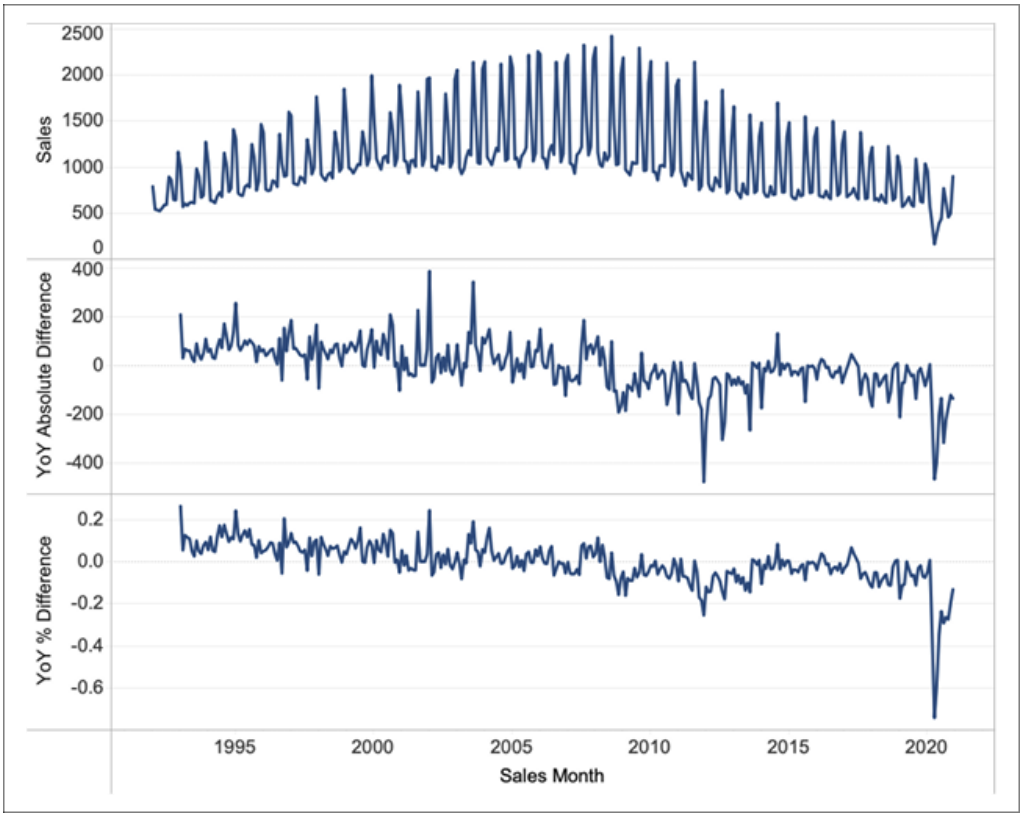


Figura 3.18: Vendas das livrarias, diferença absoluta YoY nas vendas e crescimento percentual YoY.

Este exemplo está usando a agregação `max`, mas, dependendo da análise,

`sum`, `count` ou outra agregação pode ser apropriada. Daremos destaque ao período de 1992 a 1994:

```
SELECT date_part('month',sales_month) as month_number
,to_char(sales_month,'Month') as month_name
,max(case when date_part('year',sales_month) = 1992 then sales end)
as sales_1992
,max(case when date_part('year',sales_month) = 1993 then sales end)
as sales_1993
,max(case when date_part('year',sales_month) = 1994 then sales end)
as sales_1994
FROM retail_sales
WHERE kind_of_business = 'Book stores'
and sales_month between '1992-01-01' and '1994-12-01'
GROUP BY 1,2
;
```

month_number	month_name	sales_1992	sales_1993	sales_1994
1.0	January	790	998	1053
2.0	February	539	568	635
3.0	March	535	602	634
4.0	April	523	583	610
5.0	May	552	612	684
6.0	June	589	618	724
7.0	July	592	607	678
8.0	August	894	983	1154
9.0	September	861	903	1022
10.0	October	645	669	732
11.0	November	642	692	772
12.0	December	1165	1273	1409

Ao alinhar os dados dessa forma, podemos ver algumas tendências imediatamente. Em dezembro temos as vendas mensais mais altas do ano. As vendas de 1994 foram em todos os meses mais altas do que as vendas de 1992 e 1993. É visível o aumento nas vendas de agosto a setembro, e isso é particularmente fácil de detectar em 1994.

Com um gráfico dos dados, como o da Figura 3.19, as tendências são muito mais fáceis de identificar. As vendas aumentaram de ano a ano em

todos os meses, embora o aumento em alguns meses tenha sido maior do que em outros. Com esses dados e o gráfico em mãos, podemos começar a construir um histórico sobre as vendas nas livrarias que pode ajudar no planejamento de estoque ou no agendamento de promoções de vendas, ou também pode servir como demonstração em um histórico maior sobre as vendas do varejo dos EUA.

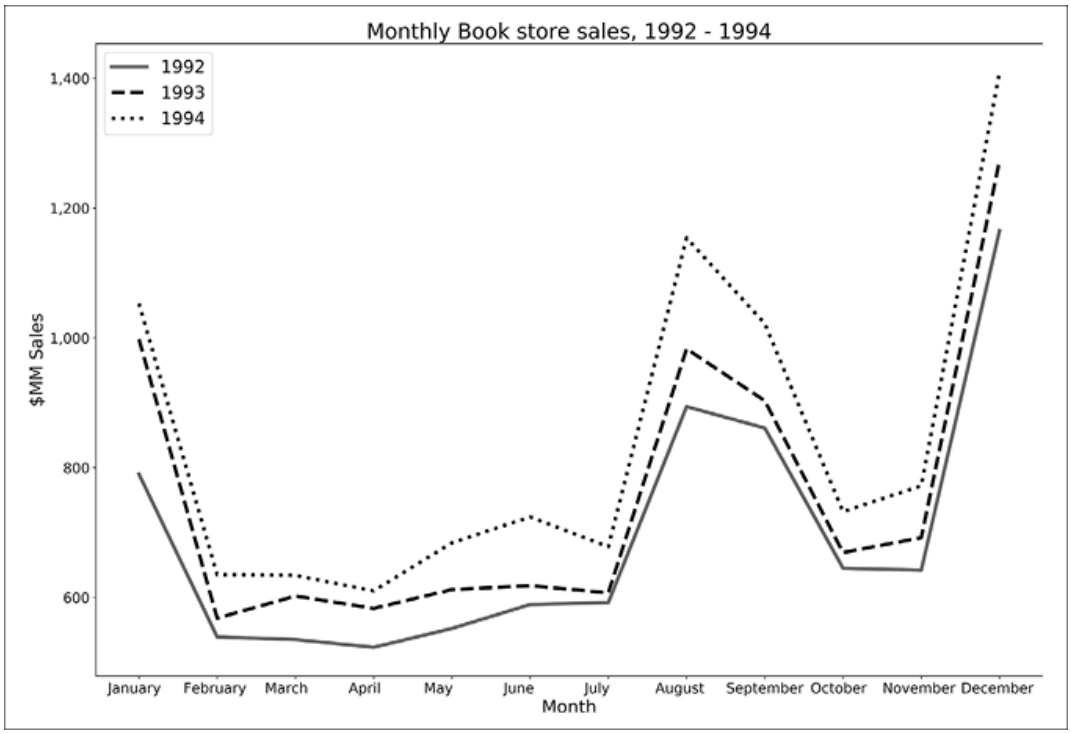


Figura 3.19: Vendas nas livrarias de 1992-1994, alinhadas por mês.

Com o SQL existem várias técnicas de remoção de ruído da sazonalidade para a comparação de dados de séries temporais. Nesta seção, vimos como comparar valores atuais com períodos anteriores comparáveis usando a função `lag` e como pivotar os dados com `date_part`, `to_char` e funções de agregação. A seguir, mostrarei algumas técnicas para a comparação de vários períodos anteriores para o posterior controle de dados de séries temporais com ruído.

Fazendo a comparação com vários períodos anteriores

Comparar dados com períodos anteriores comparáveis é uma maneira útil de reduzir o ruído que surge da sazonalidade. Fazer a comparação com um único período anterior pode ser insuficiente, principalmente se

esse período tiver sido afetado por eventos incomuns. Comparar uma segunda-feira com a segunda-feira anterior será difícil se uma delas for feriado. O mês do ano anterior pode ter sido incomum devido a eventos econômicos, a condições climáticas severas ou a uma falha no site que tenha alterado o comportamento típico. Comparar valores atuais com uma agregação de vários períodos anteriores pode ajudar a abrandar essas flutuações. Essas técnicas também combinam o que aprendemos sobre usar SQL para calcular períodos de tempo contínuos e resultados de períodos anteriores comparáveis.

A primeira técnica usa a função `lag`, como na última seção, mas aqui nos beneficiaremos do valor de deslocamento opcional. Lembre-se de que, quando nenhum deslocamento é fornecido para `lag`, a função retorna o valor imediato anterior de acordo com as cláusulas *PARTITION BY* e *ORDER BY*. Um valor de deslocamento igual a 2 pula o valor imediato anterior e retorna o valor anterior a ele, um valor de deslocamento igual a 3 retorna o valor de 3 linhas atrás e assim por diante.

Neste exemplo, compararemos as vendas do mês atual com as do mesmo mês nos três anos anteriores. Como sempre, começaremos inspecionando os valores retornados para confirmar se o SQL está funcionando como esperado:

```
SELECT sales_month, sales
,lag(sales,1) over (partition by date_part('month',sales_month)
                    order by sales_month
                    ) as prev_sales_1
,lag(sales,2) over (partition by date_part('month',sales_month)
                    order by sales_month
                    ) as prev_sales_2
,lag(sales,3) over (partition by date_part('month',sales_month)
                    order by sales_month
                    ) as prev_sales_3
FROM retail_sales
WHERE kind_of_business = 'Book stores'
;
sales_month  sales  prev_sales_1  prev_sales_2  prev_sales_3
-----------  -----  ------------  ------------  ------------
1992-01-01   790    (null)        (null)        (null)
```

```
1993-01-01   998    790          (null)       (null)
1994-01-01   1053   998          790          (null)
1995-01-01   1308   1053         998          790
1996-01-01   1373   1308         1053         998
...          ...    ...          ...          ...
```

Null foi retornado onde não existe registro anterior, e podemos confirmar que o valor correto do mesmo mês do ano anterior está sendo exibido. A partir daqui é possível calcular qualquer métrica de comparação que a análise demandar – nesse caso, o percentual da média móvel de três períodos anteriores:

```
SELECT sales_month, sales
,sales / ((prev_sales_1 + prev_sales_2 + prev_sales_3) / 3)
as pct_of_3_prev
FROM
(
    SELECT sales_month, sales
    ,lag(sales,1) over (partition by date_part('month',sales_month)
                        order by sales_month
                        ) as prev_sales_1
    ,lag(sales,2) over (partition by date_part('month',sales_month)
                        order by sales_month
                        ) as prev_sales_2
    ,lag(sales,3) over (partition by date_part('month',sales_month)
                        order by sales_month
                        ) as prev_sales_3
    FROM retail_sales
    WHERE kind_of_business = 'Book stores'
) a
;
sales_month  sales  pct_of_3_prev
-----------  -----  -------------
1995-01-01   1308   138.12
1996-01-01   1373   122.69
1997-01-01   1558   125.24
...          ...    ...
2017-01-01   1386   94.67
2018-01-01   1217   84.98
```

```
2019-01-01   1004   74.75
...          ...    ...
```

Podemos ver no resultado que as vendas de livros cresceram de acordo com a média móvel dos três anos anteriores no meio dos anos 1990, mas o cenário era diferente no fim dos anos 2010, quando a cada ano as vendas tornaram-se um percentual diminuto daquela média móvel de três anos.

Você deve ter notado que esse problema lembra o que vimos anteriormente ao calcular janelas de tempo contínuas. Como alternativa ao último exemplo, podemos usar uma função de janela `avg` com uma cláusula de frame. Para isso, *PARTITION BY* usará a mesma função `date_part`, e a cláusula *ORDER BY* será a mesma. Uma cláusula de frame será adicionada para incluir `rows between 3 preceding and 1 preceding`. Isso adicionará os valores das linhas 1, 2 e 3 anteriores, mas excluirá o valor da linha atual:

```
SELECT sales_month, sales
,sales / avg(sales) over (partition by
  date_part('month',sales_month)
                          order by sales_month
                          rows between 3 preceding and 1 preceding
                          ) as pct_of_prev_3
FROM retail_sales
WHERE kind_of_business = 'Book stores'
;
sales_month  sales  pct_of_prev_3
-----------  -----  -------------
1995-01-01   1308   138.12
1996-01-01   1373   122.62
1997-01-01   1558   125.17
...          ...    ...
2017-01-01   1386   94.62
2018-01-01   1217   84.94
2019-01-01   1004   74.73
...          ...    ...
```

Os resultados coincidem com os do exemplo anterior, confirmando que o código alternativo é equivalente.

Se você prestar atenção, notará que os valores das casas decimais são um pouco diferentes entre o resultado que usa três funções de janela `lag` e o que usa a função de janela `avg`. Isso ocorre devido à maneira como o banco de dados manipula o arredondamento decimal em cálculos intermediários. Para muitas análises, a diferença não será importante, mas tome cuidado se estiver trabalhando com dados financeiros ou outros dados altamente regulamentados.

A análise de dados com sazonalidade com frequência envolve tentar reduzir o ruído para chegarmos a decisões claras sobre as tendências subjacentes dos dados. Comparar pontos de dados com vários períodos de tempo anteriores pode fornecer uma tendência ainda mais branda para a comparação e a determinação do que está correndo no período de tempo atual. Isso requer que os dados incluam um histórico suficiente para fazermos essas comparações, mas, se tivermos uma série temporal de tamanho satisfatório, pode ser esclarecedor.

Conclusão

A análise de séries temporais é uma forma poderosa de análise de conjuntos de dados. Vimos como definir nossos dados para a análise com manipulações de data e hora. Abordamos as dimensões de data e aprendemos a aplicá-las ao cálculo de janelas de tempo contínuas. Examinamos os cálculos de período a período e como analisar dados com padrões de sazonalidade. No próximo capítulo, nos aprofundaremos em um tópico relacionado que se baseia na análise de séries temporais: a análise de coorte.

---

1 Os pontos de dados de outubro e novembro de 2020 foram suprimidos pelo divulgador devido a preocupações com a qualidade dos dados. Ficou mais difícil coletar dados por causa dos fechamentos de lojas durante a pandemia de 2020.

CAPÍTULO 4

Análise de Coorte

No Capítulo 3 abordamos a análise de séries temporais. Com essas técnicas em mãos, agora examinaremos um tipo relacionado de análise com muitas aplicações empresariais e em outras áreas: a análise de coorte.

Lembro-me da primeira vez que vi uma análise de coorte. Trabalhava em meu primeiro emprego de analista de dados em uma pequena startup. Estava examinando uma análise de compra na qual trabalhei com o CEO e ele sugeriu que eu dividisse a base de clientes em coortes para ver se o comportamento estava mudando com o tempo. Presumi que fosse alguma teoria extravagante da faculdade de administração que provavelmente seria inútil, mas ele era o CEO; logo, é claro que concordei. Acabou não sendo apenas uma brincadeira. Dividir populações em coortes e acompanhá-las com o tempo é uma maneira poderosa de analisar dados e evitar predisposições. As coortes podem fornecer pistas de como as subpopulações diferem umas das outras e de como mudam com o tempo.

Neste capítulo, primeiro examinaremos o que são coortes e discutiremos os elementos constitutivos de certos tipos de análise de coorte. Após uma introdução ao conjunto de dados de legisladores usado nos exemplos, aprenderemos a construir uma análise de retenção e a lidar com vários desafios, como definir a coorte e manipular dados esparsos. Em seguida, discutiremos a sobrevivência, o retorno e os cálculos cumulativos, que são semelhantes à análise de retenção na forma como o código SQL é estruturado. Para concluir, examinaremos como combinar a análise de coorte com a análise transversal para entender a composição de populações com o passar do tempo.

Coortes: uma estrutura de análise útil

Antes de passarmos para o código, definirei o que são coortes, considerarei as perguntas que podemos responder com esse tipo de

análise e descreverei os componentes encontrados em qualquer análise de coorte.

Uma *coorte* é um grupo de indivíduos que compartilhavam algumas características de interesse, descritas a seguir, no momento em que começamos a observá-los. Os membros da coorte costumam ser pessoas, mas podem ser qualquer tipo de entidade que quisermos estudar: empresas, produtos, ou fenômenos do mundo físico. Os indivíduos de uma coorte podem ter consciência de sua associação, assim como as crianças de uma sala de aula do primeiro ano sabem que fazem parte de um grupo de alunos dessa série, ou como os participantes do estudo de um medicamento sabem que fazem parte de um grupo que está recebendo um tratamento. Em outras situações, as entidades são agrupadas em coortes virtualmente, como quando uma empresa de software agrupa todos os clientes conquistados em um ano específico para estudar por quanto tempo eles permanecem sendo clientes. É sempre importante considerar as implicações éticas de agrupar entidades em coortes sem seu conhecimento se algum tratamento diferente for aplicado a elas.

A *análise de coorte* é uma maneira útil de comparar grupos de entidades com o passar do tempo. Muitos comportamentos importantes levam semanas, meses ou anos para ocorrer ou evoluir, e a análise de coorte é uma maneira de entender essas alterações. A análise de coorte fornece uma estrutura para a detecção de correlações entre as características da coorte e essas tendências de longo prazo, o que pode levar a hipóteses sobre os fatores causais. Por exemplo, os clientes conquistados por meio de uma campanha de marketing podem ter padrões de compra a longo prazo diferentes dos que foram persuadidos por um amigo a experimentar os produtos de uma empresa. A análise de coorte pode ser usada para monitorar novas coortes de usuários ou clientes e avaliar como elas se comparam a coortes anteriores. Esse monitoramento pode fornecer um sinal de alerta precoce de que algo deu errado (ou certo) para novos clientes. A análise de coorte também é usada para minerar dados históricos. Os testes A/B, discutidos no Capítulo 7, são o padrão-ouro para a determinação de causalidade, mas não podemos voltar no tempo e executar testes para cada pergunta sobre o passado na qual estivermos

interessados. É claro que devemos ter cuidado ao dar significado causal à análise de coorte em vez de usar a análise como uma maneira de entender os clientes e gerar hipóteses que possam ser testadas rigorosamente no futuro.

A análise de coorte tem três componentes: o agrupamento da coorte, uma série temporal de dados ao longo da qual a coorte é observada, e uma métrica que avalia uma ação executada por membros da coorte.

Geralmente o *agrupamento da coorte* é baseado em uma data inicial: a data da primeira compra ou da assinatura do cliente, a data em que um aluno iniciou os estudos e assim por diante. No entanto, as coortes também podem ser formadas a partir de outras características que sejam inatas ou mudem com o tempo. As qualidades inatas incluem o ano de nascimento e o país de origem, ou o ano em que uma empresa foi fundada. As características que podem mudar com o tempo poderiam ser a cidade de residência e o estado civil. Quando elas forem usadas, é preciso ter o cuidado de agrupar a coorte somente com base na data inicial, ou as entidades podem saltar entre grupos de coorte.

 Coorte ou segmento?

Esses dois termos costumam ser usados de maneiras semelhantes, ou até mesmo de forma intercambiável, mas é útil definir uma diferença entre eles a título de clareza. Uma coorte é um grupo de usuários (ou outras entidades) que têm uma data inicial comum e são acompanhados com o passar do tempo. Um *segmento* é um agrupamento de usuários que compartilham uma característica ou um conjunto de características comuns em um momento no tempo, não importando a data inicial. Como as coortes, os segmentos podem ser baseados em fatores inatos como a idade ou em características comportamentais. Um segmento dos usuários que fizeram uma assinatura no mesmo mês pode ser inserido em uma coorte e acompanhado com o passar do tempo. Ou diferentes agrupamentos de usuários podem ser explorados com a análise de coorte para que você possa ver quais têm as características de maior valor. As análises que abordaremos neste capítulo, como a retenção, podem ajudar a fornecer os dados concretos existentes por trás de segmentos de marketing.

O segundo componente de qualquer análise de coorte é a *série temporal*. Trata-se de uma série de compras, logins, interações ou outras ações executadas pelos clientes ou entidades participantes da coorte. É

importante que a série temporal abranja todo o tempo de vida das entidades ou haverá *viés de sobrevivência (survivorship bias)* nas coortes iniciais. O viés de sobrevivência ocorre quando só clientes que continuaram a comprar encontram-se no conjunto de dados; clientes indecisos são excluídos porque não estão mais presentes; logo, o restante dos clientes parece ser de qualidade ou compatibilidade mais alta em comparação com os de coortes mais novas (consulte "Viés de sobrevivência", na página 195). Também é importante que haja uma série temporal que seja suficientemente longa para as entidades concluírem a ação de interesse. Por exemplo, se os clientes estiverem tendendo a comprar uma vez ao mês, uma série temporal de vários meses será necessária. Se, por outro lado, as compras só estiverem ocorrendo uma vez ao ano, uma série temporal de vários anos seria preferível. Inevitavelmente, clientes conquistados mais recentemente não terão se beneficiado do mesmo tempo para executar as ações que clientes que foram conquistados no passado distante. Para gerar uma normalização, geralmente a análise de coorte mede o número de períodos que se passaram a partir de uma data inicial, em vez de usar meses do calendário. Dessa forma, as coortes podem ser comparadas no período 1, no período 2, e assim por diante para sabermos como evoluíram com o passar do tempo, não importando o mês em que a ação realmente ocorreu. Os intervalos podem ser em dias, semanas ou anos.

A *métrica de agregação* deve estar relacionada às ações que importam para a integridade da organização, como os clientes continuando a usar ou comprar o produto. Os valores das métricas são agregados ao longo da coorte, geralmente com `sum`, `count` ou `average`, embora qualquer agregação relevante funcione. O resultado será uma série temporal que poderá então ser usada para explicar as alterações no comportamento com o passar do tempo.

Neste capítulo, abordarei quatro tipos de análise de coorte: retenção, sobrevivência, retorno ou comportamento de compra repetida, e comportamento cumulativo.

Retenção

A retenção se preocupa com se o membro da coorte tem um registro na série temporal em uma data específica, o que é expresso como um número de períodos a partir da data inicial. Isso é útil em qualquer tipo

de organização em que ações repetidas sejam esperadas, desde participar de um jogo online a usar um produto ou renovar uma assinatura, e ajuda a responder às perguntas de o quanto um produto é atraente ou cativante e quantas entidades devem aparecer em datas futuras.

Sobrevivência

A sobrevivência se preocupa com quantas entidades permaneceram no conjunto de dados por um período de tempo específico ou por um período mais longo, independentemente do número ou da frequência das ações até esse momento. Ela é útil para responder a perguntas sobre a proporção da população da qual podemos esperar permanência – de maneira positiva por não ter mudado de fornecedor ou falecido, ou de maneira negativa por não ter se graduado ou não atender a algum requisito.

Retorno

O retorno ou o comportamento de compra repetida se preocupa com se uma ação ocorreu mais vezes do que as definidas em algum limite mínimo – com frequência apenas mais de uma vez – durante uma janela de tempo fixa. Esse tipo de análise é útil em situações em que o comportamento é intermitente e imprevisível, como no varejo, em que ela caracteriza a parcela de compradores recorrentes em cada coorte dentro de uma janela de tempo fixa.

Cálculos cumulativos

Os cálculos cumulativos se preocupam com o número ou o valor total medido em uma ou mais janelas de tempo fixas, independentemente de quando ele ocorreu durante essa janela. Geralmente são usados em cálculos do LTV (lifetime value, valor de tempo de vida) ou CLTV (customer lifetime value, valor de tempo de vida do cliente) .

Os quatro tipos de análise de coorte nos permitem comparar subgrupos e entender como eles diferem com o passar do tempo para tomarmos decisões melhores relacionadas às áreas de produtos, marketing e financeira. Os cálculos são semelhantes para os diferentes tipos; logo, começaremos examinando a retenção e, em seguida, mostrarei como modificar o código para calcular os outros tipos. Antes de construirmos nossa análise de coorte, descreverei o conjunto de dados que usaremos

nos exemplos deste capítulo.

Conjunto de dados de legisladores

Os exemplos de SQL deste capítulo usarão um conjunto de dados de membros antigos e atuais do Congresso dos Estados Unidos mantido em um repositório do GitHub (*https://github.com/unitedstates/congress-legislators*). Nos EUA, o Congresso é responsável por criar leis ou regulamentos; logo, seus membros também são conhecidos como legisladores. Já que o conjunto de dados é um arquivo JSON, apliquei algumas transformações para produzir um modelo de dados mais adequado para a análise, e postei os dados em um formato apropriado para o acompanhamento junto aos exemplos da pasta `legislators` do livro no GitHub (*https://oreil.ly/H2tYP*).

O repositório de código-fonte tem um ótimo dicionário de dados, então não repetirei todos os detalhes aqui. Fornecerei alguns detalhes, entretanto, que devem ajudar quem não estiver familiarizado com o governo dos EUA a acompanhar a análise deste capítulo.

O Congresso tem duas câmaras, o Senado ("sen" no conjunto de dados) e a Câmara dos Representantes ("rep"). Cada estado tem dois senadores e eles são eleitos para mandatos (terms) de seis anos. Os representantes são alocados aos estados de acordo com a população; cada representante tem um distrito pelo qual ele deve responder. Os representantes são eleitos para mandatos de dois anos. Os mandatos reais das duas câmaras pode ser mais curto caso o legislador faleça ou seja eleito ou promovido para um cargo mais alto. Quanto maior for o tempo que os legisladores permanecerem no gabinete, mais poder e influência eles acumularão por meio de posições de liderança e, portanto, é comum a candidatura à reeleição. Para concluir, um legislador pode pertencer a um partido político ou ser "independente". Na era moderna, a grande maioria dos legisladores é democrata ou republicana e a disputa entre os dois partidos é famosa. Ocasionalmente os legisladores trocam de partido ainda estando no mandato.

Na análise faremos uso de duas tabelas: `legislators` e `legislators_terms`. A tabela `legislators` contém uma lista de todas as pessoas que foram

incluídas no conjunto de dados, com o aniversário, o gênero e um conjunto de campos de ID que pode ser usado na busca da pessoa em outros conjuntos de dados. A tabela `legislators_terms` contém um registro para cada mandato de cada legislador, com a data de início e fim, e outros atributos como a câmara e o partido. O campo `id_bioguide` é usado como identificador exclusivo de um legislador e aparece nas duas tabelas. A Figura 4.1 exibe uma amostra dos dados de `legislators`. A Figura 4.2 exibe uma amostra dos dados de `legislators_terms`.

*	full_name	first_name	last_name	birthday	gender	id_bioguide	id_govtrack
1	Sherrod Brown	Sherrod	Brown	1952-11-09	M	B000944	400050
2	Maria Cantwell	Maria	Cantwell	1958-10-13	F	C000127	300018
3	Benjamin L. Cardin	Benjamin	Cardin	1943-10-05	M	C000141	400064
4	Thomas R. Carper	Thomas	Carper	1947-01-23	M	C000174	300019
5	Robert P. Casey, Jr.	Robert	Casey	1960-04-13	M	C001070	412246
6	Dianne Feinstein	Dianne	Feinstein	1933-06-22	F	F000062	300043
7	Russ Fulcher	Russ	Fulcher	1973-07-19	M	F000469	412773
8	Amy Klobuchar	Amy	Klobuchar	1960-05-25	F	K000367	412242
9	Robert Menendez	Robert	Menendez	1954-01-01	M	M000639	400272
10	Bernard Sanders	Bernard	Sanders	1941-09-08	M	S000033	400357
11	Debbie Stabenow	Debbie	Stabenow	1950-04-29	F	S000770	300093
12	Jon Tester	Jon	Tester	1956-08-21	M	T000464	412244
13	Sheldon Whitehouse	Sheldon	Whitehouse	1955-10-20	M	W000802	412247
14	Nanette Diaz Barragán	Nanette	Barragán	1976-09-15	F	B001300	412687
15	John Barrasso	John	Barrasso	1952-07-21	M	B001261	412251
16	Roger F. Wicker	Roger	Wicker	1951-07-05	M	W000437	400432
17	Lamar Alexander	Lamar	Alexander	1940-07-03	M	A000360	300002
18	Susan M. Collins	Susan	Collins	1952-12-07	F	C001035	300025
19	John Cornyn	John	Cornyn	1952-02-02	M	C001056	300027

Figura 4.1: Amostra da tabela legislators.

Agora que sabemos o que é a análise de coorte e conhecemos o conjunto de dados que usaremos nos exemplos, examinaremos como escrever SQL para a análise de retenção. A principal pergunta que o SQL nos ajudará a responder é: uma vez que os representantes assumem o mandato, por quanto tempo eles mantêm seus empregos?

*	id_bioguide	term_id	term_type	term_start	term_end	state	district	party
1	B000944	B000944-0	rep	1993-01-05	1995-01-03	OH	13	Democrat
2	C000127	C000127-0	rep	1993-01-05	1995-01-03	WA	1	Democrat
3	C000141	C000141-0	rep	1987-01-06	1989-01-03	MD	3	Democrat
4	C000174	C000174-0	rep	1983-01-03	1985-01-03	DE	0	Democrat
5	C001070	C001070-0	sen	2007-01-04	2013-01-03	PA	(null)	Democrat
6	F000062	F000062-0	sen	1992-11-10	1995-01-03	CA	(null)	Democrat
7	F000469	F000469-0	rep	2019-01-03	2021-01-03	ID	1	Republican
8	K000367	K000367-0	sen	2007-01-04	2013-01-03	MN	(null)	Democrat
9	M000639	M000639-0	rep	1993-01-05	1995-01-03	NJ	13	Democrat
10	S000033	S000033-0	rep	1991-01-03	1993-01-03	VT	0	Independent
11	S000770	S000770-0	rep	1997-01-07	1999-01-03	MI	8	Democrat
12	T000464	T000464-0	sen	2007-01-04	2013-01-03	MT	(null)	Democrat
13	W000802	W000802-0	sen	2007-01-04	2013-01-03	RI	(null)	Democrat
14	B001300	B001300-0	rep	2017-01-03	2019-01-03	CA	44	Democrat
15	B001261	B001261-0	sen	2007-06-25	2013-01-03	WY	(null)	Republican
16	W000437	W000437-0	rep	1995-01-04	1997-01-03	MS	1	Republican
17	A000360	A000360-0	sen	2003-01-07	2009-01-03	TN	(null)	Republican
18	C001035	C001035-0	sen	1997-01-07	2003-01-03	ME	(null)	Republican
19	C001056	C001056-0	sen	2002-11-30	2003-01-03	TX	(null)	Republican

Figura 4.2: Amostra da tabela legislators_terms.

Retenção

Um dos tipos mais comuns de análise de coorte é a *análise de retenção*. Reter é manter ou continuar algo. Muitas habilidades precisam ser praticadas para ser retidas. Geralmente as empresas querem que seus clientes continuem comprando seus produtos ou usando seus serviços, já que reter os clientes é mais lucrativo do que conquistar clientes novos. Os empregadores querem reter seus funcionários, porque recrutar substitutos é caro e demorado. Os políticos eleitos buscam a reeleição para continuar trabalhando nas prioridades de seus representados.

A principal questão da análise de retenção é se o tamanho inicial da coorte – número de assinantes ou de funcionários, valor gasto ou outra métrica importante – permanecerá constante, diminuirá ou aumentará com o tempo. Quando há um aumento ou uma diminuição, a quantidade e a velocidade das alterações também são questões interessantes. Na maioria das análises de retenção, o tamanho inicial tende a diminuir com o tempo, já que uma coorte pode perder, mas não ganhar, novos membros uma vez que é formada. A receita é uma exceção intrigante, porque uma coorte de clientes pode gastar em meses subsequentes mais do que gastou

no primeiro mês coletivamente, mesmo se alguns dos clientes trocarem de fornecedor.

A análise de retenção usa a contagem (`count`) de entidades ou a soma (`sum`) de quantias em dinheiro ou de ações presentes no conjunto de dados para cada período a partir da data inicial, e faz a normalização dividindo esse número pela contagem ou a soma das entidades, das quantias em dinheiro ou das ações do primeiro período de tempo. O resultado é expresso como um percentual e a retenção no período inicial é sempre de 100%. Com o tempo, geralmente a retenção baseada em contagens diminui e não pode nunca exceder 100%, enquanto a retenção baseada em dinheiro ou ações, embora com frequência decline, pode aumentar e ser maior do que 100% em um período de tempo. Normalmente a saída da análise de retenção é exibida na forma de tabela ou gráfico, que é conhecido como curva de retenção. Veremos vários exemplos de curvas de retenção posteriormente neste capítulo.

Os gráficos de curvas de retenção podem ser usados na comparação de coortes. A primeira característica na qual devemos prestar atenção é na forma da curva nos períodos iniciais, em que geralmente há uma queda inicial acentuada. Em muitas aplicações para consumidores, perder metade de uma coorte nos primeiros meses é comum. Uma coorte com uma curva que seja mais ou menos acentuada do que outras pode indicar alterações na fonte de aquisição de produtos ou de conquista de clientes merecedoras de uma investigação mais profunda. Uma segunda característica que deve ser procurada é se a curva se achata após algum número de períodos ou continua diminuindo rapidamente até chegar a zero. Uma curva achatada indica que há um momento no tempo a partir do qual grande parte da coorte que permaneceu ficou nesse estado indefinidamente. Uma curva de retenção que flexiona para cima, às vezes chamada de curva sorridente, pode ocorrer se os membros da coorte voltarem ou forem reativados após saírem do conjunto de dados durante algum período. Para concluir, curvas de retenção que medem a receita proveniente de assinaturas são monitoradas em busca de sinais de receita crescente por cliente com o passar do tempo, uma indicação de um negócio de software SaaS lucrativo.

Esta seção mostrará como criar uma análise de retenção, como adicionar

agrupamentos de coorte a partir da própria série temporal e de outras tabelas, e como manipular dados ausentes e esparsos que podem ocorrer na série temporal. Com essa estrutura em mãos, na seção subsequente você aprenderá como fazer modificações para criar os outros tipos de análise de coorte relacionados. Como resultado, esta seção sobre retenção será a mais longa do capítulo, já que você construirá o código e desenvolverá sua percepção dos cálculos.

SQL para uma curva de retenção básica

Na análise de retenção, como em outras análises de coorte, precisamos de três componentes: a definição da coorte, uma série temporal de ações e uma métrica de agregação que meça algo relevante para a organização ou o processo. Em nosso caso, os membros da coorte serão os legisladores, a série temporal será composta dos mandatos de cada legislador, e a métrica de interesse será a contagem de quem ainda está no mandato a cada período a partir da data inicial.

Começaremos calculando a retenção básica, antes de passarmos para os exemplos que incluem vários agrupamentos de coorte. A primeira etapa é encontrar a data em que cada legislador assumiu o mandato (`first_term`). Usaremos essa data para calcular o número de períodos para cada data subsequente da série temporal. Para fazê-lo, encontre o valor mínimo (`min`) de `term_start` (início do mandato) e crie o agrupamento (com *GROUP BY*) por cada `id_bioguide`, o identificador exclusivo de um legislador:

```
SELECT id_bioguide
,min(term_start) as first_term
FROM legislators_terms
GROUP BY 1
;
id_bioguide  first_term
-----------  ----------
A000118      1975-01-14
P000281      1933-03-09
K000039      1933-03-09
...          ...
```

A próxima etapa é inserir esse código em uma subconsulta e fazer a

junção (com *JOIN*) dele com a série temporal. A função `age` é aplicada para calcular os intervalos entre cada `term_start` e o `first_term` de cada legislador. A aplicação das funções `date_part` ao resultado, com o ano, fará a transformação para o número de períodos anuais. Já que as eleições ocorrem a cada dois ou seis anos, usaremos anos como o intervalo de tempo para calcular os períodos. Poderíamos usar um intervalo mais curto, mas nesse conjunto de dados há pouca flutuação diária ou semanal. A contagem de legisladores com registros desse período será o número retido:

```
SELECT date_part('year',age(b.term_start,a.first_term)) as period
,count(distinct a.id_bioguide) as cohort_retained
FROM
(
    SELECT id_bioguide, min(term_start) as first_term
    FROM legislators_terms
    GROUP BY 1
) a
JOIN legislators_terms b on a.id_bioguide = b.id_bioguide
GROUP BY 1
;
period  cohort_retained
------  ---------------
0.0     12518
1.0     3600
2.0     3619
...     ...
```


Em bancos de dados que suportam a função `datediff`, a estrutura `date_part` e `age` pode ser substituída por esta função mais simples:

```
datediff('year',first_term,term_start)
```

Alguns bancos de dados, como o Oracle, inserem `date_part` no final:

```
datediff(first_term,term_start,'year'
```

Agora que temos os períodos e o número de legisladores retidos em cada um deles, a última etapa é calcular o tamanho total da coorte (`cohort_size`) e preencher cada linha para que a coorte retida (`cohort_retained`) possa ser dividida por ele. A função de janela

`first_value` retorna o primeiro registro da cláusula *PARTITION BY*, de acordo com a ordem definida em *ORDER BY*, uma maneira conveniente de obter o tamanho da coorte em cada linha. Nesse caso, `cohort_size` vem do primeiro registro do conjunto de dados inteiro; logo, *PARTITION BY* é omitida:

```
first_value(cohort_retained) over (order by period) as cohort_size
```

Para encontrar o percentual de retenção, divida o valor de `cohort_retained` por esse mesmo cálculo:

```
SELECT period
,first_value(cohort_retained) over (order by period) as cohort_size
,cohort_retained
,cohort_retained /
first_value(cohort_retained) over (order by period) as pct_retained
FROM
(
    SELECT date_part('year',age(b.term_start,a.first_term)) as
  period
    ,count(distinct a.id_bioguide) as cohort_retained
    FROM
    (
        SELECT id_bioguide, min(term_start) as first_term
        FROM legislators_terms
        GROUP BY 1
    ) a
    JOIN legislators_terms b on a.id_bioguide = b.id_bioguide
    GROUP BY 1
) aa
;
period  cohort_size  cohort_retained  pct_retained
------  -----------  ---------------  ------------
0.0     12518        12518            1.0000
1.0     12518        3600             0.2876
2.0     12518        3619             0.2891
...     ...          ...              ...
```

Já temos um cálculo da retenção, e podemos ver que há uma grande diminuição entre o total de legisladores retidos no período 0, ou em sua

data inicial, e a parcela com registro de outro mandato que começa um ano depois. A representação dos resultados em gráfico, como na Figura 4.3, demonstra como a curva se achata e acaba chegando a zero, já que até mesmo os legisladores que prestam serviços por mais tempo falecem ou se aposentam.

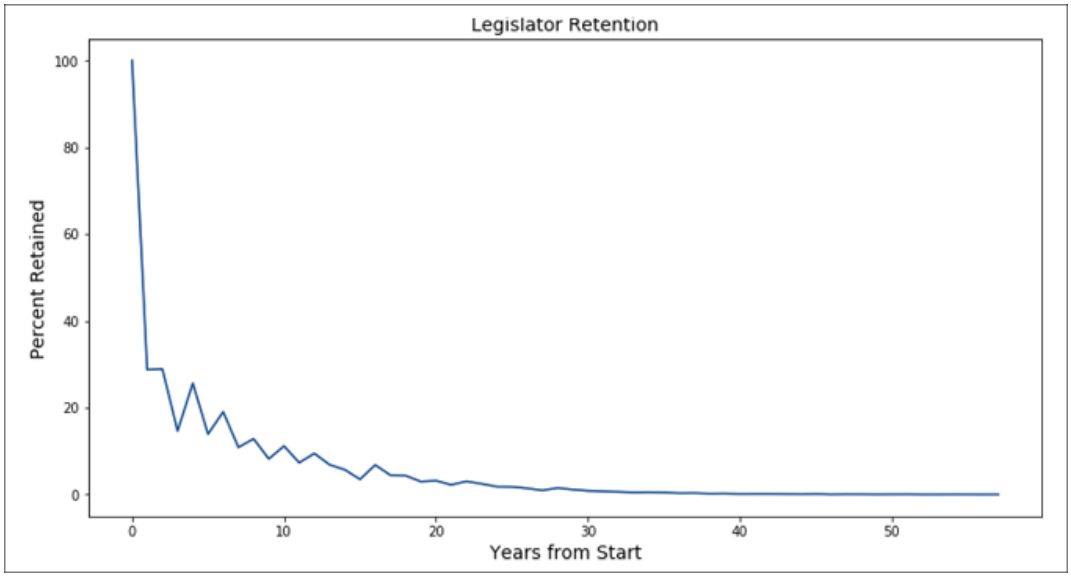


Figura 4.3: Retenção desde o início do primeiro mandato dos legisladores dos EUA.

Podemos pegar o resultado da retenção da coorte e remodelar os dados para exibi-los no formato de tabela. Pivotaremos e achataremos os resultados usando uma função de agregação com uma instrução CASE; `max` está sendo usada neste exemplo, mas outras agregações como `min` ou `avg` retornariam o mesmo resultado. A retenção é calculada para os anos 0 a 4, mas outros anos podem ser adicionados se seguirmos o mesmo padrão:

```
SELECT cohort_size
,max(case when period = 0 then pct_retained end) as yr0
,max(case when period = 1 then pct_retained end) as yr1
,max(case when period = 2 then pct_retained end) as yr2
,max(case when period = 3 then pct_retained end) as yr3
,max(case when period = 4 then pct_retained end) as yr4
FROM
(
    SELECT period
    ,first_value(cohort_retained) over (order by period)
```

```
     as cohort_size
    ,cohort_retained
     / first_value(cohort_retained) over (order by period)
     as pct_retained
    FROM
    (
        SELECT
        date_part('year',age(b.term_start,a.first_term)) as period
        ,count(*) as cohort_retained
        FROM
        (
            SELECT id_bioguide, min(term_start) as first_term
            FROM legislators_terms
            GROUP BY 1
        ) a
        JOIN legislators_terms b on a.id_bioguide = b.id_bioguide
        GROUP BY 1
    ) aa
) aaa
GROUP BY 1
;
cohort_size  yr0     yr1     yr2     yr3     yr4
-----------  ------  ------  ------  ------  ------
12518        1.0000  0.2876  0.2891  0.1463  0.2564
```

A retenção parece ser muito baixa, e pelo gráfico podemos ver que ela é denteada nos primeiros anos. Uma razão para isso estar ocorrendo é que o mandato de um representante dura dois anos, e o dos senadores é de seis, mas o conjunto de dados só contém registros do início de novos mandatos; logo, não temos dados dos anos em que um legislador ainda ocupava o gabinete, porém não tinha iniciado um novo mandato. Medir a retenção a cada ano é incorreto nesse caso. Uma opção seria medir a retenção somente em um ciclo de dois ou seis anos, mas também há outra estratégia que podemos empregar para obter os dados "ausentes". Vou abordá-la a seguir antes de voltar ao tópico de formação de grupos de coorte.

Ajustando a série temporal para aumentar a exatidão da retenção

Discutimos técnicas para a limpeza de dados "ausentes" no Capítulo 2, e voltaremos a essas técnicas nesta seção para chegar a uma curva de retenção mais branda e confiável para os legisladores. No trabalho com dados de séries temporais, como na análise de coorte, é importante considerar não só os dados que estão presentes, mas também se esses dados refletem com exatidão a presença ou a ausência de entidades em cada período de tempo. Isso é um problema principalmente em contextos em que um evento capturado nos dados leva a entidade a persistir por algum período de tempo que não é capturado neles. Por exemplo, um cliente comprando a assinatura de um software é representado nos dados na hora da transação, mas usará o software durante meses ou anos e não será necessariamente representado nos dados durante esse período. Para corrigir isso, precisamos de uma maneira de obter o período de tempo durante o qual a entidade continua presente, com uma data final explícita ou com o conhecimento da extensão da assinatura ou do mandato. Assim poderemos dizer que a entidade estava presente em qualquer data entre as datas inicial e final.

No conjunto de dados de legisladores, temos um registro para a data inicial de um mandato, mas não temos a informação de que isso "autoriza" um legislador a prestar serviços por dois ou seis anos, dependendo da câmara. Para resolver esse problema e abrandar a curva precisamos fornecer os valores "ausentes" referentes aos anos durante os quais os legisladores permanecerão no gabinete entre novos mandatos. Já que esse conjunto de dados inclui um valor final para cada mandato (`term_end`), mostrarei como criar uma análise de retenção mais precisa fornecendo datas entre os valores de início e fim. Em seguida, mostrarei como você pode atribuir datas finais quando o conjunto de dados não incluir uma data final.

Calcular a retenção usando uma data inicial e uma data final definidas nos dados é a abordagem mais precisa. Nos exemplos a seguir, consideraremos os legisladores retidos em um ano específico se eles ainda estiverem no gabinete a partir do último dia do ano, 31 de dezembro. Antes da Décima Segunda Emenda da Constituição dos EUA, os mandatos começavam em 4 de março, mas depois a data de início passou para 3 de

janeiro, ou para um dia útil subsequente se o terceiro dia cair em um fim de semana. Os legisladores podem tomar posse em outros dias do ano devido a eleições especiais fora de época ou a compromissos para assumir cargos vagos. Como resultado, as datas de `term_start` ocorrem em grande parte em janeiro, mas também se espalham ao longo do ano. Embora pudéssemos escolher outro dia, 31 de dezembro é uma estratégia para a normalização dessas diferentes datas iniciais.

A primeira etapa é criar um conjunto de dados contendo um registro para cada dia 31 de dezembro em que cada legislador continuava em seu gabinete. Isso pode ser feito pela junção (com *JOIN*) da subconsulta que encontrou `first_term` com a tabela `legislators_terms` para acharmos `term_start` e `term_end` para cada mandato. Uma segunda junção com `date_dim` recupera as datas que se encontram entre as datas de início e fim, restringindo os valores retornados a `c.month_name = 'December' and c.day_of_month = 31`. O período é calculado como os anos existentes entre a data de `date_dim` e `first_term`. Observe que, mesmo que se passarem 11 meses entre a tomada de posse em janeiro e a data de 31 de dezembro, o primeiro ano continua aparecendo como 0:

```
SELECT a.id_bioguide, a.first_term
,b.term_start, b.term_end
,c.date
,date_part('year',age(c.date,a.first_term)) as period
FROM
(
    SELECT id_bioguide, min(term_start) as first_term
    FROM legislators_terms
    GROUP BY 1
) a
JOIN legislators_terms b on a.id_bioguide = b.id_bioguide
LEFT JOIN date_dim c on c.date between b.term_start and b.term_end
and c.month_name = 'December' and c.day_of_month = 31
;
id_bioguide  first_term  term_start  term_end    date        period
-----------  ----------  ----------  ----------  ----------  ------
B000944      1993-01-05  1993-01-05  1995-01-03  1993-12-31  0.0
B000944      1993-01-05  1993-01-05  1995-01-03  1994-12-31  1.0
```

```
C000127      1993-01-05  1993-01-05  1995-01-03  1993-12-31  0.0
...          ...         ...         ...         ...         ...
```

Se não houver uma dimensão de data disponível, você pode criar uma subconsulta com as datas necessárias de algumas maneiras. Se seu banco de dados suportar a função `generate_series`, será possível criar uma subconsulta que retorne as datas desejadas:

```
SELECT generate_series::date as date
FROM generate_series('1770-12-31','2020-12-31',interval '1 year')
```

Se quiser, você pode salvar isso como uma tabela ou view para uso posterior. Alternativamente, pode consultar o conjunto de dados ou qualquer outra tabela do banco de dados que tenha um conjunto de datas completo. Nesse caso, a tabela tem todos os anos necessários, mas criaremos uma data 31 de dezembro para cada ano usando a função `make_date`:

```
SELECT distinct
make_date(date_part('year',term_start)::int,12,31)
FROM legislators_terms
```

Há várias maneiras criativas de obter a série de datas necessária. Use o método que estiver disponível e que for mais simples dentro de suas consultas.

Agora temos uma linha para cada `date` (fim do ano) para a qual quisermos calcular a retenção. A próxima etapa é calcular a `cohort_retained` de cada período, o que é feito com uma contagem de `id_bioguide`. Uma função `coalesce` é usada em `period` para definir um valor padrão igual a 0 quando da ocorrência de nulo. Este código manipula os casos em que o mandato de um legislador começa e termina no mesmo ano, reconhecendo os serviços prestados nesse ano:

```
SELECT
coalesce(date_part('year',age(c.date,a.first_term)),0) as period
,count(distinct a.id_bioguide) as cohort_retained
FROM
(
    SELECT id_bioguide, min(term_start) as first_term
    FROM legislators_terms
    GROUP BY 1
) a
JOIN legislators_terms b on a.id_bioguide = b.id_bioguide
LEFT JOIN date_dim c on c.date between b.term_start and b.term_end
```

```
and c.month_name = 'December' and c.day_of_month = 31
GROUP BY 1
;
period  cohort_retained
------  ---------------
0.0     12518
1.0     12328
2.0     8166
...     ...
```

A última etapa é calcular `cohort_size` e `pct_retained` como fizemos anteriormente usando funções de janela `first_value`:

```
SELECT period
,first_value(cohort_retained) over (order by period) as cohort_size
,cohort_retained
,cohort_retained * 1.0 /
first_value(cohort_retained) over (order by period) as pct_retained
FROM
(
    SELECT coalesce(date_part('year',age(c.date,a.first_term)),0) as
  period
    ,count(distinct a.id_bioguide) as cohort_retained
    FROM
    (
        SELECT id_bioguide, min(term_start) as first_term
        FROM legislators_terms
        GROUP BY 1
    ) a
    JOIN legislators_terms b on a.id_bioguide = b.id_bioguide
    LEFT JOIN date_dim c on c.date between b.term_start and
  b.term_end
    and c.month_name = 'December' and c.day_of_month = 31
    GROUP BY 1
) aa
;
period  cohort_size  cohort_retained  pct_retained
------  -----------  ---------------  ------------
0.0     12518        12518            1.0000
```

```
1.0     12518       12328           0.9848
2.0     12518       8166            0.6523
...     ...         ...             ...
```

Agora os resultados, representados em gráfico na Figura 4.4, são muito mais exatos. Quase todos os legisladores continuam no gabinete no ano 1, e a primeira grande diminuição ocorre no ano 2, quando alguns representantes não foram reeleitos.

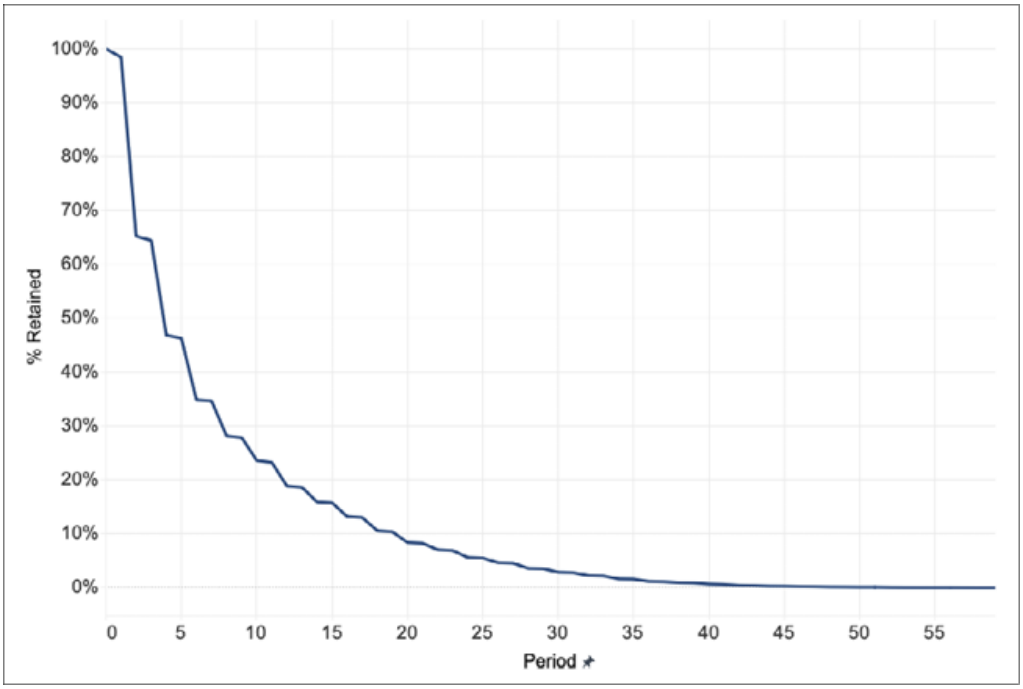


Figura 4.4: Retenção de legisladores após o ajuste dos anos reais de permanência no gabinete.

Se o conjunto de dados não tiver uma data final, há algumas opções para a sua imputação. Uma opção seria adicionar um intervalo fixo à data inicial, quando a extensão da assinatura ou do mandato for conhecida. Isso pode ser feito com a matemática de datas pela inclusão de um intervalo constante em `term_start`. Aqui, uma instrução CASE manipula a inclusão para os dois tipos de mandato (`term_type`):

```
SELECT a.id_bioguide, a.first_term
,b.term_start
,case when b.term_type = 'rep' then b.term_start + interval '2
  years'
      when b.term_type = 'sen' then b.term_start + interval '6
```

```
  years'
       end as term_end
FROM
(
    SELECT id_bioguide, min(term_start) as first_term
    FROM legislators_terms
    GROUP BY 1
) a
JOIN legislators_terms b on a.id_bioguide = b.id_bioguide
;
id_bioguide  first_term  term_start  term_end
-----------  ----------  ----------  ----------
B000944      1993-01-05  1993-01-05  1995-01-05
C000127      1993-01-05  1993-01-05  1995-01-05
C000141      1987-01-06  1987-01-06  1989-01-06
...          ...         ...         ...
```

Esse bloco de código pode então ser anexado ao código de retenção para derivar `period` e `pct_retained`. A desvantagem desse método é que ele não captura as situações em que um legislador não concluiu um mandato inteiro, o que pode ocorrer em caso de morte ou promoção para um cargo mais alto.

Uma segunda opção seria usar a data inicial subsequente, menos um dia, como a data de `term_end`. Isso pode ser calculado com a função de janela `lead`. Essa função é semelhante à função `lag` que usamos anteriormente, mas em vez de retornar um valor de uma linha anterior da partição, retorna um valor de uma linha posterior, como determinado na cláusula *ORDER BY*. O padrão é uma linha, que usaremos aqui, mas a função tem um argumento opcional que indica um número de linhas diferente. Em nosso exemplo estamos encontrando a data de `term_start` do mandato subsequente usando `lead` e depois subtraindo o intervalo `'1 day'` para derivar `term_end`:

```
SELECT a.id_bioguide, a.first_term
,b.term_start
,lead(b.term_start) over (partition by a.id_bioguide
                          order by b.term_start)
- interval '1 day' as term_end
```

```
FROM
(
    SELECT id_bioguide, min(term_start) as first_term
    FROM legislators_terms
    GROUP BY 1
) a
JOIN legislators_terms b on a.id_bioguide = b.id_bioguide
;
id_bioguide  first_term  term_start  term_end
-----------  ----------  ----------  ----------
A000001      1951-01-03  1951-01-03  (null)
A000002      1947-01-03  1947-01-03  1949-01-02
A000002      1947-01-03  1949-01-03  1951-01-02
...          ...         ...         ...
```

Esse bloco de código já pode ser anexado ao código de retenção. Esse método apresenta duas desvantagens. A primeira é que, quando não houver mandato subsequente, a função `lead` retornará nulo, deixando esse mandato sem um `term_end`. Um valor padrão, como o intervalo padrão mostrado no último exemplo, poderia ser usado nesses casos. A segunda desvantagem é que esse método presume que os mandatos sejam sempre consecutivos, sem intervalos de permanência fora do gabinete. Embora a maioria dos legisladores tenda a prestar serviços continuamente até suas carreiras parlamentares terminarem, certamente existem exemplos de lacunas entre mandatos que se estendem por vários anos.

Sempre que fizermos ajustes para fornecer dados ausentes, teremos de tomar cuidado com as suposições imaginadas. Em contextos baseados em assinatura ou em mandato, datas de início e fim explícitas tendem a ser mais precisas. Qualquer um dos outros dois métodos mostrados – adicionar um intervalo fixo ou definir a data final de acordo com a próxima data inicial – pode ser usado quando não houver uma data final presente e tivermos uma expectativa razoável de que a maioria dos clientes ou usuários permanecerá pela duração presumida.

Agora que vimos como calcular uma curva de retenção básica e fornecer datas ausentes, podemos começar a adicionar grupos de coorte. Comparar a retenção entre diferentes grupos é uma das principais razões para a

execução de uma análise de coorte. A seguir, discutiremos a formação de grupos a partir da própria série temporal e, em seguida, examinaremos a formação de grupos de coorte a partir dos dados de outras tabelas.

Coortes derivadas da própria série temporal

Já temos o código SQL para calcular a retenção, logo podemos começar a dividir as entidades em coortes. Nesta seção, mostrarei como derivar agrupamentos de coorte a partir da própria série temporal. Primeiro, discutirei coortes baseadas em tempo (time-based cohorts), que levam em consideração a primeira data, e explicarei como criar coortes baseadas em outros atributos da série temporal.

A maneira mais comum de criar coortes é baseando-as na primeira data ou hora, ou na data ou hora mínima, na qual a entidade aparece na série temporal. Isso significa que uma única tabela será necessária para a análise de retenção da coorte: a da própria série temporal. Criar a coorte pela primeira ocorrência ou ação é interessante porque geralmente grupos que começam em momentos diferentes se comportam de maneira distinta. No caso de serviços destinados aos consumidores, os primeiros usuários costumam ficar mais entusiasmados e apresentam uma fidelidade de uso diferente da de usuários posteriores, enquanto, no caso de softwares SaaS, os usuários posteriores demonstram uma permanência maior porque o produto está mais maduro. As coortes baseadas em tempo podem ser agrupadas por qualquer granularidade de tempo que seja significativa para a organização, embora coortes semanais, mensais ou anuais sejam comuns. Se não tiver certeza do agrupamento que deve usar, tente executar a análise de coorte com diferentes agrupamentos, sem deixar que os tamanhos das coortes fiquem pequenos demais, para ver onde surgem padrões relevantes. Felizmente, uma vez que você souber como construir as coortes e a análise de retenção, será fácil fazer a substituição por diferentes granularidades de tempo.

O primeiro exemplo usará coortes anuais e, em seguida, demonstrarei a troca para séculos. A principal questão a ser considerada é se a era em que um legislador assumiu o cargo pela primeira vez tem alguma correlação com sua retenção. Tendências políticas e a inclinação popular mudam com o passar do tempo, mas até que ponto?

Para calcular coortes anuais, em primeiro lugar adicionaremos o ano do `first_term` calculado anteriormente à consulta que encontra `period` e `cohort_retained`:

```
SELECT date_part('year',a.first_term) as first_year
,coalesce(date_part('year',age(c.date,a.first_term)),0) as period
,count(distinct a.id_bioguide) as cohort_retained
FROM
(
    SELECT id_bioguide, min(term_start) as first_term
    FROM legislators_terms
    GROUP BY 1
) a
JOIN legislators_terms b on a.id_bioguide = b.id_bioguide
LEFT JOIN date_dim c on c.date between b.term_start and b.term_end
and c.month_name = 'December' and c.day_of_month = 31
GROUP BY 1,2
;
first_year  period  cohort_retained
----------  ------  ---------------
1789.0      0.0     89
1789.0      2.0     89
1789.0      3.0     57
...         ...     ...
```

Essa consulta será então usada como subconsulta, e `cohort_size` e `pct_retained` serão calculados na consulta externa como anteriormente. Nesse caso, entretanto, precisamos de uma cláusula *PARTITION BY* que inclua o primeiro ano (`first_year`) para que `first_value` seja calculada apenas dentro do conjunto de linhas desse primeiro ano, em vez de ao longo de todo o conjunto de resultados da subconsulta:

```
SELECT first_year, period
,first_value(cohort_retained) over (partition by first_year
                                    order by period) as cohort_size
,cohort_retained
,cohort_retained /
first_value(cohort_retained) over (partition by first_year
                                    order by period) as pct_retained
```

```
FROM
(
    SELECT date_part('year',a.first_term) as first_year
    ,coalesce(date_part('year',age(c.date,a.first_term)),0) as
  period
    ,count(distinct a.id_bioguide) as cohort_retained
    FROM
    (
        SELECT id_bioguide, min(term_start) as first_term
        FROM legislators_terms
        GROUP BY 1
    ) a
    JOIN legislators_terms b on a.id_bioguide = b.id_bioguide
    LEFT JOIN date_dim c on c.date between b.term_start and
  b.term_end
    and c.month_name = 'December' and c.day_of_month = 31
    GROUP BY 1,2
) aa
;
first_year  period  cohort_size  cohort_retained  pct_retained
----------  ------  -----------  ---------------  ------------
1789.0      0.0     89           89               1.0000
1789.0      2.0     89           89               1.0000
1789.0      3.0     89           57               0.6404
...         ...     ...          ...              ...
```

Esse conjunto de dados inclui mais de duzentos anos iniciais, um período muito grande para ser facilmente representado em gráfico ou examinado em uma tabela. A seguir, veremos um intervalo menos granular e a criação da coorte de legisladores de acordo com o século de `first_term`. Essa alteração pode ser feita com facilidade pela substituição entre `century` e `year` na função `date_part` da subconsulta de `aa`. Lembre-se de que os nomes dos séculos são diferentes dos anos que eles representam, de modo que o século 18 durou de 1700 a 1799, o século 19 vai de 1800 a 1899 e assim por diante. O particionamento da função `first_value` muda para o campo `first_century`:

```
SELECT first_century, period
```

```
,first_value(cohort_retained) over (partition by first_century
                                    order by period) as cohort_size
,cohort_retained
,cohort_retained /
first_value(cohort_retained) over (partition by first_century
                                    order by period) as pct_retained
FROM
(
    SELECT date_part('century',a.first_term) as first_century
    ,coalesce(date_part('year',age(c.date,a.first_term)),0) as
  period
    ,count(distinct a.id_bioguide) as cohort_retained
    FROM
    (
        SELECT id_bioguide, min(term_start) as first_term
        FROM legislators_terms
        GROUP BY 1
    ) a
    JOIN legislators_terms b on a.id_bioguide = b.id_bioguide
    LEFT JOIN date_dim c on c.date between b.term_start and
  b.term_end
    and c.month_name = 'December' and c.day_of_month = 31
    GROUP BY 1,2
) aa
ORDER BY 1,2
;
first_century  period  cohort_size  cohort_retained  pct_retained
-------------  ------  -----------  ---------------  ------------
18.0           0.0     368          368              1.0000
18.0           1.0     368          360              0.9783
18.0           2.0     368          242              0.6576
...            ...     ...          ...              ...
```

Os resultados estão representados em gráfico na Figura 4.5. A retenção nos primeiros anos foi mais alta para os eleitos pela primeira vez no século 20 ou 21. Ainda estamos no século 21 e, portanto, muitos desses legisladores não tiveram a oportunidade de permanecer no mandato por cinco anos ou mais, embora também tenham sido incluídos no denominador.

Poderíamos considerar remover o século 21 da análise, mas deixei-o para demonstrar como a curva de retenção cai artificialmente devido a essa circunstância.

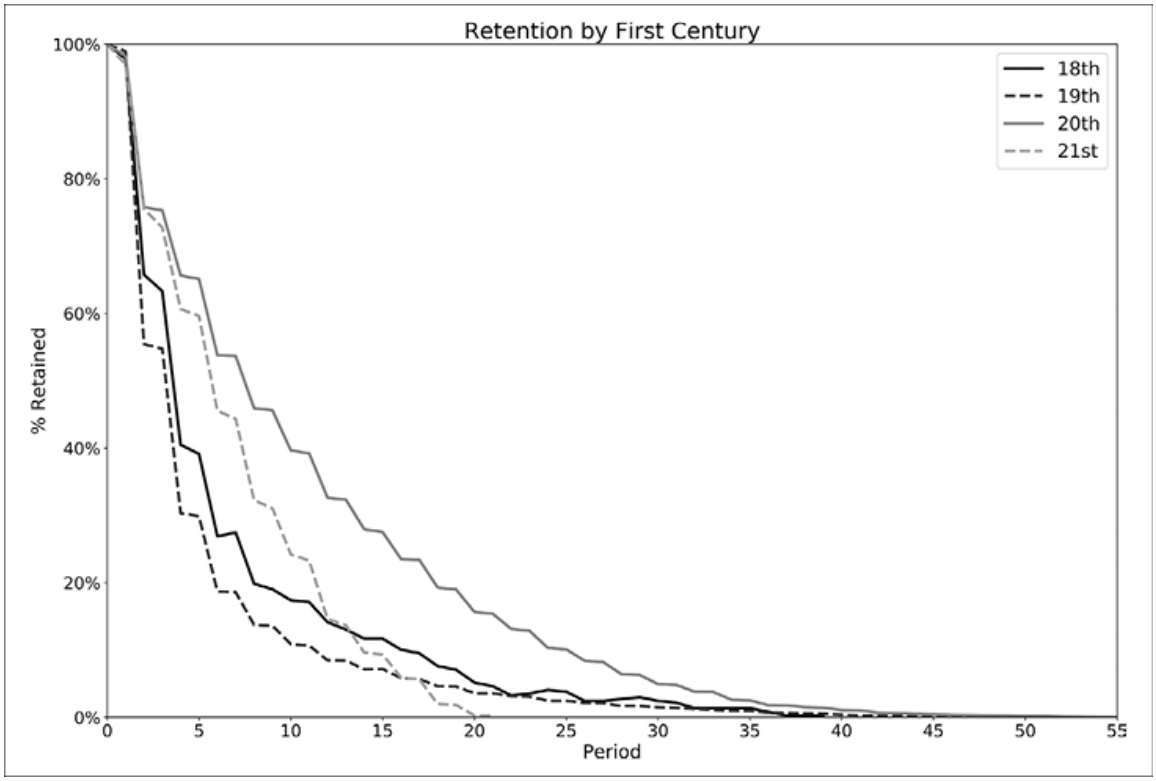


Figura 4.5: A retenção de legisladores pelo século em que o primeiro mandato começou.

As coortes podem ser definidas a partir de outros atributos de uma série temporal além da primeira data, com as opções dependendo dos valores da tabela. A tabela `legislators_terms` tem um campo `state`, que indica o estado que a pessoa está representando nesse mandato. Podemos usá-lo para criar coortes, e as basearemos no primeiro estado para assegurar que qualquer pessoa que tenha representado vários estados apareça nos dados apenas uma vez.

Na criação de coortes por um atributo que pode mudar com o tempo, é importante assegurar que cada entidade receba apenas um valor. Caso contrário, a entidade pode ser representada em várias coortes, introduzindo tendências na análise. Geralmente, o valor do registro mais antigo do conjunto de dados é usado.

Para encontrar o primeiro estado de cada legislador, podemos usar a função de janela `first_value`. Neste exemplo, também transformaremos a

função `min` em uma função de janela para evitar uma cláusula *GROUP BY* muito longa:

```
SELECT distinct id_bioguide
,min(term_start) over (partition by id_bioguide) as first_term
,first_value(state) over (partition by id_bioguide
                          order by term_start) as first_state
FROM legislators_terms
;
id_bioguide  first_term  first_state
-----------  ----------  -----------
C000001      1893-08-07  GA
R000584      2009-01-06  ID
W000215      1975-01-14  CA
...          ...         ...
```

Agora podemos anexar esse código ao código de retenção para encontrar a retenção por `first_state`:

```
SELECT first_state, period
,first_value(cohort_retained) over (partition by first_state
                                    order by period) as cohort_size
,cohort_retained
,cohort_retained /
first_value(cohort_retained) over (partition by first_state
                                    order by period) as pct_retained
FROM
(
    SELECT a.first_state
    ,coalesce(date_part('year',age(c.date,a.first_term)),0) as
  period
    ,count(distinct a.id_bioguide) as cohort_retained
    FROM
    (
        SELECT distinct id_bioguide
        ,min(term_start) over (partition by id_bioguide) as
  first_term
        ,first_value(state) over (partition by id_bioguide order by
  term_start)
         as first_state
```

```
        FROM legislators_terms
    ) a
    JOIN legislators_terms b on a.id_bioguide = b.id_bioguide
    LEFT JOIN date_dim c on c.date between b.term_start and
  b.term_end
    and c.month_name = 'December' and c.day_of_month = 31
    GROUP BY 1,2
) aa
;
first_state  period  cohort_size  cohort_retained  pct_retained
-----------  ------  -----------  ---------------  ------------
AK           0.0     19           19               1.0000
AK           1.0     19           19               1.0000
AK           2.0     19           15               0.7895
...          ...     ...          ...              ...
```

As curvas de retenção dos cinco estados com o número total de legisladores mais alto estão representadas em gráfico na Figura 4.6. Os eleitos em Illinois e Massachusetts têm a retenção mais alta, enquanto os de Nova York têm a retenção mais baixa. Determinar as razões seria uma ramificação interessante dessa análise.

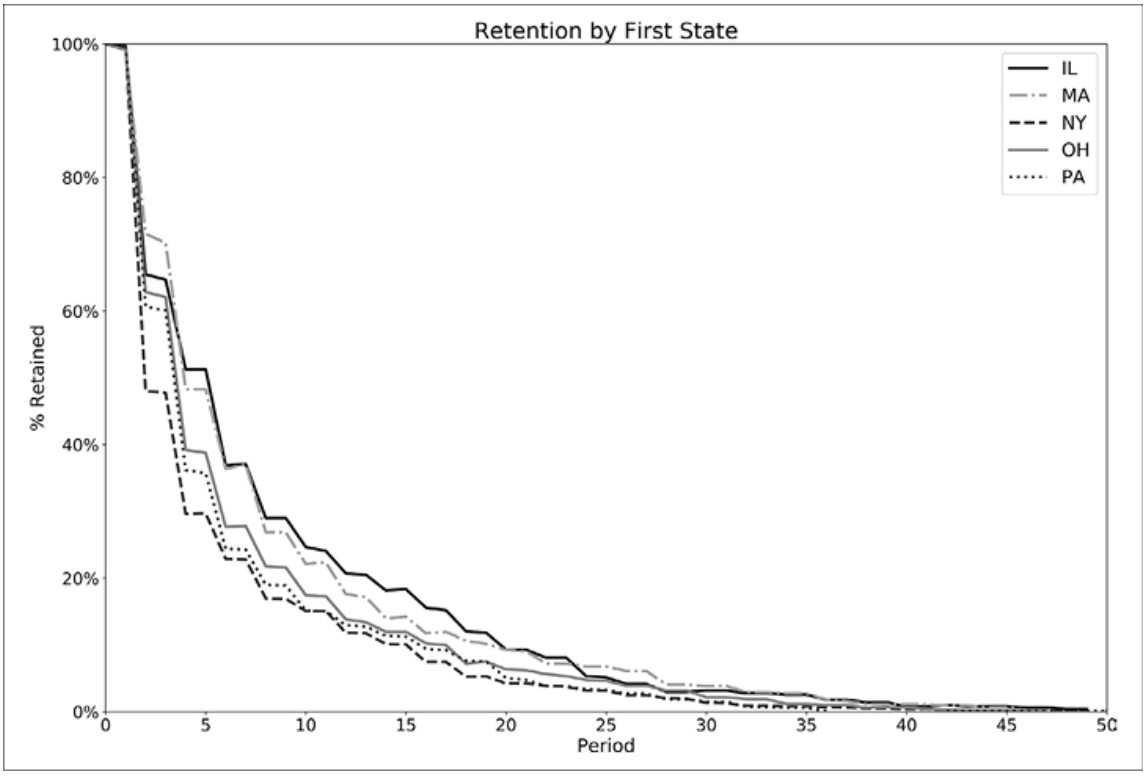


Figura 4.6: Retenção de legisladores pelo primeiro estado: os cinco principais estados por total de legisladores.

Definir coortes a partir da série temporal é relativamente simples com o uso de uma data mínima para cada entidade e a conversão dessa data em um mês, ano ou século conforme apropriado para a análise. Alternar-se entre mês e ano ou outros níveis de granularidade também é fácil, permitindo que várias opções sejam testadas para a busca de um agrupamento que seja significativo para a organização. Outros atributos podem ser usados na criação da coorte com a função de janela `first_value`. A seguir, examinaremos casos em que o atributo de criação da coorte vem de uma tabela que não é a da série temporal.

Definindo a coorte a partir de uma tabela separada

Com frequência as características que definem uma coorte estão em uma tabela diferente da que contém a série temporal. Por exemplo, um banco de dados poderia ter uma tabela de clientes com informações como a fonte de captação ou a data de registro de acordo com as quais os clientes poderiam ser classificados em coortes.

Neste exemplo, consideraremos se o gênero do legislador tem algum

impacto sobre sua retenção. A tabela `legislators` tem um campo `gender` (gênero), no qual F significa feminino e M masculino, que podemos usar para criar a coorte de legisladores. Para fazê-lo, executaremos a junção da tabela `legislators` como alias `d` para adicionar `gender` ao cálculo de `cohort_retained`, em vez de usarmos o ano ou o século:

```
SELECT d.gender
,coalesce(date_part('year',age(c.date,a.first_term)),0) as period
,count(distinct a.id_bioguide) as cohort_retained
FROM
(
    SELECT id_bioguide, min(term_start) as first_term
    FROM legislators_terms
    GROUP BY 1
) a
JOIN legislators_terms b on a.id_bioguide = b.id_bioguide
LEFT JOIN date_dim c on c.date between b.term_start and b.term_end
and c.month_name = 'December' and c.day_of_month = 31
JOIN legislators d on a.id_bioguide = d.id_bioguide
GROUP BY 1,2
;
gender  period  cohort_retained
------  ------  ---------------
F       0.0     366
M       0.0     12152
F       1.0     349
M       1.0     11979
...     ...     ...
```

Fica imediatamente claro que mais pessoas do sexo masculino, e não do sexo feminino, tiveram mandatos legislativos. Agora podemos calcular `percent_retained` para comparar a retenção desses grupos:

```
SELECT gender, period
,first_value(cohort_retained) over (partition by gender
                                    order by period) as cohort_size
,cohort_retained
,cohort_retained/
first_value(cohort_retained) over (partition by gender
```

```
                                        order by period) as pct_retained
FROM
(
    SELECT d.gender
    ,coalesce(date_part('year',age(c.date,a.first_term)),0) as
  period
    ,count(distinct a.id_bioguide) as cohort_retained
    FROM
    (
        SELECT id_bioguide, min(term_start) as first_term
        FROM legislators_terms
        GROUP BY 1
    ) a
    JOIN legislators_terms b on a.id_bioguide = b.id_bioguide
    LEFT JOIN date_dim c on c.date between b.term_start and
  b.term_end
    and c.month_name = 'December' and c.day_of_month = 31
    JOIN legislators d on a.id_bioguide = d.id_bioguide
    GROUP BY 1,2
) aa
;
gender  period  cohort_size  cohort_retained  pct_retained
------  ------  -----------  ---------------  ------------
F       0.0     366          366              1.0000
M       0.0     12152        12152            1.0000
F       1.0     366          349              0.9536
M       1.0     12152        11979            0.9858
...     ...     ...          ...              ...
```

Podemos ver pelos resultados representados em gráfico na Figura 4.7 que nos períodos 2 a 29 a retenção é mais alta para as legisladoras femininas do que para seus colegas masculinos. A primeira legisladora feminina não assumiu o mandato antes de 1917, quando Jeannette Rankin ingressou no Congresso como representante republicana de Montana. Como vimos anteriormente, a retenção tem aumentado em séculos mais recentes.

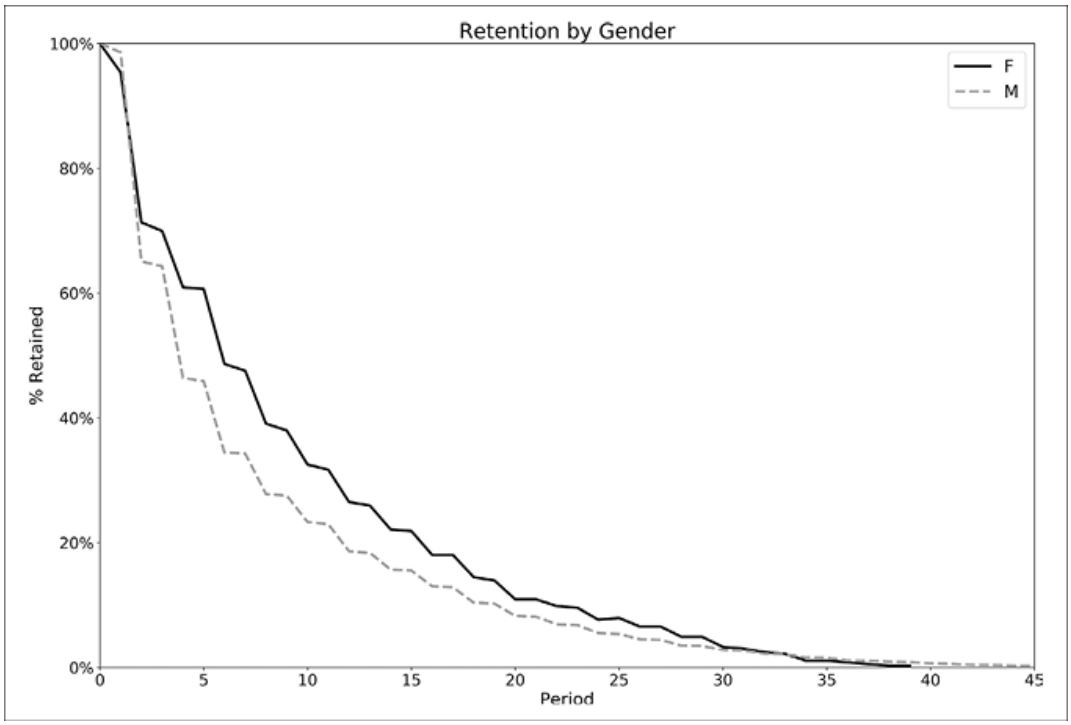


Figura 4.7: Retenção de legisladores por gênero.

Para fazer uma comparação mais justa, poderíamos restringir a inclusão na análise apenas aos legisladores cujo `first_term` teve início desde que mulheres começaram a ser aceitas no Congresso. Podemos fazer isso adicionando um filtro *WHERE* à subconsulta de `aa`. Aqui os resultados também ficarão restritos a mandatos iniciados antes de 2000, para assegurarmos que as coortes contemplem a possibilidade de pelo menos 20 anos de permanência no cargo:

```
SELECT gender, period
,first_value(cohort_retained) over (partition by gender
                                    order by period) as cohort_size
,cohort_retained
,cohort_retained /
first_value(cohort_retained) over (partition by gender
                                    order by period) as pct_retained
FROM
(
    SELECT d.gender
    ,coalesce(date_part('year',age(c.date,a.first_term)),0) as
  period
```

```
    ,count(distinct a.id_bioguide) as cohort_retained
    FROM
    (
        SELECT id_bioguide, min(term_start) as first_term
        FROM legislators_terms
        GROUP BY 1
    ) a
    JOIN legislators_terms b on a.id_bioguide = b.id_bioguide
    LEFT JOIN date_dim c on c.date between b.term_start and
  b.term_end
    and c.month_name = 'December' and c.day_of_month = 31
    JOIN legislators d on a.id_bioguide = d.id_bioguide
    WHERE a.first_term between '1917-01-01' and '1999-12-31'
    GROUP BY 1,2
) aa
;
gender  period  cohort_size  cohort_retained  pct_retained
------  ------  -----------  ---------------  ------------
F       0.0     200          200              1.0000
M       0.0     3833         3833             1.0000
F       1.0     200          187              0.9350
M       1.0     3833         3769             0.9833
...     ...     ...          ...              ...
```

Os legisladores masculinos ainda superam em número as legisladoras femininas, mas por uma margem menor. A retenção das coortes está representada em gráfico na Figura 4.8. Com as coortes revisadas, os legisladores masculinos têm retenção mais alta até o ano 7, mas a partir do ano 12 as legisladoras femininas têm retenção mais alta. A diferença entre as duas análises de coorte baseadas em gênero destaca a importância da definição de coortes apropriadas e de assegurar que elas tenham períodos de tempo comparáveis para a demanda atual ou a conclusão de outras ações de interesse. Para melhorar ainda mais essa análise, poderíamos criar uma coorte tanto pelo ano ou década de início *quanto* pelo gênero, a fim de acompanhar alterações adicionais na retenção durante o século 20 e no século 21.

As coortes podem ser definidas de várias maneiras, a partir da série

temporal ou de outras tabelas. Com a estrutura que desenvolvemos, subconsultas, views ou outras tabelas derivadas podem ser introduzidas, trazendo um conjunto de cálculos totalmente novo para ser a base de uma coorte. Muitos critérios, como o ano inicial e o gênero, podem ser usados. Um cuidado que precisamos ter ao dividir populações em coortes com base em múltiplos critérios é que isso pode levar a coortes esparsas, em que alguns dos grupos definidos sejam pequenos demais e não tenham representação no conjunto de dados para todos os períodos de tempo. A próxima seção discutirá métodos para a superação desse desafio.

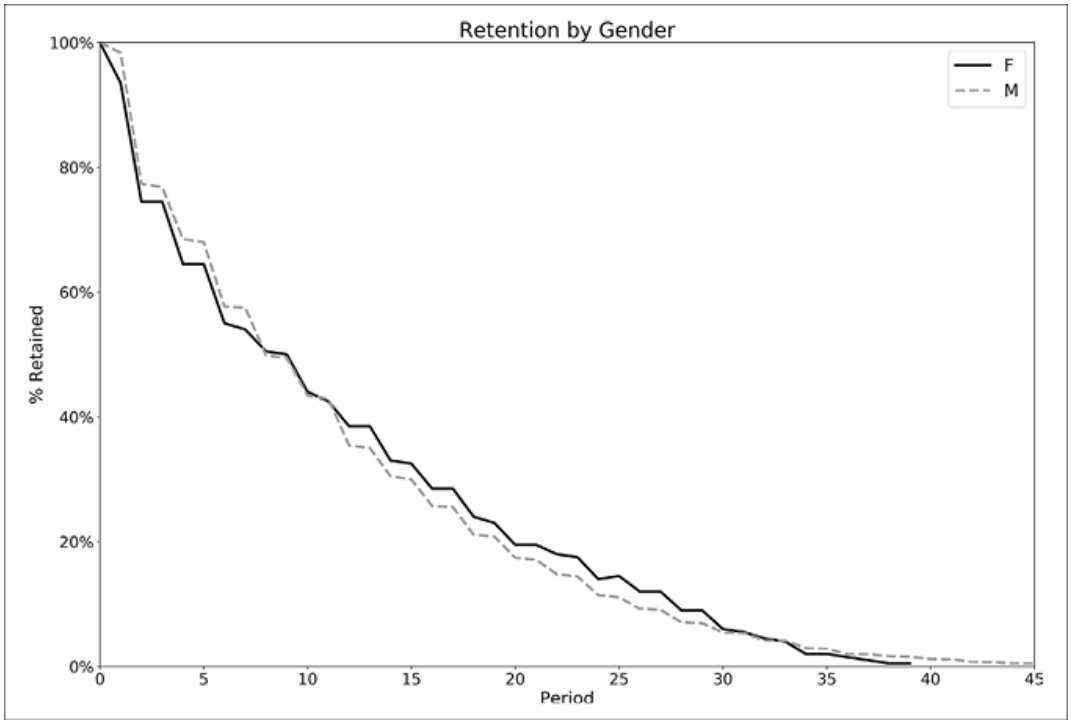


Figura 4.8: Retenção de legisladores por gênero: coortes de 1917 a 1999.

Lidando com coortes esparsas

Em um conjunto de dados ideal, cada coorte tem alguma ação ou registro na série temporal para cada período de interesse. Já vimos que datas "ausentes" podem ocorrer devido a assinaturas ou mandatos que se estendem por vários períodos, e examinamos como corrigir isso usando uma dimensão de data para inferir datas intermediárias. Outro problema pode surgir quando, devido a critérios de agrupamento, a coorte torna-se pequena demais e, como resultado, é representada apenas

esporadicamente nos dados. Uma coorte pode desaparecer do conjunto de resultados, quando na verdade preferiríamos que ela aparecesse com um valor de retenção zero. Esse problema chama-se *coortes esparsas*, e pode ser resolvido com o uso cuidadoso de *LEFT JOIN*s.

Para demonstrar isso, tentaremos criar a coorte de legisladoras femininas pelo primeiro estado que elas representaram para ver se há alguma diferença na retenção. Já vimos que existiam relativamente poucas legisladoras femininas. Se dividirmos sua coorte ainda mais usando o estado, é altamente provável que criemos algumas coortes esparsas nas quais haja muito poucos membros. Antes de fazer ajustes no código, adicionaremos `first_state` (calculado na seção sobre a derivação de coortes a partir da série temporal) ao exemplo em que incluímos o gênero e examinaremos os resultados:

```
SELECT first_state, gender, period
,first_value(cohort_retained) over (partition by first_state, gender
                                    order by period) as cohort_size
,cohort_retained
,cohort_retained /
first_value(cohort_retained) over (partition by first_state, gender
                                    order by period) as pct_retained
FROM
(
    SELECT a.first_state, d.gender
    ,coalesce(date_part('year',age(c.date,a.first_term)),0) as
  period
    ,count(distinct a.id_bioguide) as cohort_retained
    FROM
    (
        SELECT distinct id_bioguide
        ,min(term_start) over (partition by id_bioguide) as
  first_term
        ,first_value(state) over (partition by id_bioguide
                                  order by term_start) as
  first_state
        FROM legislators_terms
    ) a
```

```
    JOIN legislators_terms b on a.id_bioguide = b.id_bioguide
    LEFT JOIN date_dim c on c.date between b.term_start and
  b.term_end
    and c.month_name = 'December' and c.day_of_month = 31
    JOIN legislators d on a.id_bioguide = d.id_bioguide
    WHERE a.first_term between '1917-01-01' and '1999-12-31'
    GROUP BY 1,2,3
) aa
;
first_state  gender  period  cohort_size  cohort_retained
  pct_retained
-----------  ------  ------  -----------  --------------
  --
AZ           F       0.0     2            2               1.0000
AZ           M       0.0     26           26              1.0000
AZ           F       1.0     2            2               1.0000
...          ...     ...     ...          ...             ...
```

Representar os resultados dos primeiros 20 períodos em gráfico, como na Figura 4.9, revela as coortes esparsas. O Alasca não teve nenhuma legisladora feminina, enquanto a curva de retenção do gênero feminino do Arizona desaparece após o ano 3. Só a Califórnia, um estado grande com muitos legisladores, tem curvas de retenção completas para os dois gêneros. Esse padrão se repete para outros estados pequenos e grandes.

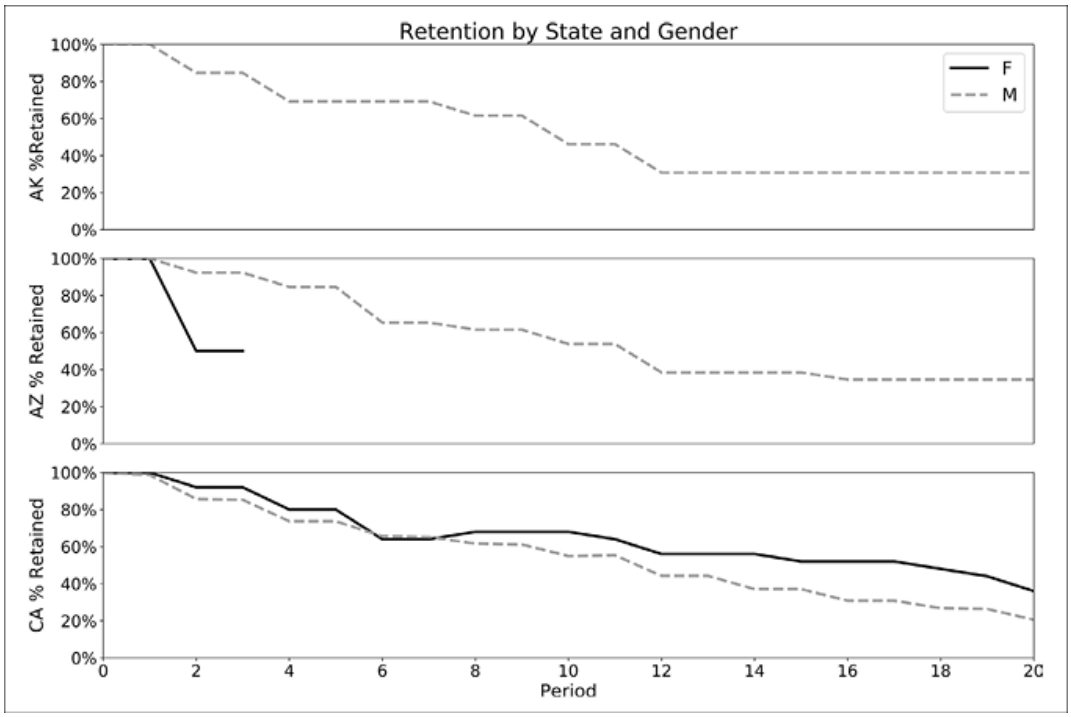


Figura 4.9: Retenção de legisladores por gênero e primeiro estado.

Agora examinaremos como assegurar que haja um registro para cada período a fim de que a consulta retorne valores iguais a zero para a retenção em vez de nulos. A primeira etapa é consultar todas as combinações de períodos, nesse caso de `first_state` e `gender`, com o tamanho de coorte inicial de cada combinação. Isso pode ser feito com a junção da subconsulta de `aa`, que calcula a coorte, com uma subconsulta `generate_series` que retorne todos os inteiros de 0 a 20, com o critério `on 1 = 1`. Esta é uma maneira prática de forçar uma *JOIN* Cartesiana quando as duas subconsultas não tiverem nenhum campo em comum:

```
SELECT aa.gender, aa.first_state, cc.period, aa.cohort_size
FROM
(
    SELECT b.gender, a.first_state
    ,count(distinct a.id_bioguide) as cohort_size
    FROM
    (
        SELECT distinct id_bioguide
        ,min(term_start) over (partition by id_bioguide) as
  first_term
```

```
        ,first_value(state) over (partition by id_bioguide
                                  order by term_start) as
  first_state
        FROM legislators_terms
    ) a
    JOIN legislators b on a.id_bioguide = b.id_bioguide
    WHERE a.first_term between '1917-01-01' and '1999-12-31'
    GROUP BY 1,2
) aa
JOIN
(
    SELECT generate_series as period
    FROM generate_series(0,20,1)
) cc on 1 = 1
;
gender  state  period  cohort
------  -----  ------  ------
F       AL     0       3
F       AL     1       3
F       AL     2       3
...     ...    ...     ...
```

A próxima etapa é fazer a junção de volta para os períodos reais do mandato, com uma *LEFT JOIN* para assegurar que todos os períodos de tempo permaneçam no resultado final:

```
SELECT aaa.gender, aaa.first_state, aaa.period, aaa.cohort_size
,coalesce(ddd.cohort_retained,0) as cohort_retained
,coalesce(ddd.cohort_retained,0) / aaa.cohort_size as pct_retained
FROM
(
    SELECT aa.gender, aa.first_state, cc.period, aa.cohort_size
    FROM
    (
        SELECT b.gender, a.first_state
        ,count(distinct a.id_bioguide) as cohort_size
        FROM
        (
            SELECT distinct id_bioguide
```

```
            ,min(term_start) over (partition by id_bioguide)
             as first_term
            ,first_value(state) over (partition by id_bioguide
                                      order by term_start)
                                      as first_state
            FROM legislators_terms
        ) a
        JOIN legislators b on a.id_bioguide = b.id_bioguide
        WHERE a.first_term between '1917-01-01' and '1999-12-31'
        GROUP BY 1,2
    ) aa
    JOIN
    (
        SELECT generate_series as period
        FROM generate_series(0,20,1)
    ) cc on 1 = 1
) aaa
LEFT JOIN
(
    SELECT d.first_state, g.gender
    ,coalesce(date_part('year',age(f.date,d.first_term)),0) as
  period
    ,count(distinct d.id_bioguide) as cohort_retained
    FROM
    (
        SELECT distinct id_bioguide
        ,min(term_start) over (partition by id_bioguide) as
  first_term
        ,first_value(state) over (partition by id_bioguide
                                  order by term_start) as
  first_state
        FROM legislators_terms
    ) d
    JOIN legislators_terms e on d.id_bioguide = e.id_bioguide
    LEFT JOIN date_dim f on f.date between e.term_start and
  e.term_end
     and f.month_name = 'December' and f.day_of_month = 31
    JOIN legislators g on d.id_bioguide = g.id_bioguide
```

```
    WHERE d.first_term between '1917-01-01' and '1999-12-31'
    GROUP BY 1,2,3
) ddd on aaa.gender = ddd.gender and aaa.first_state =
  ddd.first_state
and aaa.period = ddd.period
;
gender  first_state  period  cohort_size  cohort_retained
  pct_retained
------  -----------  ------  -----------  --------------  ----------
  --
F       AL           0       3            3               1.0000
F       AL           1       3            1               0.3333
F       AL           2       3            0               0.0000
...     ...          ...     ...          ...             ...
```

Agora podemos pivotar os resultados e confirmar se existe um valor para cada coorte em cada período:

```
gender  first_state  yr0    yr2     yr4     yr6     yr8      yr10
------  -----------  -----  ------  ------  ------  ------   ------
F       AL           1.000  0.0000  0.0000  0.0000  0.0000   0.0000
F       AR           1.000  0.8000  0.2000  0.4000  0.4000   0.4000
F       CA           1.000  0.9200  0.8000  0.6400  0.6800   0.6800
...     ...          ...    ...     ...     ...     ...      ...
```

Observe que a essa altura o código SQL já ficou bastante longo. Uma das partes mais difíceis da criação de SQL para a análise de retenção de coortes é manter a lógica simples e o código organizado, um tópico que discutirei melhor no Capítulo 8. Na construção de um código de retenção, acho útil avançar etapa a etapa, verificando os resultados no decorrer do processo. Também faço verificações aleatórias de coortes individuais a fim de validar se o resultado final é preciso.

As coortes podem ser definidas de várias maneiras. Até agora, normalizamos todas as nossas coortes baseando-as na primeira data em que elas aparecem nos dados da série temporal. Contudo, essa não é a única opção, e análises interessantes podem ser feitas a partir da metade do tempo de vida de uma entidade. Antes de concluir nosso trabalho sobre a análise de retenção, examinaremos essa maneira adicional de definir coortes.

Definindo coortes a partir de datas diferentes da primeira data

Geralmente as coortes baseadas em tempo são definidas a partir da primeira ocorrência da entidade na série temporal ou a partir de alguma outra data mais antiga, como a data de registro. No entanto, a criação de coortes baseadas em uma data diferente pode ser útil e reveladora. Por exemplo, poderíamos examinar a retenção de todos os clientes usuários de um serviço a partir de uma data específica. Esse tipo de análise pode ser usado para sabermos se alterações de produto ou marketing tiveram um impacto de longo prazo sobre os clientes existentes.

Ao usar uma data diferente da primeira, temos de tomar cuidado para definir precisamente os critérios de inclusão em cada coorte. Uma opção é selecionar as entidades presentes em uma data específica do calendário. É relativamente fácil inserir isso em um código SQL, mas pode ser problemático se uma grande parcela da população de usuários regulares não estiver presente todo dia, fazendo a retenção variar de acordo com o dia escolhido. Uma opção para a correção desse problema seria calcularmos a retenção para várias datas iniciais e então encontrarmos a média dos resultados.

Outra opção seria usar uma janela de tempo, como uma semana ou mês. Qualquer entidade que aparecer no conjunto de dados durante essa janela será incluída na coorte. Embora geralmente essa abordagem represente melhor o negócio ou o processo, a desvantagem é que o código SQL será mais complexo, e o tempo da consulta pode ser mais lento devido aos cálculos mais volumosos do banco de dados. Encontrar o equilíbrio certo entre o desempenho da consulta e a precisão dos resultados requer perícia.

Examinaremos como calcular a análise a partir de uma data intermediária com o conjunto de dados de legisladores, considerando a retenção de legisladores que estavam no cargo no ano 2000. Criaremos a coorte pelo `term_type`, que tem os valores “sen” para senadores e “rep” para representantes. A definição incluirá legisladores que estavam no mandato em qualquer momento durante o ano 2000: os que ingressaram antes cujos mandatos terminaram durante ou após o ano 2000 estarão qualificados, assim como aqueles que começaram um mandato em 2000.

Podemos embutir no código qualquer data de 2000 para ser o `first_term`, já que verificaremos posteriormente se os legisladores estavam no mandato em algum momento durante o ano 2000. A data inicial mínima (`min_start`) dos mandatos pertencentes a essa janela também será calculada para uso em uma etapa posterior:

```
SELECT distinct id_bioguide, term_type, date('2000-01-01') as
  first_term
,min(term_start) as min_start
FROM legislators_terms
WHERE term_start <= '2000-12-31' and term_end >= '2000-01-01'
GROUP BY 1,2,3
;
id_bioguide  term_type  first_term  min_start
-----------  ---------  ----------  ---------
C000858      sen        2000-01-01  1997-01-07
G000333      sen        2000-01-01  1995-01-04
M000350      rep        2000-01-01  1999-01-06
...          ...        ...         ...
```

Podemos então anexar esse código ao nosso código de retenção, com dois ajustes. Em primeiro lugar, um critério adicional de *JOIN* é incluído entre a subconsulta de `a` e a tabela `legislators_terms` para retornarmos somente mandatos iniciados na data de `min_start` ou depois dela. Em segundo lugar, um filtro adicional é incluído em `date_dim` para que só sejam retornadas datas de 2000 ou posteriores:

```
SELECT term_type, period
,first_value(cohort_retained) over (partition by term_type order by
  period)
as cohort_size
,cohort_retained
,cohort_retained /
first_value(cohort_retained) over (partition by term_type order by
  period)
as pct_retained
FROM
(
    SELECT a.term_type
```

```
   ,coalesce(date_part('year',age(c.date,a.first_term)),0) as
period
   ,count(distinct a.id_bioguide) as cohort_retained
   FROM
   (
       SELECT distinct id_bioguide, term_type
       ,date('2000-01-01') as first_term
       ,min(term_start) as min_start
       FROM legislators_terms
       WHERE term_start <= '2000-12-31' and term_end >= '2000-01-
01'
       GROUP BY 1,2,3
   ) a
   JOIN legislators_terms b on a.id_bioguide = b.id_bioguide
   and b.term_start >= a.min_start
   LEFT JOIN date_dim c on c.date between b.term_start and
b.term_end
   and c.month_name = 'December' and c.day_of_month = 31
   and c.year >= 2000
   GROUP BY 1,2
) aa
;
term_type  period  cohort_size  cohort_retained  pct_retained
---------  ------  -----------  ---------------  ------------
rep        0.0     440          440              1.0000
sen        0.0     101          101              1.0000
rep        1.0     440          392              0.8909
sen        1.0     101          89               0.8812
...        ...     ...          ...              ...
```

A Figura 4.10 mostra que, apesar dos mandatos mais longos dos senadores, a retenção entre as duas coortes foi semelhante, e na verdade foi pior para os senadores após 10 anos. Uma análise adicional comparando os diferentes anos em que os legisladores foram eleitos pela primeira vez, ou usando outros atributos da coorte, pode gerar alguns insights interessantes.

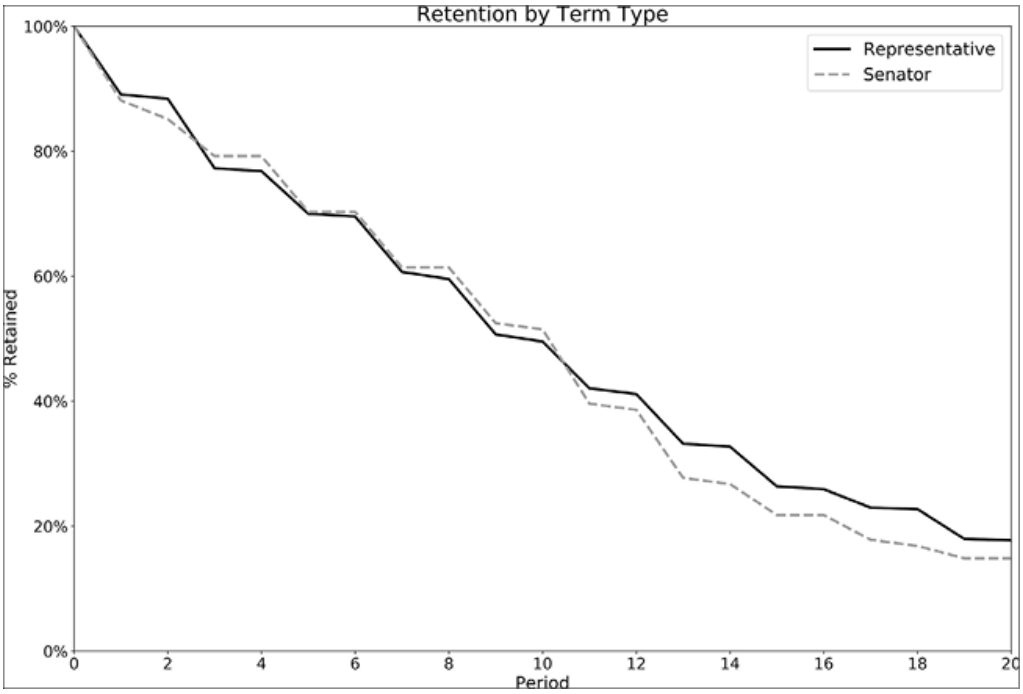


Figura 4.10: Retenção por tipo de mandato dos legisladores que estavam no cargo durante o ano 2000.

Um uso comum de coorte baseada em um valor diferente do valor inicial ocorre quando tentamos analisar a retenção após uma entidade ter alcançado um limite, como um número específico de compras ou determinado valor gasto. Como acontece com qualquer coorte, é importante tomar cuidado ao definir o que qualificará uma entidade para que esteja em uma coorte e a data que será usada como data inicial.

A retenção por coorte é uma maneira poderosa de entendermos o comportamento das entidades do conjunto de dados de uma série temporal. Vimos como calcular a retenção com SQL e como criar coortes baseadas na própria série temporal, em outras tabelas e em pontos intermediários do tempo de vida da entidade. Também examinamos como usar funções e *JOIN*s para ajustar datas dentro da série temporal e fazer a compensação no caso de coortes esparsas. Há vários tipos de análises que estão relacionadas à análise de retenção de coorte: sobrevivência, retorno e cálculos cumulativos, e todas se baseiam no código SQL que desenvolvemos para a retenção. Iremos examiná-las a seguir.

Análises de coorte relacionadas

Na última seção, aprendemos como escrever SQL para a análise de retenção de coorte. A retenção captura se uma entidade estava no conjunto de dados de uma série temporal em uma data ou janela de tempo específica. Além da presença em uma data específica, geralmente a análise se preocupa com questões como por quanto tempo uma entidade perdurou, se uma entidade executou várias ações, e quantas dessas ações ocorreram. Todas essas perguntas podem ser respondidas com um código semelhante ao da retenção e que esteja de acordo com o critério de coorte selecionado. Examinaremos o primeiro deles, a sobrevivência.

Sobrevivência

A *sobrevivência*, também chamada de análise de sobrevivência, se preocupa com perguntas relacionadas a quanto tempo algo durou, ou com o tempo que se passou até ocorrer um evento específico como a rotatividade ou o falecimento. A análise de sobrevivência pode responder a perguntas sobre a parcela da população que tem probabilidades de perdurar após um período de tempo específico. As coortes podem ajudar a identificar ou pelo menos fornecer hipóteses sobre as características ou circunstâncias que aumentam ou diminuem a probabilidade de sobrevivência.

Isso é semelhante a uma análise de retenção, mas em vez de calcular se uma entidade estava presente em um período específico, calculamos se ela está presente nesse período ou em um período posterior da série temporal. Em seguida, a parcela da coorte total é calculada. Normalmente um ou mais períodos são selecionados dependendo da natureza do conjunto de dados analisado. Por exemplo, se quisermos saber a parcela de participantes de um jogo que sobreviveram por uma semana ou mais, podemos procurar ações que ocorreram após uma semana a partir do início e considerar os jogadores que continuam. Por outro lado, se estivermos interessados em quantos alunos ainda estão na escola após um número de anos específico, podemos procurar a ausência de um evento de graduação em um conjunto de dados. O número de períodos pode ser selecionado (com *SELECT*) pelo cálculo de uma média ou do tempo de vida típico ou pela escolha de períodos de tempo que sejam significativos para a organização ou o processo analisado, como um mês, um ano ou um período mais longo.

Neste exemplo, examinaremos a parcela de legisladores que sobreviveram no cargo por uma década ou mais após seu primeiro mandato terminar. Já que não precisamos saber as datas específicas de cada mandato, podemos começar calculando a primeira e a última data de `term_start`, usando as agregações `min` e `max`:

```
SELECT id_bioguide
,min(term_start) as first_term
,max(term_start) as last_term
FROM legislators_terms
GROUP BY 1
;
id_bioguide  first_term  last_term
-----------  ----------  ---------
A000118      1975-01-14  1977-01-04
P000281      1933-03-09  1937-01-05
K000039      1933-03-09  1951-01-03
...          ...         ...
```

Em seguida, adicionaremos à consulta uma função `date_part` para encontrar o século da primeira data (`min`) de `term_start` e calcularemos a permanência (`tenure`) como o número de anos entre a primeira e a última data de `term_start` encontradas com a função `age`:

```
SELECT id_bioguide
,date_part('century',min(term_start)) as first_century
,min(term_start) as first_term
,max(term_start) as last_term
,date_part('year',age(max(term_start),min(term_start))) as tenure
FROM legislators_terms
GROUP BY 1
;
id_bioguide  first_century  first_term  last_term  tenure
-----------  -------------  ----------  ---------  ------
A000118      20.0           1975-01-14  1977-01-04  1.0
P000281      20.0           1933-03-09  1937-01-05  3.0
K000039      20.0           1933-03-09  1951-01-03  17.0
...          ...            ...         ...         ...
```

Para concluir, calcularemos `cohort_size` com uma contagem de todos os

legisladores, assim como calcularemos quantos deles sobreviveram por pelo menos 10 anos usando uma instrução CASE e a agregação `count`. O percentual dos que sobreviveram é calculado com a divisão desses dois valores:

```
SELECT first_century
,count(distinct id_bioguide) as cohort_size
,count(distinct case when tenure >= 10 then id_bioguide
                     end) as survived_10
,count(distinct case when tenure >= 10 then id_bioguide end)
/ count(distinct id_bioguide) as pct_survived_10
FROM
(
    SELECT id_bioguide
    ,date_part('century',min(term_start)) as first_century
    ,min(term_start) as first_term
    ,max(term_start) as last_term
    ,date_part('year',age(max(term_start),min(term_start))) as
  tenure
    FROM legislators_terms
    GROUP BY 1
) a
GROUP BY 1
;
century  cohort  survived_10  pct_survived_10
-------  ------  -----------  ---------------
18       368     83           0.2255
19       6299    892          0.1416
20       5091    1853         0.3640
21       760     119          0.1566
```

Já que os mandatos podem ou não ser consecutivos, também podemos calcular a parcela de legisladores de cada século que sobreviveu por cinco ou mais mandatos completos. Na subconsulta, adicionaremos uma contagem para encontrar o número total de mandatos por legislador. Em seguida, na consulta externa, dividiremos o número de legisladores com cinco ou mais mandatos pelo tamanho total da coorte:

```
SELECT first_century
```

```
,count(distinct id_bioguide) as cohort_size
,count(distinct case when total_terms >= 5 then id_bioguide end)
as survived_5
,count(distinct case when total_terms >= 5 then id_bioguide end)
/ count(distinct id_bioguide) as pct_survived_5_terms
FROM
(
    SELECT id_bioguide
    ,date_part('century',min(term_start)) as first_century
    ,count(term_start) as total_terms
    FROM legislators_terms
    GROUP BY 1
) a
GROUP BY 1
;
century  cohort  survived_5  pct_survived_5_terms
-------  ------  ----------  --------------------
18       368     63          0.1712
19       6299    711         0.1129
20       5091    2153        0.4229
21       760     205         0.2697
```

Dez anos ou cinco mandatos é um pouco arbitrário. Também podemos calcular a sobrevivência para cada número de anos ou períodos e exibir os resultados na forma de gráfico ou tabela. Aqui, calcularemos a sobrevivência para cada número de mandatos de 1 a 20. Isso é feito com uma *JOIN* Cartesiana para uma subconsulta que contém esses inteiros derivados pela função `generate_series`:

```
SELECT a.first_century, b.terms
,count(distinct id_bioguide) as cohort
,count(distinct case when a.total_terms >= b.terms then id_bioguide
                     end) as cohort_survived
,count(distinct case when a.total_terms >= b.terms then id_bioguide
                     end)
/ count(distinct id_bioguide) as pct_survived
FROM
(
    SELECT id_bioguide
```

```
    ,date_part('century',min(term_start)) as first_century
    ,count(term_start) as total_terms
    FROM legislators_terms
    GROUP BY 1
) a
JOIN
(
    SELECT generate_series as terms
    FROM generate_series(1,20,1)
) b on 1 = 1
GROUP BY 1,2
;
century  terms  cohort  cohort_survived  pct_survived
-------  -----  ------  --------------  ------------
18       1      368     368             1.0000
18       2      368     249             0.6766
18       3      368     153             0.4157
...      ...    ...     ...             ...
```

Os resultados estão representados em gráfico na Figura 4.11. A sobrevivência foi mais alta no século 20, um resultado que está de acordo com os que vimos anteriormente nos quais a retenção também era mais alta no século 20.

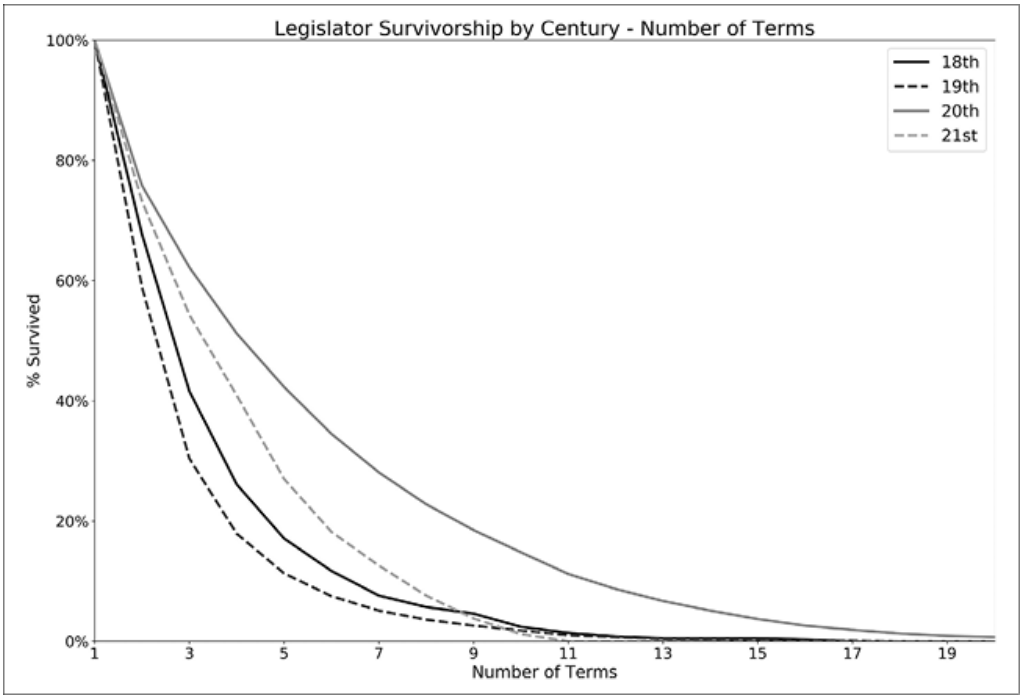


Figura 4.11: Sobrevivência dos legisladores: parcela da coorte que permaneceu no cargo durante esses mandatos ou por períodos mais longos.

A sobrevivência está intimamente relacionada com a retenção. Enquanto a retenção conta as entidades presentes em um número específico de períodos a partir do início, a sobrevivência só considera se uma entidade estava presente a partir de um período específico ou posterior. Como resultado, o código é mais simples, já que precisa apenas da primeira e da última data da série temporal, ou de uma contagem de datas. A criação da coorte é feita de maneira semelhante à da retenção, e as definições de coortes podem vir de dentro da série temporal ou ser derivadas de outra tabela ou subconsulta.

A seguir consideraremos outro tipo de análise que em alguns aspectos é o oposto da sobrevivência. Em vez de calcular se uma entidade está presente no conjunto de dados em um momento específico ou posterior, calcularemos se uma entidade volta a ocorrer ou repete uma ação em determinado período ou em um momento anterior. Essa abordagem se chama retorno ou comportamento de compra repetida.

Retorno, ou comportamento de compra repetida

A sobrevivência é útil para sabermos por quanto tempo uma coorte deve

perdurar. Outro tipo útil de análise de coorte tenta saber se podemos esperar que o membro de uma coorte retorne dentro de uma janela de tempo fornecida e a intensidade da atividade durante essa janela. Esta análise se chama *retorno* ou *comportamento de compra repetida*.

Por exemplo, um site de e-commerce poderia querer saber não só quantos compradores novos foram conquistados por meio de uma campanha de marketing, mas também se eles se tornaram compradores recorrentes. Uma maneira de descobrir isso é simplesmente calculando as compras totais por cliente. No entanto, não é sensato comparar os clientes conquistados dois anos atrás com os conquistados há um mês, já que aqueles tiveram um tempo mais longo para retornar. É quase certo que a coorte mais antiga pareça mais interessante do que a mais nova. Embora de certa forma isso seja verdade, fornece um cenário incompleto do provável comportamento das coortes durante todo o seu tempo de vida.

Para fazer comparações justas entre coortes com diferentes datas iniciais, precisamos criar uma análise baseada em uma *time box*, ou uma janela de tempo fixa a partir da primeira data, e considerar se os membros da coorte retornaram dentro dessa janela. Dessa forma, todas as coortes terão um período de tempo igual a ser considerado, se incluirmos somente as coortes para as quais a janela completa tiver se passado. A análise de retorno é comum para organizações varejistas, mas também pode ser aplicada a outras áreas. Por exemplo, uma universidade poderia querer saber quantos alunos se matricularam em um segundo curso, ou talvez um hospital precisasse saber quantos pacientes precisam de tratamentos médicos contínuos após um incidente inicial.

Para demonstrar a análise de retorno, podemos fazer uma nova pergunta sobre o conjunto de dados de legisladores: quantos legisladores têm mais de um tipo de mandato, e, especificamente, que parcela deles começa como representante e depois se torna senador (alguns senadores depois se tornam representantes, mas isso é menos comum). Já que relativamente poucos fazem essa transição, criaremos uma coorte de legisladores pelo século em que eles se tornaram representantes pela primeira vez.

A primeira etapa é encontrar o tamanho da coorte para cada século, usando a subconsulta e os cálculos de `date_part` vistos anteriormente, somente para legisladores com `term_type = 'rep'`:

```
SELECT date_part('century',a.first_term) as cohort_century
,count(id_bioguide) as reps
FROM
(
    SELECT id_bioguide, min(term_start) as first_term
    FROM legislators_terms
    WHERE term_type = 'rep'
    GROUP BY 1
) a
GROUP BY 1
;
cohort_century  reps
--------------  ----
18              299
19              5773
20              4481
21              683
```

Em seguida, executaremos um cálculo semelhante, com uma *JOIN* para a tabela `legislators_terms`, para encontrar os representantes que depois se tornaram senadores. Isso é feito com as cláusulas `b.term_type = 'sen'` e `b.term_start > a.first_term`:

```
SELECT date_part('century',a.first_term) as cohort_century
,count(distinct a.id_bioguide) as rep_and_sen
FROM
(
    SELECT id_bioguide, min(term_start) as first_term
    FROM legislators_terms
    WHERE term_type = 'rep'
    GROUP BY 1
) a
JOIN legislators_terms b on a.id_bioguide = b.id_bioguide
and b.term_type = 'sen' and b.term_start > a.first_term
GROUP BY 1
;
cohort_century  rep_and_sen
--------------  -----------
18              57
```

```
19              329
20              254
21              25
```

Para concluir, faremos a junção dessas duas subconsultas e calcularemos o percentual de representantes que se tornaram senadores. Uma *LEFT JOIN* é usada; normalmente essa cláusula é recomendada para assegurar que todas as coortes sejam incluídas tendo ou não o evento subsequente ocorrido. Mesmo se houver um século no qual nenhum representante tenha se tornado senador, iremos incluí-lo no conjunto de resultados:

```
SELECT aa.cohort_century
,bb.rep_and_sen / aa.reps as pct_rep_and_sen
FROM
(
    SELECT date_part('century',a.first_term) as cohort_century
    ,count(id_bioguide) as reps
    FROM
    (
        SELECT id_bioguide, min(term_start) as first_term
        FROM legislators_terms
        WHERE term_type = 'rep'
        GROUP BY 1
    ) a
    GROUP BY 1
) aa
LEFT JOIN
(
    SELECT date_part('century',b.first_term) as cohort_century
    ,count(distinct b.id_bioguide) as rep_and_sen
    FROM
    (
        SELECT id_bioguide, min(term_start) as first_term
        FROM legislators_terms
        WHERE term_type = 'rep'
        GROUP BY 1
    ) b
```

```
    JOIN legislators_terms c on b.id_bioguide = b.id_bioguide
    and c.term_type = 'sen' and c.term_start > b.first_term
    GROUP BY 1
) bb on aa.cohort_century = bb.cohort_century
;
cohort_century  pct_rep_and_sen
--------------  --------------
18              0.1906
19              0.0570
20              0.0567
21              0.0366
```

Os representantes do século 18 apresentaram uma maior probabilidade de terem se tornado senadores. No entanto, ainda não aplicamos uma time box para assegurar uma comparação justa. Embora possamos presumir com segurança que todos os legisladores que estavam no cargo nos séculos 18 e 19 não estão mais vivos, muitos dos que foram eleitos pela primeira vez nos séculos 20 e 21 ainda estão no meio de suas carreiras. A inclusão do filtro `WHERE age(c.term_start, b.first_term) <= interval '10 years'` na subconsulta de `bb` cria uma time box de 10 anos. Observe que podemos aumentar ou diminuir facilmente a janela alterando a constante do intervalo. Um filtro adicional, `WHERE first_term <= '2009-12-31'`, aplicado à subconsulta de `a` exclui os legisladores que tinham menos de 10 anos de carreira quando o conjunto de dados foi montado:

```
SELECT aa.cohort_century
,bb.rep_and_sen * 100.0 / aa.reps as pct_10_yrs
FROM
(
    SELECT date_part('century',a.first_term)::int as cohort_century
    ,count(id_bioguide) as reps
    FROM
    (
        SELECT id_bioguide, min(term_start) as first_term
        FROM legislators_terms
        WHERE term_type = 'rep'
        GROUP BY 1
    ) a
```

```
    WHERE first_term <= '2009-12-31'
    GROUP BY 1
) aa
LEFT JOIN
(
    SELECT date_part('century',b.first_term)::int as cohort_century
    ,count(distinct b.id_bioguide) as rep_and_sen
    FROM
    (
        SELECT id_bioguide, min(term_start) as first_term
        FROM legislators_terms
        WHERE term_type = 'rep'
        GROUP BY 1
    ) b
    JOIN legislators_terms c on b.id_bioguide = c.id_bioguide
    and c.term_type = 'sen' and c.term_start > b.first_term
    WHERE age(c.term_start, b.first_term) <= interval '10 years'
    GROUP BY 1
) bb on aa.cohort_century = bb.cohort_century
;
Cohort_century  pct_10_yrs
--------------  ----------
18              0.0970
19              0.0244
20              0.0348
21              0.0764
```

Mesmo com esse novo ajuste, o século 18 teve a parcela mais alta de representantes se tornando senadores dentro de 10 anos, o século 21 tem a segunda parcela mais alta e o século 20 teve uma parcela mais alta do que o século 19.

Já que 10 anos é um período um pouco arbitrário, também poderíamos comparar várias janelas de tempo. Uma opção seria executar a consulta muitas vezes com diferentes intervalos e observar os resultados. Outra seria calcularmos várias janelas no mesmo conjunto de resultados, usando um conjunto de instruções CASE dentro de agregações `count distinct` para formar os intervalos em vez de especificar o intervalo na cláusula

WHERE:

```
SELECT aa.cohort_century
,bb.rep_and_sen_5_yrs * 1.0 / aa.reps as pct_5_yrs
,bb.rep_and_sen_10_yrs * 1.0 / aa.reps as pct_10_yrs
,bb.rep_and_sen_15_yrs * 1.0 / aa.reps as pct_15_yrs
FROM
(
    SELECT date_part('century',a.first_term) as cohort_century
    ,count(id_bioguide) as reps
    FROM
    (
        SELECT id_bioguide, min(term_start) as first_term
        FROM legislators_terms
        WHERE term_type = 'rep'
        GROUP BY 1
    ) a
    WHERE first_term <= '2009-12-31'
    GROUP BY 1
) aa
LEFT JOIN
(
    SELECT date_part('century',b.first_term) as cohort_century
    ,count(distinct case when age(c.term_start,b.first_term)
                            <= interval '5 years'
                        then b.id_bioguide end) as
  rep_and_sen_5_yrs
    ,count(distinct case when age(c.term_start,b.first_term)
                            <= interval '10 years'
                        then b.id_bioguide end) as
  rep_and_sen_10_yrs
    ,count(distinct case when age(c.term_start,b.first_term)
                            <= interval '15 years'
                        then b.id_bioguide end) as
  rep_and_sen_15_yrs
    FROM
    (
        SELECT id_bioguide, min(term_start) as first_term
```

```
        FROM legislators_terms
        WHERE term_type = 'rep'
        GROUP BY 1
    ) b
    JOIN legislators_terms c on b.id_bioguide = c.id_bioguide
    and c.term_type = 'sen' and c.term_start > b.first_term
    GROUP BY 1
) bb on aa.cohort_century = bb.cohort_century
;
cohort_century  pct_5_yrs  pct_10_yrs  pct_15_yrs
--------------  ---------  ----------  ----------
18              0.0502     0.0970      0.1438
19              0.0088     0.0244      0.0409
20              0.0100     0.0348      0.0478
21              0.0400     0.0764      0.0873
```

Com essa saída, podemos ver como a parcela de representantes que se tornaram senadores evoluiu com o tempo tanto dentro de cada coorte quanto entre elas. Além da representação no formato de tabela, geralmente representar a saída em gráfico revela tendências interessantes. Na Figura 4.12, as coortes baseadas em séculos são substituídas por coortes baseadas na primeira década, e as tendências no decorrer de 10 e 20 anos são mostradas. A conversão dos representantes em senadores durante as primeiras décadas da nova legislatura dos EUA era claramente diferente dos padrões nos anos seguintes.

Determinar o comportamento repetido dentro de uma time box fixa é uma ferramenta útil para a comparação de coortes. Isso é verdade principalmente quando os comportamentos têm natureza intermitente, como o comportamento de compra ou o consumo de um conteúdo ou serviço. Na próxima seção, examinaremos como calcular não só se uma entidade teve uma ação subsequente, mas também quantas ações subsequentes ela teve, agregando-as com cálculos cumulativos.

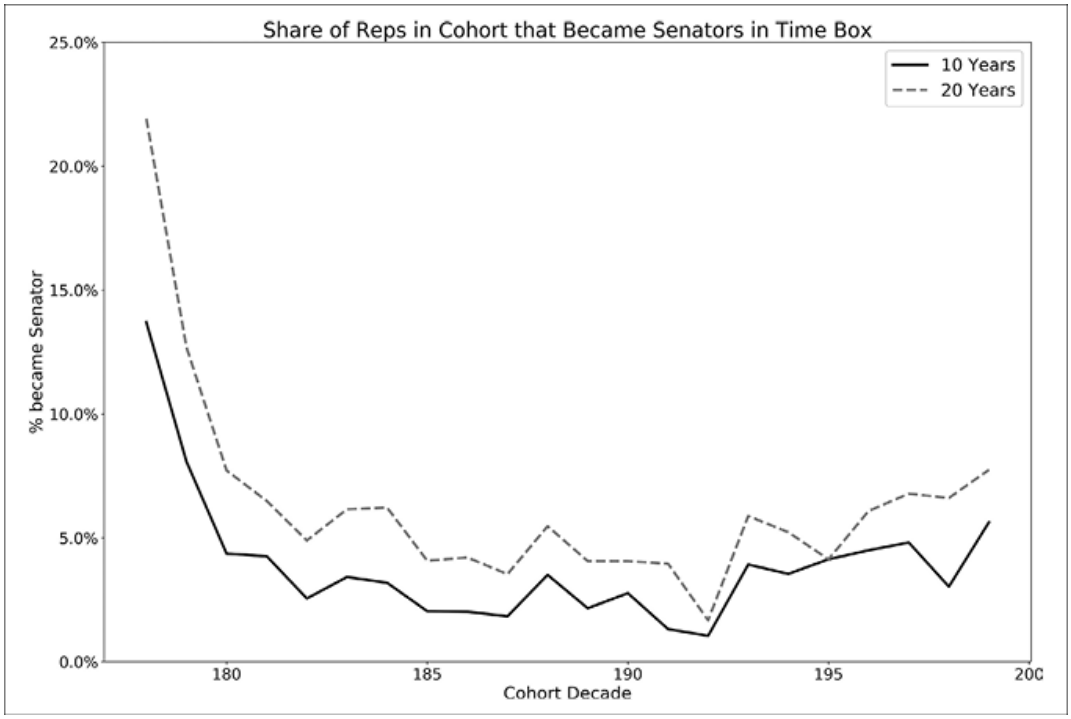


Figura 4.12: Tendência da parcela de representantes de cada coorte, definida pela década inicial, que posteriormente se tornaram senadores.

Cálculos cumulativos

A análise de coorte cumulativa pode ser usada para estabelecer o *valor de tempo de vida cumulativo*, também chamado de *valor de tempo de vida do cliente* (os acrônimos CLTV e LTV são usados de forma intercambiável), e monitorar coortes mais novas para podermos prever qual será o LTV total. Isso é possível porque geralmente o comportamento inicial está altamente relacionado com o comportamento de longo prazo. Os usuários de um serviço que retornam com frequência em seus primeiros dias ou semanas de uso tendem a ser os que têm mais probabilidades de permanecer no longo prazo. Clientes que compram uma segunda ou terceira vez desde cedo devem continuar comprando no decorrer de um período de tempo mais longo. Assinantes que fazem a renovação após o primeiro mês ou ano normalmente permanecem sendo assinantes no decorrer de muitos meses ou anos subsequentes.

Nesta seção, falarei principalmente sobre as atividades de clientes geradoras de receitas, mas essa análise também pode ser aplicada a situações em que os clientes ou entidades tiveram despesas, como por

meio da devolução de produtos, de interações de suporte ou do uso de serviços de assistência médica.

Com os cálculos cumulativos, nos preocuparemos menos com se uma entidade executou uma ação em uma data específica e mais com o total atingido a partir de determinada data. Os cálculos cumulativos usados nesse tipo de análise geralmente são contagens ou somas. Utilizaremos novamente o conceito de time box para assegurar que sejam feitas comparações equivalentes entre as coortes. Examinaremos o número de mandatos iniciados dentro de 10 anos a partir do primeiro `term_start`, criando a coorte de legisladores por século e tipo do primeiro mandato:

```
SELECT date_part('century',a.first_term) as century
,first_type
,count(distinct a.id_bioguide) as cohort
,count(b.term_start) as terms
FROM
(
    SELECT distinct id_bioguide
    ,first_value(term_type) over (partition by id_bioguide
                                  order by term_start) as first_type
    ,min(term_start) over (partition by id_bioguide) as first_term
    ,min(term_start) over (partition by id_bioguide)
     + interval '10 years' as first_plus_10
    FROM legislators_terms
) a
LEFT JOIN legislators_terms b on a.id_bioguide = b.id_bioguide
and b.term_start between a.first_term and a.first_plus_10
GROUP BY 1,2
;
century  first_type  cohort  terms
-------  ----------  ------  -----
18       rep         297     760
18       sen         71      101
19       rep         5744    12165
19       sen         555     795
20       rep         4473    16203
20       sen         618     1008
```

```
21        rep          683      2203
21        sen          77       118
```

A coorte maior é a de representantes que foram eleitos pela primeira vez no século 19, mas a coorte com o maior número de mandatos iniciados dentro de 10 anos é a de representantes que foram eleitos pela primeira vez no século 20. Esse tipo de cálculo pode ser útil para conhecermos a contribuição geral dada por uma coorte para uma organização. O total de vendas ou o total de compras repetidas podem ser métricas valiosas. No entanto, quase sempre o objetivo é conhecer a contribuição por entidade. Os cálculos que poderíamos fazer incluem a média de ações por pessoa, o AOV (average order value, valor médio do pedido), os itens por pedido e os pedidos por cliente. Para usar como padrão o tamanho da coorte, só precisamos fazer a divisão pela coorte inicial, o que calculamos anteriormente com a retenção, a sobrevivência e o retorno. Aqui, faremos isso e também pivotaremos os resultados para o formato de tabela para facilitar as comparações:

```
SELECT century
,max(case when first_type = 'rep' then cohort end) as rep_cohort
,max(case when first_type = 'rep' then terms_per_leg end)
as avg_rep_terms
,max(case when first_type = 'sen' then cohort end) as sen_cohort
,max(case when first_type = 'sen' then terms_per_leg end)
as avg_sen_terms
FROM
(
    SELECT date_part('century',a.first_term) as century
    ,first_type
    ,count(distinct a.id_bioguide) as cohort
    ,count(b.term_start) as terms
    ,count(b.term_start)
     / count(distinct a.id_bioguide) as terms_per_leg
    FROM
    (
        SELECT distinct id_bioguide
        ,first_value(term_type) over (partition by id_bioguide
                                      order by term_start
```

```
                                        ) as first_type
        ,min(term_start) over (partition by id_bioguide) as
  first_term
        ,min(term_start) over (partition by id_bioguide)
         + interval '10 years' as first_plus_10
        FROM legislators_terms
    ) a
    LEFT JOIN legislators_terms b on a.id_bioguide = b.id_bioguide
    and b.term_start between a.first_term and a.first_plus_10
    GROUP BY 1,2
) aa
GROUP BY 1
;
century  rep_cohort  avg_rep_terms  sen_cohort  avg_sen_terms
-------  ----------  -------------  ----------  -------------
18       297         2.6            71          1.4
19       5744        2.1            555         1.4
20       4473        3.6            618         1.6
21       683         3.2            77          1.5
```

Com os mandatos cumulativos normalizados pelo tamanho da coorte, podemos confirmar que os representantes eleitos pela primeira vez no século 20 tiveram o número médio de mandatos mais alto, enquanto os que começaram no século 19 tiveram o número médio de mandatos mais baixo. Os senadores tiveram mandatos em menor número, porém mais longos, do que os dos representantes, e os que começaram no século 20 também tiveram o número médio de mandatos mais alto.

Com frequência os cálculos cumulativos são usados para encontrar o valor de tempo de vida do cliente. Geralmente o LTV é calculado com o uso de valores monetários, como o total de reais gastos por um cliente ou a margem bruta (receita menos custos) gerada por um cliente no decorrer de seu tempo de vida. Para facilitar as comparações entre as coortes, o "tempo de vida" costuma ser selecionado de modo a refletir o tempo de vida médio do cliente, ou períodos que sejam convenientes para a análise, como 3, 5 ou 10 anos. O conjunto de dados de legisladores não contém métricas financeiras, mas seria fácil introduzir valores em reais em qualquer um dos códigos SQL anteriores. Felizmente, o SQL é uma

linguagem suficientemente flexível para podemos adaptar esses templates e abranger uma ampla variedade de questões analíticas.

A análise de coorte inclui um conjunto de técnicas que podem ser usadas para responder a perguntas relacionadas ao comportamento com o passar do tempo e a como vários atributos podem contribuir para gerar diferenças entre grupos. A sobrevivência, o retorno e os cálculos cumulativos lançam uma luz sobre essas questões. Tendo um bom entendimento de como as coortes se comportam, geralmente temos de voltar nossa atenção para a composição ou a combinação das coortes com o tempo, para saber como isso pode afetar a sobrevivência, o retorno ou os valores cumulativos já que essas medidas são muito diferentes nas coortes individuais.

Análise transversal, considerada com base em uma coorte

Até agora neste capítulo examinamos a análise de coorte. Acompanhamos o comportamento das coortes no decorrer do tempo com as análises de retenção, sobrevivência, retorno e comportamento cumulativo. Um dos desafios dessas análises, entretanto, é que, mesmo que elas facilitem identificar alterações nas coortes, pode ser difícil detectar alterações na composição geral de uma base de clientes ou usuários.

Mix shifts, que são alterações na composição da base de clientes ou usuários com o tempo, também podem ocorrer, tornando as coortes posteriores diferentes das mais antigas. As mix shifts podem ocorrer devido a uma expansão internacional, a uma alteração entre estratégias de conquista de clientes orgânicas e pagas, ou a uma passagem de um público-alvo de nicho para um mais amplo de mercado de massa. A criação de coortes, ou segmentos, adicionais ao longo de qualquer uma dessas linhas de ação suspeitas pode ajudar a diagnosticar se uma mix shift está ocorrendo.

Poderíamos confrontar a análise de coorte com a análise transversal, que compara indivíduos ou grupos em um momento específico no tempo. Por exemplo, um estudo transversal poderia correlacionar os anos de escolaridade com a renda atual. Visto pelo lado positivo, geralmente coletar conjuntos de dados para a análise transversal é mais fácil, já que

não é necessária uma série temporal. A análise transversal pode ser reveladora, gerando hipóteses para investigação posterior. Pelo lado negativo, costuma ocorrer um tipo de viés de seleção chamado viés de sobrevivência, que pode levar a conclusões falsas.

Viés de sobrevivência

"Vamos dar uma olhada em nossos melhores clientes e ver o que eles têm em comum." Essa ideia aparentemente inocente e bem-intencionada pode levar a algumas conclusões muito problemáticas. O *viés de sobrevivência* é o erro lógico que ocorre quando nos concentramos nas pessoas ou coisas que passaram em algum processo de seleção e ignoramos as que não passaram. Normalmente isso acontece porque as entidades não existem mais no conjunto de dados no momento da seleção, porque foram malsucedidas, mudaram, ou não fazem mais parte da população por alguma outra razão. Concentrarmo-nos apenas na população restante pode levar a conclusões excessivamente otimistas, porque as falhas foram ignoradas.

Muito tem sido escrito sobre algumas pessoas que deixaram a universidade e fundaram empresas de tecnologia altamente bem-sucedidas. Isso não significa que você deve abandonar imediatamente a universidade, já que a grande maioria das pessoas que o faz não se torna um CEO de sucesso. Essa parte da população só gera manchetes sensacionalistas; logo, é fácil nos esquecermos dessa realidade.

No contexto dos clientes bem-sucedidos, o viés de sobrevivência pode surgir como uma observação de que os melhores clientes tendem a morar, por exemplo, na Califórnia ou no Texas e tendem a ter de 18 a 30 anos. É uma população grande, e no fim das contas essas características podem ser compartilhadas por muitos clientes que mudaram antes da data da análise. Voltar à população original pode revelar que outros grupos demográficos, como o de 41 a 50 anos de Vermont, continuam sendo clientes e estão gastando mais com o passar do tempo, ainda que haja menos deles em termos absolutos. A análise de coorte ajuda a distinguir e a reduzir o viés de sobrevivência.

A análise de coorte é uma maneira de resolvermos o viés de sobrevivência pela inclusão de todos os membros de uma coorte inicial na análise. Podemos pegar uma série de seções transversais da análise de coorte para entender como a combinação de entidades mudou com o tempo. Em qualquer data específica, usuários de várias coortes estarão presentes. Podemos empregar a análise transversal para examiná-las, como camadas de sedimento, para revelar novos insights. No próximo exemplo, criaremos uma série temporal da parcela de legisladores de cada coorte para cada

ano do conjunto de resultados.

A primeira etapa é encontrar o número de legisladores no cargo a cada ano pela junção da tabela `legislators` com `date_dim`, em que a data de `date_dim` deve estar entre as datas de início e fim de cada mandato. Aqui usamos 31 de dezembro para cada ano para encontrar os legisladores que estão no cargo no fim de cada ano:

```
SELECT b.date, count(distinct a.id_bioguide) as legislators
FROM legislators_terms a
JOIN date_dim b on b.date between a.term_start and a.term_end
and b.month_name = 'December' and b.day_of_month = 31
and b.year <= 2019
GROUP BY 1
;
date        legislators
----------  -----------
1789-12-31  89
1790-12-31  95
1791-12-31  99
...         ...
```

Em seguida, adicionaremos os critérios de criação de coortes por século fazendo a junção em uma subconsulta com `first_term` calculado:

```
SELECT b.date
,date_part('century',first_term) as century
,count(distinct a.id_bioguide) as legislators
FROM legislators_terms a
JOIN date_dim b on b.date between a.term_start and a.term_end
and b.month_name = 'December' and b.day_of_month = 31
and b.year <= 2019
JOIN
(
    SELECT id_bioguide, min(term_start) as first_term
    FROM legislators_terms
    GROUP BY 1
) c on a.id_bioguide = c.id_bioguide
GROUP BY 1,2
;
```

```
date        century  legislators
----------  -------  -----------
1789-12-31  18       89
1790-12-31  18       95
1791-12-31  18       99
...         ...      ...
```

Para concluir, calcularemos o percentual do total de legisladores de cada ano que a coorte secular representa. Isso pode ser feito de duas maneiras, dependendo da forma de saída desejada. A primeira maneira é mantendo uma linha para cada combinação de data e século e usando uma função de janela `sum` no denominador do cálculo do percentual:

```
SELECT date
,century
,legislators
,sum(legislators) over (partition by date) as cohort
,legislators / sum(legislators) over (partition by date)
as pct_century
FROM
(
    SELECT b.date
    ,date_part('century',first_term) as century
    ,count(distinct a.id_bioguide) as legislators
    FROM legislators_terms a
    JOIN date_dim b on b.date between a.term_start and a.term_end
    and b.month_name = 'December' and b.day_of_month = 31
    and b.year <= 2019
    JOIN
    (
        SELECT id_bioguide, min(term_start) as first_term
        FROM legislators_terms
        GROUP BY 1
    ) c on a.id_bioguide = c.id_bioguide
    GROUP BY 1,2
) a
;
date        century  legislators  cohort  pct_century
----------  -------  -----------  ------  -----------
```

```
2018-12-31  20       122          539     0.2263
2018-12-31  21       417          539     0.7737
2019-12-31  20       97           537     0.1806
2019-12-31  21       440          537     0.8194
...         ...      ...          ...     ...
```

A segunda abordagem resulta em uma linha por ano, com uma coluna para cada século, um formato de tabela no qual pode ser mais fácil procurar tendências:

```
SELECT date
,coalesce(sum(case when century = 18 then legislators end)
          / sum(legislators),0) as pct_18
,coalesce(sum(case when century = 19 then legislators end)
          / sum(legislators),0) as pct_19
,coalesce(sum(case when century = 20 then legislators end)
          / sum(legislators),0) as pct_20
,coalesce(sum(case when century = 21 then legislators end)
          / sum(legislators),0) as pct_21
FROM
(
    SELECT b.date
    ,date_part('century',first_term) as century
    ,count(distinct a.id_bioguide) as legislators
    FROM legislators_terms a
    JOIN date_dim b on b.date between a.term_start and a.term_end
     and b.month_name = 'December' and b.day_of_month = 31
     and b.year <= 2019
    JOIN
    (
        SELECT id_bioguide, min(term_start) as first_term
        FROM legislators_terms
        GROUP BY 1
    ) c on a.id_bioguide = c.id_bioguide
    GROUP BY 1,2
) aa
GROUP BY 1
;
date        pct_18  pct_19  pct_20  pct_21
```

```
----------  ------  ------  ------  ------
2017-12-31  0       0       0.2305  0.7695
2018-12-31  0       0       0.2263  0.7737
2019-12-31  0       0       0.1806  0.8193
...         ...     ...     ...     ...
```

Podemos representar a saída em gráfico, como na Figura 4.13, para ver como as coortes de legisladores mais novas tomam gradualmente o lugar de coortes mais antigas, até elas próprias serem substituídas por novas coortes.

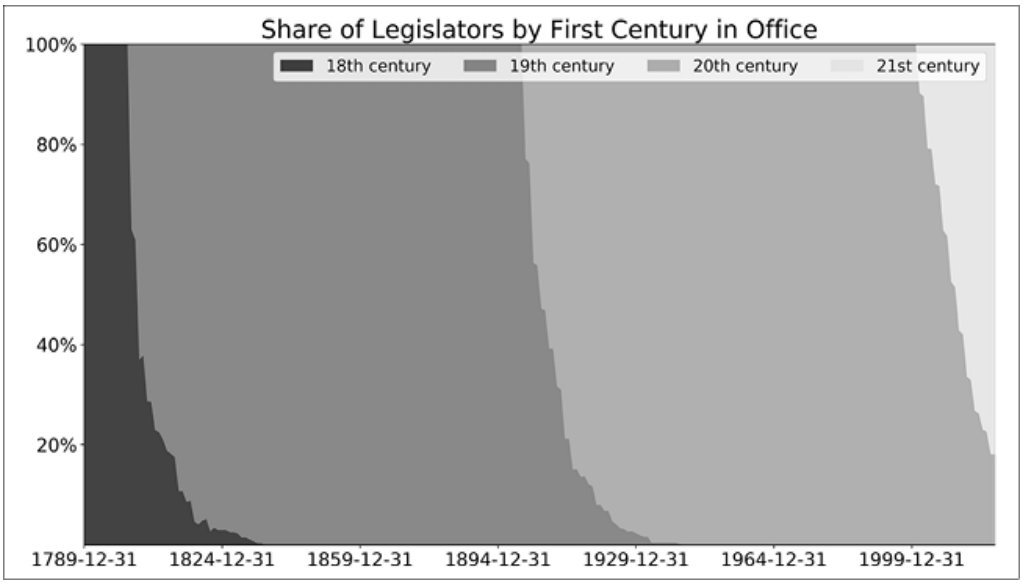


Figura 4.13: Percentual de legisladores em cada ano, pelo século em que foram eleitos pela primeira vez.

Em vez de criar a coorte em `first_term`, podemos criá-la com base na permanência no cargo. Sabermos a parcela de clientes que são relativamente novos, que há algum tempo são usuários, ou que são usuários de longa data em vários momentos no tempo pode ser revelador. Examinaremos como a permanência dos legisladores no Congresso mudou com o tempo.

A primeira etapa é calcular, para cada ano, o número acumulado de anos no cargo para cada legislador. Já que pode haver lacunas entre os mandatos se os legisladores não forem eleitos ou deixarem o cargo por outras razões, primeiro encontraremos na subconsulta cada ano em que o legislador estava no cargo no fim do ano. Em seguida, usaremos uma

função de janela `count`, com a janela abrangendo as linhas `unbounded preceding`, ou todas as linhas anteriores desse legislador, e a linha atual (`current row`):

```
SELECT id_bioguide, date
,count(date) over (partition by id_bioguide
                   order by date rows between
                   unbounded preceding and current row
                   ) as cume_years
FROM
(
    SELECT distinct a.id_bioguide, b.date
    FROM legislators_terms a
    JOIN date_dim b on b.date between a.term_start and a.term_end
     and b.month_name = 'December' and b.day_of_month = 31
     and b.year <= 2019
) aa
;
id_bioguide  date        cume_years
-----------  ----------  ----------
A000001      1951-12-31  1
A000001      1952-12-31  2
A000002      1947-12-31  1
A000002      1948-12-31  2
A000002      1949-12-31  3
...          ...         ...
```

Agora contaremos o número de legisladores para cada combinação de `date` e `cume_years` para criar uma distribuição:

```
SELECT date, cume_years
,count(distinct id_bioguide) as legislators
FROM
(
SELECT id_bioguide, date
,count(date) over (partition by id_bioguide
                   order by date rows between
                   unbounded preceding and current row
                   ) as cume_years
```

```
FROM
(
    SELECT distinct a.id_bioguide, b.date
    FROM legislators_terms a
    JOIN date_dim b on b.date between a.term_start and a.term_end
     and b.month_name = 'December' and b.day_of_month = 31
     and b.year <= 2019
    GROUP BY 1,2
    ) aa
) aaa
GROUP BY 1,2
;
date         cume_years  legislators
-----------  ----------  ----------
1789-12-31   1           89
1790-12-31   1           6
1790-12-31   2           89
1791-12-31   1           37
...          ...         ...
```

Antes de calcular o percentual de cada permanência no mandato por ano e ajustar o formato da apresentação, poderíamos considerar agrupar as permanências. A criação rápida de um perfil para os resultados obtidos até agora revela que em alguns anos quase 40 permanências diferentes são representadas. Isso deve ser difícil de visualizar e interpretar:

```
SELECT date, count(*) as tenures
FROM
(
    SELECT date, cume_years
    ,count(distinct id_bioguide) as legislators
    FROM
    (
        SELECT id_bioguide, date
        ,count(date) over (partition by id_bioguide
                           order by date rows between
                           unbounded preceding and current row
                           ) as cume_years
        FROM
```

```
        (
            SELECT distinct a.id_bioguide, b.date
            FROM legislators_terms a
            JOIN date_dim b
             on b.date between a.term_start and a.term_end
             and b.month_name = 'December' and b.day_of_month = 31
             and b.year <= 2019
            GROUP BY 1,2
        ) aa
    ) aaa
    GROUP BY 1,2
) aaaa
GROUP BY 1
;
date         tenures
-----------  -------
1998-12-31   39
1994-12-31   39
1996-12-31   38
...          ...
```

Portanto, poderíamos agrupar os valores. Não há uma maneira definitiva de agrupar permanências. Se houver grupos de permanências definidos organizacionalmente, use-os. Caso contrário, geralmente tento dividi-las em três a cinco grupos de tamanho aproximadamente igual. Aqui agruparemos as permanências em quatro coortes, em que `cume_years` será menor ou igual a 4 anos, terá entre 5 e 10 anos, terá entre 11 e 20 anos, e será maior ou igual a 21 anos:

```
SELECT date, tenure
,legislators / sum(legislators) over (partition by date)
as pct_legislators
FROM
(
    SELECT date
    ,case when cume_years <= 4 then '1 to 4'
          when cume_years <= 10 then '5 to 10'
          when cume_years <= 20 then '11 to 20'
```

```
      else '21+' end as tenure
    ,count(distinct id_bioguide) as legislators
    FROM
    (
        SELECT id_bioguide, date
        ,count(date) over (partition by id_bioguide
                           order by date rows between
                           unbounded preceding and current row
                           ) as cume_years
        FROM
        (
            SELECT distinct a.id_bioguide, b.date
            FROM legislators_terms a
            JOIN date_dim b
             on b.date between a.term_start and a.term_end
             and b.month_name = 'December' and b.day_of_month = 31
             and b.year <= 2019
            GROUP BY 1,2
        ) a
    ) aa
    GROUP BY 1,2
) aaa
;
date        tenure    pct_legislators
----------  -------   ---------------
2019-12-31  1 to 4    0.2998
2019-12-31  5 to 10   0.3203
2019-12-31  11 to 20  0.2011
2019-12-31  21+       0.1788
...         ...       ...
```

A representação em gráfico dos resultados na Figura 4.14 mostra que, nos primeiros anos de legislatura do país, a maioria dos legisladores apresentava muito pouca continuidade. Em anos mais recentes, a parcela de legisladores com 21 anos ou mais no gabinete tem aumentado. Também há aumentos periódicos interessantes referentes aos legisladores com permanência de 1 a 4 anos que pode refletir mudanças nas tendências políticas.

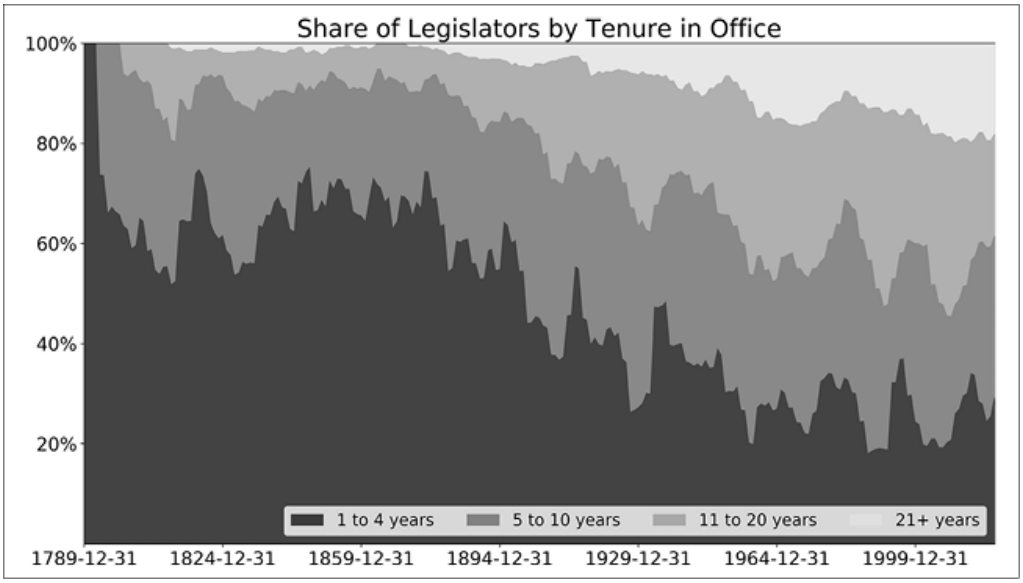


Figura 4.14: Percentual de legisladores por número de anos no cargo.

A seção transversal de uma população em qualquer momento no tempo é composta de membros de várias coortes. Criar uma série temporal dessas seções transversais é outra maneira interessante de analisar tendências. Combinar essa abordagem com os insights provenientes da retenção pode fornecer um cenário mais robusto das tendências de qualquer organização.

Conclusão

A análise de coorte é uma maneira útil para investigarmos como os grupos mudam com o tempo, seja a partir do ponto de vista da retenção, do comportamento repetido ou de ações cumulativas. Ela é retrospectiva e olha para o passado das populações usando atributos intrínsecos ou atributos derivados do comportamento. Correlações interessantes e possivelmente úteis podem ser encontradas com esse tipo de análise. No entanto, como se costuma dizer, correlação não implica causalidade. Para determinar a causalidade real, experimentos randomizados são o padrão ouro. O Capítulo 7 dará mais detalhes sobre a análise experimental.

Antes de nos voltarmos para a experimentação, entretanto, temos alguns outros tipos de análise para abordar. A seguir, examinaremos a análise de texto: geralmente componentes de análise de texto surgem em outras análises, e essa é uma faceta interessante da análise propriamente dita.

CAPÍTULO 5

Análise de texto

Nos dois últimos capítulos, examinamos várias aplicações das datas e números com a análise de séries temporais e de coorte. No entanto, geralmente os conjuntos de dados não são compostos apenas de valores numéricos e dos timestamps associados. Indo de atributos qualitativos a texto livre, os campos de caracteres costumam estar cheios de informações potencialmente interessantes. Embora os bancos de dados tenham um desempenho de destaque em cálculos numéricos como contar, somar e encontrar a média de itens, eles também são muito bons na execução de operações com dados de texto.

Começarei este capítulo fornecendo uma visão geral dos tipos de tarefas de análise de texto para os quais o SQL é útil, e mostrando outros tipos para os quais outra linguagem de programação seria uma opção melhor. Em seguida, introduzirei nosso conjunto de dados de avistamentos de OVNIs. Depois, passaremos para a codificação, abordando características e perfis de textos, o parsing (análise) de dados com SQL, a execução de transformações, a construção de um texto novo derivado das partes e, para concluir, a busca de elementos dentro de blocos maiores de texto, inclusive com expressões regulares.

Por que fazer análise de texto com SQL?

Entre os imensos volumes de dados gerados todo dia, uma grande parcela é composta de texto: palavras, frases, parágrafos e até mesmo documentos mais longos. Os dados de texto usados para a análise podem vir de várias fontes, incluindo descritores fornecidos por humanos ou programas de computador, arquivos de log, tíquetes de suporte, pesquisa com clientes, postagens em mídia social, ou feeds de notícias. Os textos existentes nos bancos de dados variam de *estruturados* (situação em que os dados estão em campos de tabela diferentes com significados distintos) a

semiestruturados (nesse caso, os dados estão em colunas separadas, mas podem precisar de parsing ou limpeza para ser úteis) ou em grande parte *não estruturados* (nos quais campos longos de tipo VARCHAR ou BLOB contêm strings de tamanho arbitrário que precisam de extensa estruturação antes da análise posterior). Felizmente, o SQL tem várias funções úteis que podem ser combinadas para a execução de muitas tarefas de estruturação e análise de texto.

O que é análise de texto?

Análise de texto é o processo de derivar significado e insights a partir de dados de texto. Há duas categorias amplas de análise de texto, cuja diferença está em a saída ser qualitativa ou quantitativa. A *análise qualitativa*, que também pode ser chamada de *análise textual*, tenta entender e sintetizar o significado de um texto individual ou de um conjunto de textos, geralmente aplicando outros conhecimentos ou conclusões exclusivas. Esse trabalho costuma ser feito por jornalistas, historiadores e pesquisadores da experiência do usuário. A *análise quantitativa* de texto também tenta sintetizar informações de dados textuais, mas a saída é quantitativa. As tarefas incluem a categorização e a extração de dados, e normalmente a análise é feita na forma de contagens ou frequências, quase sempre apresentando tendências com o passar do tempo. O SQL é mais adequado para a análise quantitativa; logo, é nela que o restante do capítulo se concentrará. No entanto, se você tiver a oportunidade de trabalhar com algum especialista no primeiro tipo de análise de texto, aproveite a experiência. Combinar as análises qualitativa e quantitativa é uma ótima maneira de obter novos insights e persuadir colegas relutantes.

A análise de texto engloba vários objetivos ou estratégias. O primeiro é a extração de texto, tarefa na qual um trecho de dados útil deve ser obtido a partir do restante do texto. Outro objetivo é a categorização, em que informações são extraídas ou analisadas a partir dos dados textuais para a atribuição de tags ou categorias às linhas de um banco de dados. Ainda outra estratégia é a análise de sentimento, em que o objetivo é entender o estado de espírito ou a intenção do criador do texto em uma escala que vai de negativo a positivo.

Embora a análise de texto já exista há algum tempo, o interesse e as

pesquisas nessa área aumentaram com o advento do machine learning e dos recursos computacionais que geralmente são necessários no trabalho com grandes volumes de dados de texto. O NLP (*natural language processing, processamento de linguagem natural*) fez grandes avanços no reconhecimento, na classificação e até mesmo na geração de dados de texto totalmente novos. A linguagem humana é muito complexa, com diferentes idiomas e dialetos, gramáticas e gírias, sem mencionar os milhares de palavras, algumas tendo significados sobrepostos ou modificando sutilmente o significado de outras palavras. Como veremos, o SQL é bom para alguns tipos de análise de texto, mas, para outros, existem linguagens e ferramentas que são mais adequadas.

Por que o SQL é uma boa opção para a análise de texto

Há várias razões para usarmos SQL para análise de texto. Uma das mais óbvias é quando os dados já estão em um banco de dados. Os bancos de dados modernos têm muito poder computacional e ele pode ser usado para tarefas textuais, além das outras tarefas que discutimos até agora. A transferência de dados para um flat file[1] para análise com outra linguagem ou ferramenta é demorada; logo, fazer o máximo de trabalho possível com SQL dentro do banco de dados tem suas vantagens.

Se os dados ainda não estiverem em um banco de dados, no caso de conjuntos de dados relativamente grandes pode valer a pena movê-los para um banco de dados. Os bancos de dados são mais poderosos do que as planilhas para o processamento de transformações em muitos registros. O SQL é menos propenso a erros do que as planilhas, já que não é necessário copiar e colar, e os dados originais permanecem intactos. Os dados poderiam ser alterados com um comando *UPDATE*, mas é difícil isso ocorrer acidentalmente.

O SQL também é uma boa opção quando o objetivo final é algum tipo de quantificação. Contar quantos tíquetes de suporte contêm uma frase-chave (key phrase) e fazer o parsing das categorias de um texto maior para serem usadas no agrupamento de registros são bons exemplos de quando o SQL se destaca. O SQL é bom na limpeza e na estruturação de campos de texto. A *limpeza* inclui a remoção de caracteres adicionais ou espaço em branco, a correção de letras maiúsculas incorretas e a padronização de

grafias. A *estruturação* envolve a criação de novas colunas a partir de elementos extraídos ou derivados de outros campos ou a construção de novos campos baseados em partes armazenadas em diferentes locais. Funções de string podem ser aninhadas ou aplicadas aos resultados de outras funções, permitindo a execução de quase qualquer manipulação que possa ser necessária.

O código SQL para a análise de texto pode ser simples ou complexo, mas é sempre baseado em regras. Em um sistema baseado em regras, o computador segue rigorosamente um conjunto de regras ou instruções. Isso é o oposto do machine learning, em que o computador se adapta de acordo com os dados. As regras são adequadas porque são fáceis para os humanos entenderem. Elas são escritas na forma de código e podem ser verificadas para garantir que produzam a saída desejada. A desvantagem é que podem se tornar longas e complicadas, principalmente quando existem muitos casos diferentes para serem manipulados. Isso também pode dificultar a manutenção. Se a estrutura ou o tipo dos dados inseridos na coluna mudar, o conjunto de regras terá de ser atualizado. Em mais de uma ocasião, comecei com o que parecia uma simples instrução CASE com 4 ou 5 linhas e acabei vendo-a crescer para 50 ou 100 linhas à medida que a aplicação mudava. Mesmo assim as regras podem ser a abordagem certa, mas manter a sincronia com a equipe de desenvolvimento no que diz respeito a alterações é uma boa ideia.

Para concluir, o SQL é uma boa opção quando sabemos antecipadamente o que estamos procurando. Há várias funções poderosas, incluindo as expressões regulares, que permitem procurar, extrair ou substituir informações específicas. "Quantos revisores mencionam 'bateria com pouca vida útil' em suas revisões?" é uma pergunta que o SQL pode ajudá-lo a responder. Por outro lado, não será tão fácil responder a "Por que esses clientes estão zangados?".

Quando o SQL não é uma boa opção

Basicamente o SQL nos permite aproveitar o poder do banco de dados para aplicar um conjunto de regras, embora geralmente regras poderosas, a um conjunto textual para torná-lo mais útil para análise. Certamente o SQL não é a única opção para a análise de texto, e há vários casos de uso

para os quais ele não é a melhor opção. É bom conhecê-los.

A primeira categoria engloba os casos de uso para os quais um ser humano seria a opção mais apropriada. Quando o conjunto de dados for muito pequeno ou muito novo, rotular manualmente pode ser mais rápido e informativo. Além disso, se o objetivo for ler todos os registros e criar um resumo qualitativo dos principais temas, um humano será uma opção melhor.

A segunda categoria é quando há a necessidade de busca e recuperação de registros específicos contendo strings de texto com baixa latência. Ferramentas como o Elasticsearch ou o Splunk foram desenvolvidas para indexar strings para esses casos de uso. O desempenho pode ser um problema com o uso de SQL e bancos de dados; essa é uma das principais razões para tentarmos estruturar os dados em colunas discretas que possam ser pesquisadas com mais facilidade pelo mecanismo do banco de dados (database engine).

A terceira categoria é a composta das tarefas do grupo mais amplo do NLP, em que as abordagens de machine learning e as linguagens que as executam, como Python, são uma opção melhor. A análise de sentimentos, usada para avaliar intervalos de sentimentos positivos ou negativos em textos, só pode ser manipulada de uma maneira mais simples com SQL. Por exemplo, as palavras “amor” e “ódio” poderiam ser extraídas e usadas para categorizar registros, mas dado o conjunto de palavras que podem expressar emoções positivas e negativas, assim como todas as maneiras que temos para refutar essas palavras, seria quase impossível criar um conjunto de regras com SQL para manipulá-las. As tags de parte do discurso, em que as palavras de um texto são rotuladas como substantivos, verbos e assim por diante, são manipuladas mais adequadamente com as bibliotecas disponíveis em Python. A geração de linguagem, ou a criação de um texto totalmente novo baseado nos aprendizados provenientes de exemplos de texto, é outro caso que seria manipulado de uma forma melhor em outras ferramentas. Veremos como é possível criar texto novo concatenando trechos de dados, mas o SQL é limitado por regras e não aprenderá automaticamente com novos exemplos no conjunto de dados, assim como também não se adaptará a eles.

Agora que discutimos as diversas razões para a utilização de SQL na análise de texto, assim como os tipos de casos de uso a serem evitados, examinaremos o conjunto de dados empregado nos exemplos antes de passarmos para o código SQL.

Conjunto de dados de avistamentos de OVNIs

Nos exemplos deste capítulo, usaremos um conjunto de dados de avistamentos de OVNIs organizado pelo *National UFO Reporting Center* (*http://www.nuforc.org*). O conjunto de dados é composto de aproximadamente 95.000 relatos postados entre 2006 e 2020. Os relatos são de pessoas que forneceram informações por meio de um formulário online.

A tabela com a qual trabalharemos chama-se `ufo` e tem apenas duas colunas. A primeira é uma coluna composta chamada `sighting_report` (relato de avistamento), que contém informações sobre quando o avistamento ocorreu, quando ele foi relatado e quando foi postado. Ela também contém metadados sobre o local, a forma e a duração do evento de avistamento. A segunda coluna é um campo de texto chamado `description`, que contém a descrição completa do evento. A Figura 5.1 exibe uma amostra dos dados.

Por meio dos exemplos e da discussão deste capítulo, mostrarei como fazer o parsing da primeira coluna para a obtenção de dados estruturados e descritores. Também mostrarei como podemos executar várias análises no campo `description`. Se eu fosse trabalhar com esses dados de maneira contínua, poderia considerar a criação de um pipeline ETL[2], uma tarefa que processa os dados da mesma forma regularmente, e o armazenamento dos dados estruturados resultantes em uma nova tabela. Nos exemplos do capítulo, entretanto, usaremos a tabela bruta.

*	sighting_report	description
1	Occurred : 6/24/1980 14:00 (Entered as : 06/24/80 14:00)Reported: 4/6/2006 8:45:08 PM 20:45Posted: 5/15/2006Location: Mount W...	Missing Time: Two PeopleMy mother and I have a long history of UFO sightings/involvement
2	Occurred : 4/6/2006 02:05 (Entered as : 04/06/06 02:05)Reported: 4/6/2006 6:06:21 PM 18:06Posted: 5/15/2006Location: Ottoville, O...	Bright lights near Ottoville, OhioHeading westward on SR 189 out of Ft. Jennings, Ohio at th
3	Occurred : 9/11/2001 09:00 (Entered as : 9/11/01 10:00)Reported: 4/6/2006 4:04:27 PM 16:04Posted: 5/15/2006Location: Erie, PASh...	Planes guided into the Trade Centers by UFOs.I can't believe not many people reported this
4	Occurred : 4/6/2006 21:50 (Entered as : 04/06/06 21:50)Reported: 4/6/2006 2:26:52 PM 14:26Posted: 5/15/2006Location: Leeds (UK/...	two bright lights in sky - Brightened then dimmed - definatly not a planetwo lights in the sky
5	Occurred : 4/4/2006 02:00 (Entered as : 4/4/06 2:00)Reported: 4/6/2006 12:16:18 PM 12:16Posted: 5/15/2006Location: Frazier Park, ...	The Love of my Life is snoring so I can't sleep, I go into the kitchen and watch the snow fall,
6	Occurred : 4/5/2006 22:25 (Entered as : 04/05/06 22:25)Reported: 4/6/2006 11:40:07 AM 11:40Posted: 5/15/2006Location: Oklahom...	Two hovering orange circles drop third orange circle over oklahoma.My girlfriend and i were
7	Occurred : 11/15/2005 15:00 (Entered as : 11-15-05 15:00)Reported: 4/6/2006 9:53:25 AM 09:53Posted: 5/15/2006Location: Utah (u...	bright changing light hover and landed behind a mountian. lat 37°49'15.60"N lon 112°43'22.3
8	Occurred : 4/5/2006 22:15 (Entered as : 04/05/06 22:15)Reported: 4/5/2006 11:24:51 PM 23:24Posted: 5/15/2006Location: Graham, ...	At first glance, it looked like a plane, we realized that it wasn't moven, but dancing in place
9	Occurred : 10/15/1994 14:00 (Entered as : 10/15/1994 14:00)Reported: 4/6/2006 8:32:00 AM 08:32Posted: 5/15/2006Location: Circle...	Silver disc saucer hovers above town, then disappearsI was driving my car with three friend
10	Occurred : 4/15/1973 21:30 (Entered as : 04/15/73 21:30)Reported: 4/6/2006 8:24:26 AM 08:24Posted: 5/15/2006Location: Cold Lak...	Strange Lights Indeed Upon returning home on a beautiful spring evening, in the approximat
11	Occurred : 4/5/2006 03:15 (Entered as : 04/05/06 3:15)Reported: 4/6/2006 7:05:26 AM 07:05Posted: 5/15/2006Location: Erie, PASha...	Three disk like objects flew over Erie, PA at around 3:15, there were lights and not any notic
12	Occurred : 10/8/2005 06:15 (Entered as : 10/08/05 6:15)Reported: 4/6/2006 6:24:23 AM 06:24Posted: 5/15/2006Location: Orlando, F...	Well me and my girlfriend were waiting for the bus one morning and I was looking at the sky
13	Occurred : 1/19/2006 19:00 (Entered as : january 19 19:00)Reported: 4/6/2006 3:50:39 AM 03:50Posted: 5/15/2006Location: Torranc...	square shaped object over Southern CaliforniaDriving home from work, from the corner of my
14	Occurred : 4/6/2006 01:30 (Entered as : 04/06/06 01:30)Reported: 4/6/2006 2:41:06 AM 02:41Posted: 5/15/2006Location: Scottsdale...	Black triangle seen flying silently in the night sky.I saw what looked like the Stealth Bomber
15	Occurred : 1/1/2006 04:45 (Entered as : 010106 04:45)Reported: 4/6/2006 12:22:32 AM 00:22Posted: 5/15/2006Location: Claremore,...	Bright geosynchronus light that appears every morning.On 01/01/06 at approx. 0445, my pa
16	Occurred : 4/5/2006 08:31 (Entered as : 4-5-06 8:31)Reported: 4/5/2006 10:40:21 PM 22:40Posted: 5/15/2006Location: Shingletown,...	scarry!I saw amassive blue objedt floating in the sky for about 60 seconds the n spilt in 2 t
17	Occurred : 10/1/1997 08:00 (Entered as : oct 97 08:00)Reported: 4/5/2006 9:37:03 PM 21:37Posted: 5/15/2006Location: Oregon (rura...	noisy rectangular object falling from the sky and crash landedA friend of mine and I were ou
18	Occurred : 4/5/2006 20:15 (Entered as : 04/05/06 20:15)Reported: 4/5/2006 8:53:29 PM 20:53Posted: 5/15/2006Location: Centralia, ...	Sighted over Cooks Hill Road and Joppish Road just west of I-5. Dusk clear sky. Large fireb
19	Occurred : 4/5/2006 22:25 (Entered as : 04/05/06 22:25)Reported: 4/5/2006 8:50:32 PM 20:50Posted: 5/15/2006Location: San Marco...	Flying triangle in sky over San MarcosI looked up thinking that I was seeing a plane, but it w
20	Occurred : 4/5/2006 20:30 (Entered as : 04/05/2006 20:30)Reported: 4/5/2006 8:49:44 PM 20:49Posted: 5/15/2006Location: Des Moi...	Weird fireball that just diappeared without fanfare and followed up by 4 fighters.Say it streal
21	Occurred : 9/14/1981 20:30 (Entered as : 09/14/81 20:30)Reported: 4/5/2006 8:20:41 PM 20:20Posted: 5/15/2006Location: Lilburn, G...	I was coming home from my friends house,I stopped to rest on the side of the road near a w
22	Occurred : 3/25/2006 13:00 (Entered as : 03-25-06 13:00)Reported: 4/5/2006 7:34:04 PM 19:34Posted: 5/15/2006Location: Summers...	white square object moving very fastThis thing was unlike anything I have ever seen. Instea

Figura 5.1: Amostra da tabela ufo.

Passemos para o código, começando com o SQL de exploração e caracterização do texto dos avistamentos.

Características do texto

O tipo de dado mais flexível de um banco de dados é o VARCHAR, porque quase qualquer dado pode ser inserido nos campos desse tipo. Como resultado, os dados de texto dos bancos de dados têm várias formas e tamanhos. Como ocorre com outros conjuntos de dados, a criação de perfis e a caracterização dos dados é uma das primeiras tarefas que devemos executar. Depois podemos desenvolver uma estratégia para os tipos de limpeza e parsing que talvez sejam necessários na análise.

Uma maneira de conhecermos os dados do texto seria identificando o número de caracteres de cada valor, o que pode ser feito com a função `length` (ou `len` em alguns bancos de dados). Essa função recebe o campo de strings ou caracteres como argumento e é semelhante às funções encontradas em outras linguagens e programas de planilha:

```
SELECT length('Sample string');
length
------
13
```

Podemos criar uma distribuição de tamanhos de campos para ter uma ideia do tamanho típico e também se há algum valor discrepante (outlier) extremo que possa precisar ser manipulado de maneiras especiais:

```
SELECT length(sighting_report), count(*) as records
```

```
FROM ufo
GROUP BY 1
ORDER BY 1
;
length  records
------  -------
90      1
91      4
92      8
...     ...
```

Podemos ver na Figura 5.2 que a maioria dos registros tem entre aproximadamente 150 e 180 caracteres, e muito poucos têm menos de 140 ou mais de 200 caracteres. Os tamanhos do campo `description` variam de 5 a 64.921 caracteres. É possível presumir que existe muito mais variedade nesse campo antes mesmo de executarmos qualquer criação de perfil adicional.

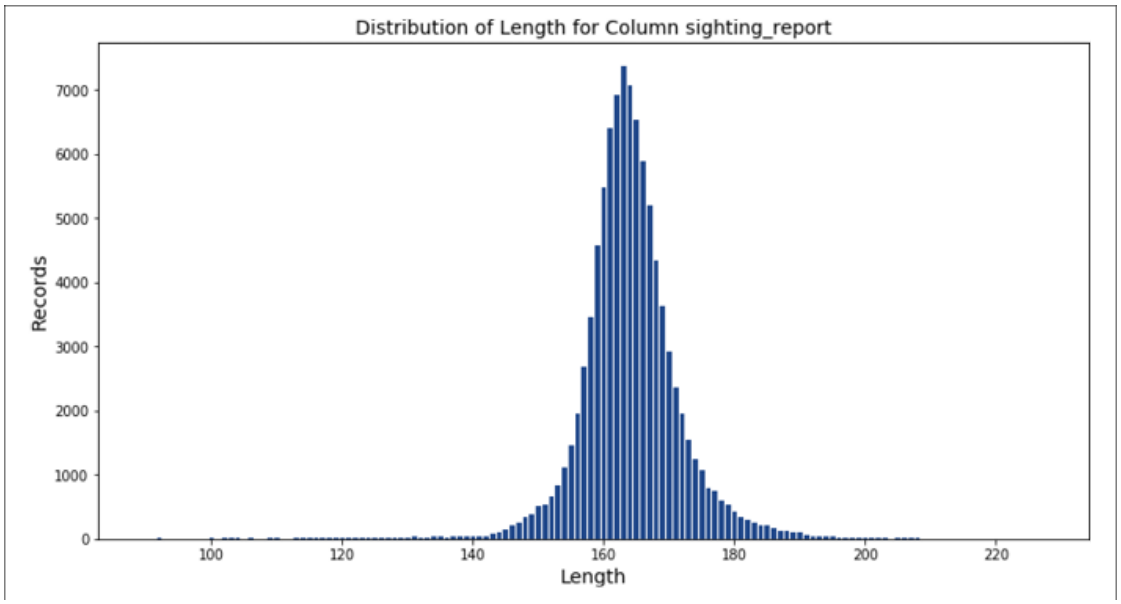


Figura 5.2: Distribuição dos tamanhos de campos da primeira coluna da tabela ufo.

Examinaremos alguns exemplos de linhas da coluna `sighting_report`. Em uma ferramenta de consulta, eu poderia rolar por cerca de uma centena de linhas para me familiarizar com o conteúdo, mas estas são boas representantes dos valores da coluna:

```
Occurred : 3/4/2018 19:07 (Entered as : 03/04/18 19:07)Reported:
  3/6/2018 7:05:12
```

```
PM 19:05Posted: 3/8/2018Location: Colorado Springs, COShape:
  LightDuration:3 minutes
Occurred : 10/16/2017 21:42 (Entered as : 10/16/2017 21:42)Reported:
  3/6/2018
5:09:47 PM 17:09Posted: 3/8/2018Location: North Dayton, OHShape:
  SphereDuration:~5 minutes
Occurred : 2/15/2018 00:10 (Entered as : 2/15/18 0:10)Reported:
  3/6/2018
6:19:54 PM 18:19Posted: 3/8/2018Location: Grand Forks, NDShape:
  SphereDuration:
5 seconds
```

Esses dados são o que eu chamaria de semiestruturados, ou sobrecarregados. Eles não podem ser usados em uma análise na forma como se encontram, mas há claramente informações distintas armazenadas aqui, e o padrão é semelhante entre as linhas. Por exemplo, cada linha tem a palavra "Occurred" seguida do que parece ser um timestamp, a palavra "Location" seguida de um local e "Duration" seguida de um período de tempo.

Os dados podem acabar ficando em campos sobrecarregados por várias razões, mas consigo detectar duas que são as mais comuns. Uma é quando não há campos suficientes disponíveis no sistema ou na aplicação de origem para armazenar todos os atributos necessários; logo, vários atributos são inseridos no mesmo campo. Outra é quando os dados são armazenados no blob JSON de uma aplicação para a acomodação de atributos esparsos ou inclusões frequentes de novos atributos. Embora os dois cenários estejam longe do ideal do ponto de vista da análise, contanto que haja uma estrutura consistente, geralmente é possível manipulá-los com SQL.

Nossa próxima etapa é tornar esse campo mais usável fazendo o parsing dele para obter vários campos novos, cada um contendo uma única informação. As etapas desse processo são:

- Planeje o(s) campo(s) que deseja obter na saída
- Aplique funções de parsing
- Aplique transformações, incluindo conversões de tipos de dados
- Verifique os resultados quando aplicados ao conjunto de dados inteiro, já que geralmente alguns registros não seguem o padrão

- Repita essas etapas até os dados estarem nas colunas e formatos desejados

As novas colunas que obteremos com o parsing de `sighting_report` são `occurred`, `entered_as`, `reported`, `posted`, `location`, `shape` e `duration`. A seguir, conheceremos as funções de parsing e trabalharemos na estruturação do conjunto de dados `ufo`.

Parsing do texto

Parsing de dados com SQL é o processo de extrair as partes de um valor textual para torná-las mais úteis para a análise. O parsing divide os dados na parte que queremos e no "restante", embora normalmente nosso código só retorne a parte que queremos.

As funções de parsing mais simples retornam um número de caracteres fixo do começo ou do fim de uma string. A função `left` retorna caracteres do lado esquerdo ou do começo da string, enquanto a função `right` retorna caracteres do lado direito ou do fim da string. Fora isso, elas funcionam da mesma forma, recebendo o valor submetido ao parsing como primeiro argumento e o número de caracteres como segundo argumento. Os dois argumentos podem ser um campo do banco de dados ou um cálculo, o que permite a obtenção de resultados dinâmicos:

```
SELECT left('The data is about UFOs',3) as left_digits
,right('The data is about UFOs',4) as right_digits
;
left_digits  right_digits
-----------  -----
The          UFOs
```

No conjunto de dados `ufo`, podemos extrair a primeira palavra, "Occurred", usando a função `left`:

```
SELECT left(sighting_report,8) as left_digits
,count(*)
FROM ufo
GROUP BY 1
;
left_digits  count
```

```
-----------  -----
Occurred     95463
```

Podemos confirmar se todos os registros começam com essa palavra, o que seria bom por significar que pelo menos essa parte do padrão é consistente. No entanto, o que queremos realmente são os valores de "Occurred", e não a palavra propriamente dita, logo tentaremos novamente. No primeiro registro do exemplo, o fim do timestamp do que ocorreu está no caractere 25. Para remover "Occurred" e reter apenas o timestamp, podemos retornar os 14 caracteres da direita usando a função `right`. Observe que as funções `right` e `left` estão aninhadas – o primeiro argumento da função `right` é o resultado da função `left`:

```
SELECT right(left(sighting_report,25),14) as occurred
FROM ufo
;
occurred
--------------
3/4/2018 19:07
10/16/2017 21:
2/15/2018 00:1
...
```

Embora esse código retorne o resultado correto para o primeiro registro, infelizmente ele não pode manipular os registros que têm valores de dois dígitos para o mês ou o dia. Poderíamos aumentar o número de caracteres retornado pelas funções `left` e `right`, mas o resultado incluiria caracteres demais para o primeiro registro.

As funções `left` e `right` são úteis para a extração de partes de tamanho fixo de uma string, como em nossa extração da palavra "Occurred", mas para padrões mais complexos, uma função chamada `split_part` é mais útil. A ideia existente por trás dessa função é a divisão de uma string em partes baseadas em um delimitador e a posterior seleção de uma parte específica. Um *delimitador* é composto de um ou mais caracteres que são usados para especificar o limite entre regiões de texto ou outros dados. O delimitador vírgula e o delimitador tabulação provavelmente são os mais comuns, já que são usados em arquivos de texto (com extensões como *.csv*, *.tsv* ou *.txt*) para indicar onde as colunas começam e terminam. No

entanto, qualquer sequência de caracteres pode ser usada, o que será útil para nossa tarefa de parsing. A forma da função é a descrita a seguir:

```
split_part(string ou nome do campo, delimitador, índice)
```

O índice é a posição do texto a ser retornado em relação ao delimitador. Logo, um índice = 1 retorna todo o texto à esquerda da primeira ocorrência do delimitador, índice = 2 retorna o texto entre a primeira e a segunda ocorrência do delimitador (ou todo o texto à direita do delimitador se este só aparecer uma vez) e assim por diante. Não há índice igual a zero, e os valores devem ser inteiros positivos:

```
SELECT split_part('This is an example of an example string'
              ,'an example'
              ,1);
split_part
----------
This is
SELECT split_part('This is an example of an example string'
              ,'an example'
              ,2);
split_part
----------
of
```

 O MySQL tem uma função `substring_index` em vez de `split_part`. O SQL Server não tem uma função `split_part`.

É bom ressaltar que os espaços existentes no texto serão retidos a menos que sejam especificados como parte do delimitador. Examinaremos como fazer o parsing dos elementos da coluna `sighting_report`. Como lembrete, um exemplo de um valor seria:

```
Occurred : 6/3/2014 23:00 (Entered as : 06/03/14 11:00)Reported:
  6/3/2014 10:33:24
PM 22:33Posted: 6/4/2014Location: Bethesda, MDShape:
  LightDuration:15 minutes
```

O valor que queremos que nossa consulta retorne é o texto existente entre "Occurred : " e " (Entered". Ou seja, queremos a string "6/3/2014 23:00". Se verificarmos o texto do exemplo, tanto "Occurred :" quanto "(Entered"

aparecem apenas uma vez. Dois pontos (:) aparecem várias vezes, tanto para separar o rótulo do valor quanto no meio dos timestamps. Isso pode tornar complicado fazer o parsing com o uso de dois pontos. O caractere de parêntese de abertura aparece apenas uma vez. Temos algumas opções para o que podemos especificar como delimitador, optando pela seleção de strings mais longas ou apenas pelos poucos caracteres necessários para a divisão da string de maneira precisa. Tendo a ser um pouco mais verboso para assegurar que obterei a parte exata que desejo, mas isso depende da situação.

Primeiro, divida `sighting_report` em "Occurred : " e veja o resultado:

```
SELECT split_part(sighting_report,'Occurred : ',2) as split_1
FROM ufo
;
split_1
------------------------------------------------------------
6/3/2014 23:00 (Entered as : 06/03/14 11:00)Reported: 6/3/2014
  10:33:24 PM
22:33Posted: 6/4/2014Location: Bethesda, MDShape: LightDuration:15
  minutes
```

Removemos o rótulo com sucesso, mas ainda temos muito texto adicional. Verificaremos o resultado ao fazer a divisão em " (Entered":

```
SELECT split_part(sighting_report,' (Entered',1) as split_2
FROM ufo
;
split_2
------------------------
Occurred : 6/3/2014 23:00
```

Agora chegamos mais perto, porém ainda temos o rótulo no resultado. Felizmente, aninhar as funções `split_part` retornará apenas o valor de data e hora desejado:

```
SELECT split_part(
     split_part(sighting_report,' (Entered',1)
     ,'Occurred : ',2) as occurred
FROM ufo
;
```

```
occurred
---------------
6/3/2014 23:00
4/25/2014 21:15
5/25/2014
```

Finalmente o resultado inclui os valores desejados. A verificação de algumas linhas adicionais mostra que os valores de dois dígitos do dia e do mês estão sendo manipulados apropriadamente, assim como as datas que não têm um valor para as horas. No entanto, alguns registros estão omitindo o valor de "Entered as"; logo, mais uma divisão é necessária para manipular os registros em que o rótulo "Reported" marca o fim da string desejada:

```
SELECT
split_part(
  split_part(
    split_part(sighting_report,' (Entered',1)
    ,'Occurred : ',2)
    ,'Reported',1) as occurred
FROM ufo
;
occurred
---------------
6/24/1980 14:00
4/6/2006 02:05
9/11/2001 09:00
...
```

Os valores mais comuns de `occurred` extraídos com o código SQL estão representados no gráfico da Figura 5.3.

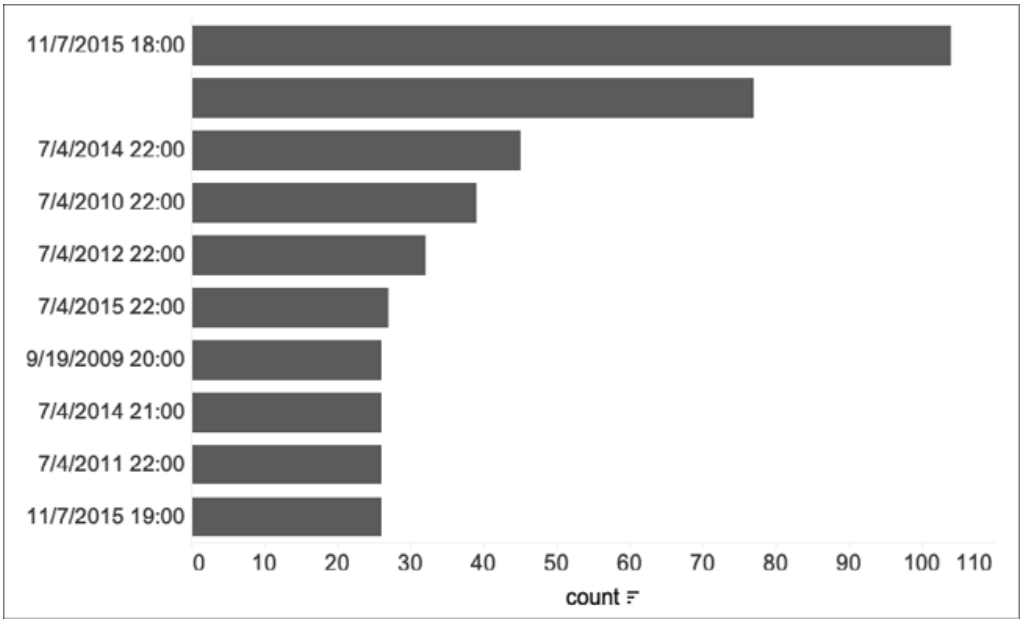


Figura 5.3: Os 10 valores mais comuns de occurred nos avistamentos de OVNIs.

Encontrar um conjunto de funções que seja adequado para todos os valores do conjunto de dados é uma das partes mais difíceis do parsing de texto. Geralmente são necessárias várias rodadas de tentativa e erro e a criação de perfis dos resultados no decorrer do processo para se chegar ao que se quer.

A próxima etapa é aplicar funções de parsing semelhantes para extrair os outros campos desejados, usando delimitadores de começo e fim para isolar apenas a parte relevante da string. A consulta final usa `split_part` várias vezes, com diferentes argumentos para cada valor:

```
SELECT
    split_part(
        split_part(
            split_part(sighting_report,' (Entered',1)
            ,'Occurred : ',2)
        ,'Reported',1) as occurred
    ,split_part(
        split_part(sighting_report,')',1)
            ,'Entered as : ',2) as entered_as
    ,split_part(
        split_part(
            split_part(
                split_part(sighting_report,'Post',1)
```

```
                'Reported: ',2)
            ,' AM',1)
        ,' PM',1) as reported
    ,split_part(split_part(sighting_report,'Location',1),'Posted:
  ',2)
      as posted
    ,split_part(split_part(sighting_report,'Shape',1),'Location:
  ',2)
      as location
    ,split_part(split_part(sighting_report,'Duration',1),'Shape:
  ',2)
      as shape
    ,split_part(sighting_report,'Duration:',2) as duration
FROM ufo
;
occurred   entered_as   reported  posted   location     shape
  duration
--------   ----------   --------  -------  -----------  --------
  -   -----------
7/4/2...   07/04/2...   7/5...    7/5/...  Columbus...  Formation
  15 minutes
7/4/2...   07/04/2...   7/5...    7/5/...  St.
  John...  Circle      2-3 minutes
7/4/2...   07/7/1...    7/5...    7/5/...  Royal
  Pa...  Circle      3 minutes
...        ...          ...       ...      ...          ...
  ...
```

Com esse parsing SQL, agora os dados estão em um formato muito mais estruturado e usável. Antes de terminarmos, entretanto, há algumas transformações que limparão um pouco mais os dados. Examinaremos essas funções de transformação de strings a seguir.

Transformações de texto

As transformações alteram de alguma forma o valor das strings. Vimos várias funções de transformação de datas e timestamps no Capítulo 3. Há um conjunto de funções em SQL que operam especificamente com o valor de strings. Elas são úteis para o trabalho com dados de parsing, mas

também para qualquer dado textual de um banco de dados que precise ser ajustado ou limpo para a análise.

Entre as transformações mais comuns estão as que alteram a capitalização. A função `upper` converte todas as letras para sua forma maiúscula, enquanto a função `lower` converte todas as letras para sua forma minúscula. Por exemplo:

```
SELECT upper('Some sample text');
upper
----------------
SOME SAMPLE TEXT
SELECT lower('Some sample text');
lower
----------------
some sample text
```

Essas funções são úteis para a padronização de valores que possam ter sido inseridos diferentemente. Por exemplo, qualquer pessoa reconhecerá que "Califórnia", "caLifórNia" e "CALIFÓRNIA" são o mesmo estado, mas um banco de dados os tratará como valores distintos. Se fôssemos contar os avistamentos de OVNIs por estado com esses valores, acabaríamos obtendo três registros para Califórnia, o que resultaria em conclusões de análise incorretas. Convertê-los para usar apenas letras maiúsculas ou minúsculas resolveria esse problema. Alguns bancos de dados, incluindo o Postgres, têm uma função `initcap` que capitaliza a primeira letra de cada palavra de uma string. Isto é útil para nomes próprios, como os nomes dos estados:

```
SELECT initcap('caLiforNia'), initcap('golden gate bridge');
initcap     initcap
----------  ------------------
California  Golden Gate Bridge
```

O campo `shape` do conjunto de dados no qual executamos o parsing contém um valor, "TRIANGULAR", que está todo em maiúsculas. Para limpá-lo e padronizá-lo de acordo com os outros valores, que têm somente a primeira letra capitalizada, aplique a função `initcap`:

```
SELECT distinct shape, initcap(shape) as shape_clean
FROM
```

```
(
    SELECT split_part(
             split_part(sighting_report,'Duration',1)
             ,'Shape: ',2) as shape
    FROM ufo
) a
;
shape       shape_clean
----------  -----------
...         ...
Sphere      Sphere
TRIANGULAR  Triangular
Teardrop    Teardrop
...         ...
```

O número de avistamentos de cada forma é mostrado na Figura 5.4. Luz (light) é sem dúvida a forma mais comum, seguida de círculo (circle) e triângulo (triangle). Alguns avistamentos não relatam uma forma; logo, uma contagem com valor nulo também aparece no gráfico.

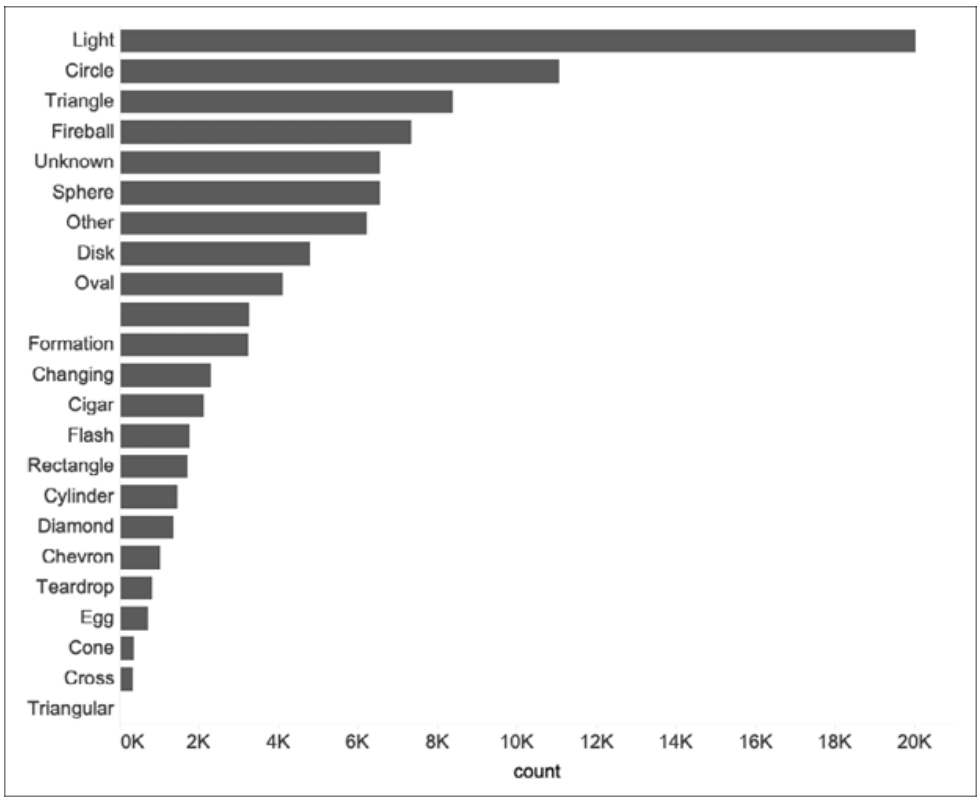


Figura 5.4: Frequência das formas nos avistamentos de OVNIs.

Outra função de transformação útil chama-se `trim`; ela remove os espaços em branco do começo e do fim de uma string. Caracteres de espaço em branco adicionais são um problema comum quando da extração de valores de strings mais longas ou quando os dados são criados por entradas fornecidas por humanos ou por cópia de uma aplicação para outra. Como exemplo, podemos remover os espaços iniciais de "California" na string a seguir usando a função `trim`:

```
SELECT trim('  California  ');
trim
----------
California
```

A função `trim` tem alguns parâmetros adicionais que a tornam flexível para resolver vários desafios de limpeza de dados. Primeiro, ela pode remover caracteres do início de uma string, do fim da string, ou de ambos. A remoção nas duas extremidades é o padrão, mas as outras opções podem ser especificadas com `leading` ou `trailing`. Além disso, `trim` pode remover qualquer caractere, e não apenas espaços em branco. Logo, por exemplo, se uma aplicação inserisse um cifrão ($) no início do nome de cada estado por alguma razão, poderíamos removê-lo com `trim`:

```
SELECT trim(leading '$' from '$California');
```

Alguns dos valores do campo `duration` têm espaços iniciais; logo, aplicar `trim` resultará em uma saída mais limpa:

```
SELECT duration, trim(duration) as duration_clean
FROM
(
    SELECT split_part(sighting_report,'Duration:',2) as duration
    FROM ufo
) a
;
duration               duration_clean
---------------------  --------------------
~2 seconds             ~2 seconds
15 minutes             15 minutes
20 minutes (ongoing)   20 minutes (ongoing)
```

O número de avistamentos com tempos de duração mais comuns está

representado no gráfico da Figura 5.5. Os avistamentos que duram entre 1 e 10 minutos são frequentes. Alguns avistamentos não relatam a duração; logo, uma contagem com valor nulo aparece no gráfico.

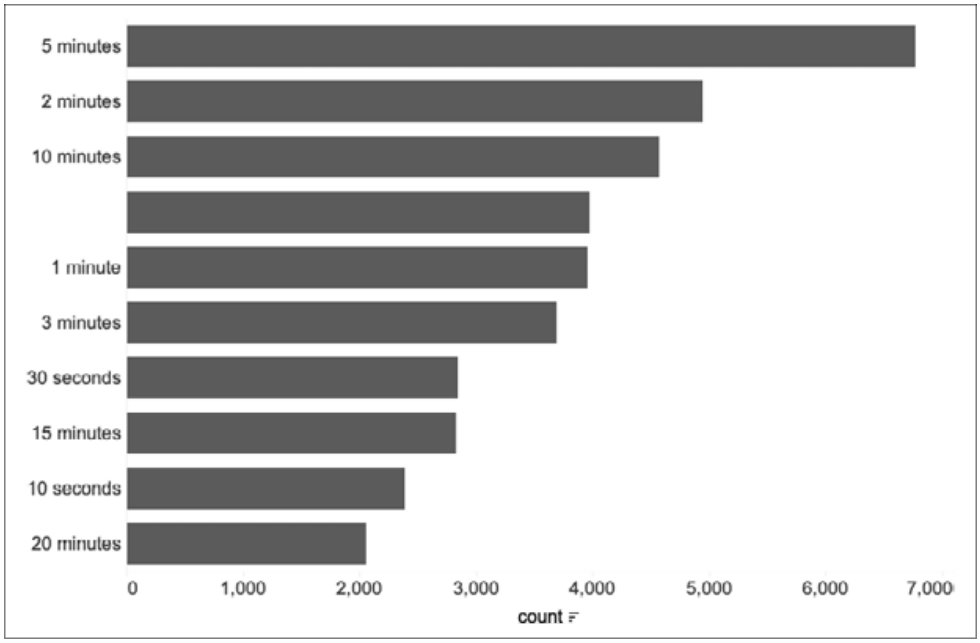


Figura 5.5: Os 10 tempos de duração mais comuns nos avistamentos de OVNIs.

O próximo tipo de transformação é a conversão de tipo de dado. Esse tipo de transformação, discutido no Capítulo 2, será útil para assegurar que os resultados de nosso parsing tenha o tipo de dado desejado. Em nosso caso, há dois campos que devem ser tratados como timestamps – as colunas `occurred` e `reported` – e a coluna `posted` precisa ter um tipo de data. Os tipos de dados podem ser alterados com uma conversão, por meio do uso do operador de dois pontos duplos (::) ou da sintaxe `CAST field as type`. Deixaremos os valores de `entered_as`, `location`, `shape` e `duration` como VARCHAR:

```
SELECT occurred::timestamp
,reported::timestamp as reported
,posted::date as posted
FROM
(
    SELECT
    split_part(
      split_part(
```

```
        split_part(sighting_report,' (Entered',1)
        ,'Occurred : ',2)
      ,'Reported',1)
      as occurred
    ,split_part(
      split_part(
        split_part(
          split_part(sighting_report,'Post',1)
          ,'Reported: ',2)
        ,' AM',1),' PM',1)
      as reported
    ,split_part(
      split_part(sighting_report,'Location',1)
      ,'Posted: ',2)
      as posted
    FROM ufo
) a
;
occurred             reported             posted
-------------------  -------------------  ----------
2015-05-24 19:30:00  2015-05-25 10:07:21  2015-05-29
2015-05-24 22:40:00  2015-05-25 09:09:09  2015-05-29
2015-05-24 22:30:00  2015-05-24 10:49:43  2015-05-29
...                  ...                  ...
```

Uma amostra dos dados foi convertida para os novos formatos. Observe que o banco de dados adiciona os segundos ao timestamp, ainda que não houvesse segundos no valor original, e reconhece corretamente datas que estavam no formato mês/dia/ano (mm/dd/aaaa).[3] No entanto, há um problema na aplicação dessas transformações ao conjunto de dados inteiro. Alguns registros não têm valores, aparecendo como uma string vazia, e outros têm o valor das horas, mas sem uma data associada. Embora uma string vazia e um valor nulo pareçam conter a mesma informação – nada – os bancos de dados os tratam diferentemente. Uma string vazia continua sendo uma string e não pode ser convertida para outro tipo de dado. Passar todos os registros que não estejam em conformidade para o valor nulo com uma instrução CASE permite que a

conversão de tipos funcione apropriadamente. Já que sabemos que as datas devem conter pelo menos oito caracteres (quatro dígitos para o ano, um ou dois dígitos tanto para o mês quanto para o dia e dois caracteres "-" ou "/"), uma maneira de garantir isso é configurando qualquer registro que tiver o tamanho menor do que 8 com nulo por meio de uma instrução CASE:

```
SELECT
case when occurred = '' then null
     when length(occurred) < 8 then null
     else occurred::timestamp
     end as occurred
,case when length(reported) < 8 then null
      else reported::timestamp
      end as reported
,case when posted = '' then null
      else posted::date
      end as posted
FROM
(
    SELECT
    split_part(
      split_part(
        split_part(sighting_report,'(Entered',1)
        ,'Occurred : ',2)
      ,'Reported',1) as occurred
    ,split_part(
      split_part(
        split_part(
          split_part(sighting_report,'Post',1)
          ,'Reported: ',2)
        ,' AM',1)
      ,' PM',1) as reported
    ,split_part(
       split_part(sighting_report,'Location',1)
       ,'Posted: ',2) as posted
    FROM ufo
) a
```

```
;
occurred             reported             posted
-------------------  -------------------  ----------
1991-10-01 14:00:00  2018-03-06 08:54:22  2018-03-08
2018-03-04 19:07:00  2018-03-06 07:05:12  2018-03-08
2017-10-16 21:42:00  2018-03-06 05:09:47  2018-03-08
...                  ...                  ...
```

A última transformação que discutirei nesta seção é a função `replace`. Poderíamos ter uma palavra, uma frase ou outra string dentro de um campo que quiséssemos alterar para outra string ou remover totalmente. A função `replace` é útil para essa tarefa. Ela recebe três argumentos – o texto original, a string a ser encontrada e a string que a substituirá:

```
replace(string ou campo, string a ser encontrada, string substituta)
```

Logo, por exemplo, se quisermos alterar as referências a "unidentified flying objects" para "UFOs", podemos usar a função `replace`:

```
SELECT replace('Some unidentified flying objects were noticed
above...','unidentified flying objects','UFOs');
replace
------------------------------
Some UFOs were noticed above...
```

Essa função encontrará e substituirá todas as ocorrências da string do segundo argumento, independentemente de onde ela aparecer. Uma string vazia pode ser usada como terceiro argumento, o que é uma boa maneira de remover partes indesejadas de uma string. Como outras funções de string, `replace` pode ser aninhada, com a saída de uma das funções tornando-se a entrada da outra.

No conjunto de dados de avistamentos de OVNIs submetido a parsing com o qual estamos trabalhando, alguns dos valores de `location` incluem qualificadores indicando que o avistamento ocorreu perto ("near"), próximo ("close to") ou fora de ("outside of") uma cidade ou município. Podemos usar `replace` para padronizar esses termos com "near":

```
SELECT location
,replace(replace(location,'close to','near')
         ,'outside of','near') as location_clean
FROM
```

```
(
    SELECT split_part(split_part(sighting_report,'Shape',1)
                      ,'Location: ',2) as location
    FROM ufo
) a
;
location                      location_clean
----------------------------  --------------------
Tombstone (outside of), AZ    Tombstone (near), AZ
Terrell (close to), TX        Terrell (near), TX
Tehachapie (outside of), CA   Tehachapie (near), CA
...                           ...
```

Os 10 principais locais de avistamentos estão representados no gráfico da Figura 5.6.

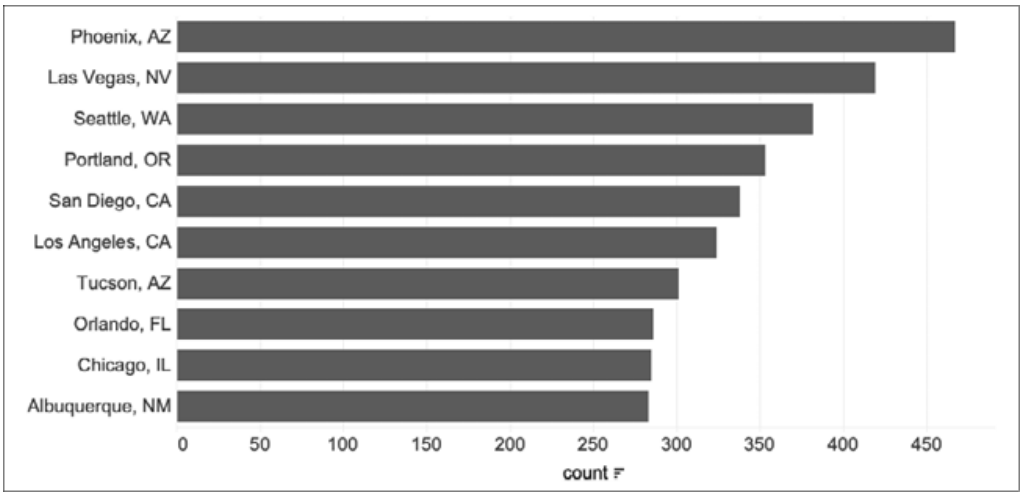


Figura 5.6: Locais mais comuns de avistamentos de OVNIs.

Fizemos o parsing e limpamos todos os elementos do campo `sighting_report` transformando-os em colunas distintas com os tipos apropriados. O código final ficou assim:

```
SELECT
case when occurred = '' then null
     when length(occurred) < 8 then null
     else occurred::timestamp
     end as occurred
,entered_as
,case when length(reported) < 8 then null
```

```
      else reported::timestamp
      end as reported
,case when posted = '' then null
      else posted::date
      end as posted
,replace(replace(location,'close to','near'),'outside of','near')
as location
,initcap(shape) as shape
,trim(duration) as duration
FROM
(
    SELECT
    split_part(
        split_part(split_part(sighting_report,' (Entered',1)
          ,'Occurred : ',2)
          ,'Reported',1) as occurred
    ,split_part(
      split_part(sighting_report,')',1)
        ,'Entered as : ',2) as entered_as
    ,split_part(
      split_part(
        split_part(
          split_part(sighting_report,'Post',1)
          ,'Reported: ',2)
        ,' AM',1)
      ,' PM',1) as reported
    ,split_part(
       split_part(sighting_report,'Location',1)
       ,'Posted: ',2) as posted
    ,split_part(
       split_part(sighting_report,'Shape',1)
       ,'Location: ',2) as location
    ,split_part(
       split_part(sighting_report,'Duration',1)
       ,'Shape: ',2) as shape
    ,split_part(sighting_report,'Duration:',2) as duration
    FROM ufo
```

```
) a
;
occurred   entered_as  reported  posted   location     shape      du
  ration
--------   ----------  --------  -------  ----------   --------   --
  --------
1988-...   8-8-198...  2018-...  2018...  Amity, ...   Unknown    4
  minutes
2018-...   07/41/1...  2018-...  2018...  Bakersf...   Triangle   15
  minutes
2018-...   08/01/1...  2018-...  2018...  Naples,...   Light      10
  seconds
...        ...         ...       ...      ...          ...        ..
  .
```

Esse fragmento de código SQL pode ser reutilizado em outras consultas ou usado para a cópia dos dados brutos dos OVNIs em uma tabela nova e limpa. Alternativamente, ele poderia ser transformado em uma view ou inserido em uma expressão de tabela comum para reutilização. O Capítulo 8 discutirá essas estratégias com mais detalhes.

Vimos como aplicar funções de parsing e de transformação para limpar e melhorar o valor da análise de dados de texto que tenham alguma estrutura. A seguir, examinaremos o outro campo do conjunto de dados de avistamentos de OVNIs, o campo de texto livre `description`, e aprenderemos como usar funções SQL para procurar elementos específicos.

Encontrando elementos dentro de blocos de texto maiores

O parsing e as transformações são operações comuns aplicadas aos dados de texto para prepará-los para a análise. Outra operação comum executada com dados de texto é a busca de strings dentro de blocos de texto maiores. Isso pode ser feito para a filtragem de resultados, a categorização de registros ou a substituição das strings procuradas por valores alternativos.

Correspondências com curingas: LIKE, ILIKE

O SQL tem várias funções para a busca de correspondência de padrões dentro de strings. O operador LIKE busca uma ocorrência do padrão especificado dentro da string. Para permitir que ele busque um padrão sem se preocupar apenas com uma correspondência exata, símbolos curinga podem ser adicionados antes, depois ou no meio do padrão. O curinga "%" encontra zero ou mais caracteres, enquanto o curinga "_" encontra exatamente um caractere. Se o objetivo for encontrar o próprio símbolo "%" ou "_", insira o símbolo de escape barra invertida ("\") na frente desse caractere:

```
SELECT 'this is an example string' like '%example%';
true
SELECT 'this is an example string' like '%abc%';
false
SELECT 'this is an example string' like '%this_is%';
true
```

O operador LIKE pode ser usado em várias cláusulas dentro da instrução SQL. Ele pode ser usado na cláusula *WHERE* para a filtragem de registros. Por exemplo, alguns narradores mencionam que estavam com o cônjuge no momento do avistamento e, portanto, poderíamos querer descobrir quantos relatos mencionam a palavra "wife". Já que queremos encontrar a string em qualquer local do texto na descrição, inseriremos o curinga "%" antes e depois de "wife":

```
SELECT count(*)
FROM ufo
WHERE description like '%wife%'
;
count
-----
6231
```

Podemos ver que mais de seis mil relatos mencionam "wife". No entanto, esse código só retornará correspondências para a string de letra minúscula. E se houver narradores que mencionaram "Wife" ou deixaram a tecla Caps Lock ativa e digitaram "WIFE"? Há duas opções para fazermos com que a busca não diferencie maiúsculas de minúsculas. Uma é transformar o campo a ser pesquisado usando a função `upper` ou `lower` discutida na seção anterior, o que produzirá o efeito de fazer com que a

busca não se importe com a capitalização já que todos os caracteres estarão em maiúsculas ou minúsculas:

```
SELECT count(*)
FROM ufo
WHERE lower(description) like '%wife%'
;
count
-----
6439
```

Outra maneira de fazer isso é com o operador ILIKE, que na verdade é um operador LIKE que não diferencia maiúsculas de minúsculas. A desvantagem é que ele não está disponível em todos os bancos de dados; principalmente o MySQL e o SQL Server não suportam esse operador. No entanto, será uma opção de sintaxe elegante e compacta se você estiver trabalhando em um banco de dados que o suporte:

```
SELECT count(*)
FROM ufo
WHERE description ilike '%wife%'
;
count
-----
6439
```

Qualquer uma dessas variações de LIKE e ILIKE pode ser invertida com NOT. Logo, por exemplo, para encontrar os registros que não mencionam "wife", podemos usar NOT LIKE:

```
SELECT count(*)
FROM ufo
WHERE lower(description) not like '%wife%'
;
count
-----
89024
```

A filtragem envolvendo várias strings pode ser feita com os operadores AND e OR:

```
SELECT count(*)
```

```
FROM ufo
WHERE lower(description) like '%wife%'
or lower(description) like '%husband%'
;
count
-----
10571
```

Tome cuidado ao usar parênteses para controlar a ordem das operações quando empregar OR junto com operadores AND ou você pode obter resultados inesperados. Por exemplo, essas cláusulas *WHERE* não retornam o mesmo resultado, já que OR é avaliado antes de AND:

```
SELECT count(*)
FROM ufo
WHERE lower(description) like '%wife%'
or lower(description) like '%husband%'
and lower(description) like '%mother%'
;
count
-----
6610
SELECT count(*)
FROM ufo
WHERE (lower(description) like '%wife%'
       or lower(description) like '%husband%'
       )
and lower(description) like '%mother%'
;
count
-----
382
```

Além de na filtragem em cláusulas *WHERE* ou *JOIN...ON*, LIKE pode ser usado na cláusula *SELECT* para categorizar ou agregar certos registros. Começaremos com a categorização. O operador LIKE pode ser usado dentro de uma instrução CASE para rotular e agrupar registros. Algumas das descrições mencionam a atividade com a qual o observador estava ocupado no momento, ou antes, do avistamento, como dirigir ou

caminhar. Podemos descobrir quantas descrições contêm esses termos usando uma instrução CASE com LIKE:

```
SELECT
case when lower(description) like '%driving%' then 'driving'
     when lower(description) like '%walking%' then 'walking'
     when lower(description) like '%running%' then 'running'
     when lower(description) like '%cycling%' then 'cycling'
     when lower(description) like '%swimming%' then 'swimming'
     else 'none' end as activity
,count(*)
FROM ufo
GROUP BY 1
ORDER BY 2 desc
;
activity  count
--------  -----
none      77728
driving   11675
walking   4516
running   1306
swimming  196
cycling   42
```

A atividade mais comum foi dirigir (walking), enquanto um número menor de pessoas relataram avistamentos ao nadar (swimming) ou andar de bicicleta (cycling). Isso já era esperado, já que essas atividades são menos comuns do que dirigir.

Embora valores derivados de funções de transformação por parsing de texto possam ser usados em critérios de *JOIN*, geralmente o desempenho do banco de dados é um problema. Considere fazer o parsing e/ou a transformação em uma subconsulta e depois fazer a junção do resultado com uma correspondência exata na cláusula *JOIN*.

Observe que essa instrução CASE rotula cada descrição com apenas uma das atividades e avalia se cada registro coincide com o padrão na ordem em que a instrução foi escrita. Uma descrição que contém tanto “driving” quanto “walking” será rotulada com “driving”. Isso é apropriado para

muitos casos, mas principalmente na análise de textos mais longos como os de revisões, comentários de pesquisas, ou tíquetes de suporte, é importante a possibilidade de rotular registros com várias categorias. Para esse tipo de caso de uso, uma série de colunas de flag binária ou BOOLEAN é necessária.

Vimos anteriormente que LIKE pode ser usado para gerar uma resposta BOOLEAN igual a TRUE ou FALSE, e podemos utilizar essa abordagem para rotular linhas. No conjunto de dados, várias descrições mencionam a direção na qual o objeto ia quando foi detectado, como norte ou sul, e algumas mencionam mais de uma direção. Poderíamos rotular cada registro com um campo indicando se a descrição menciona cada direção:

```
SELECT description ilike '%south%' as south
,description ilike '%north%' as north
,description ilike '%east%' as east
,description ilike '%west%' as west
,count(*)
FROM ufo
GROUP BY 1,2,3,4
ORDER BY 1,2,3,4
;
south  north  east   west   count
-----  -----  ----   -----  -----
false  false  false  false  43757
false  false  false  true   3963
false  false  true   false  5724
false  false  true   true   4202
false  true   false  false  4048
false  true   false  true   2607
false  true   true   false  3299
false  true   true   true   2592
true   false  false  false  3687
true   false  false  true   2571
true   false  true   false  3041
true   false  true   true   2491
true   true   false  false  3440
true   true   false  true   2064
```

```
true   true   true   false  2684
true   true   true   true   5293
```

O resultado é uma matriz de BOOLEANs que pode ser usada para sabermos a frequência com que ocorrem várias combinações de direções ou identificarmos quando uma direção é usada sem nenhuma das outras direções na mesma descrição.

Todas as combinações são úteis em contextos específicos, principalmente na construção dos conjuntos de dados que serão usados por outras pessoas que explorarão os dados ou em uma ferramenta de BI ou de virtualização. No entanto, pode ser mais útil resumir mais os dados e executar uma agregação com registros que contenham um padrão de string. Aqui contaremos os registros, mas outras agregações como `sum` e `average` podem ser usadas se o conjunto de dados tiver outros campos numéricos, como o de quantidade de vendas:

```
SELECT
count(case when description ilike '%south%' then 1 end) as south
,count(case when description ilike '%north%' then 1 end) as north
,count(case when description ilike '%west%' then 1 end) as west
,count(case when description ilike '%east%' then 1 end) as east
FROM ufo
;
south  north  west   east
-----  -----  -----  -----
25271  26027  25783  29326
```

Agora temos um resumo muito mais compacto da frequência de termos referentes à direção no campo de descrição e podemos ver que “east” (leste) é mencionado mais vezes do que outras direções. Os resultados estão representados no gráfico da Figura 5.7.

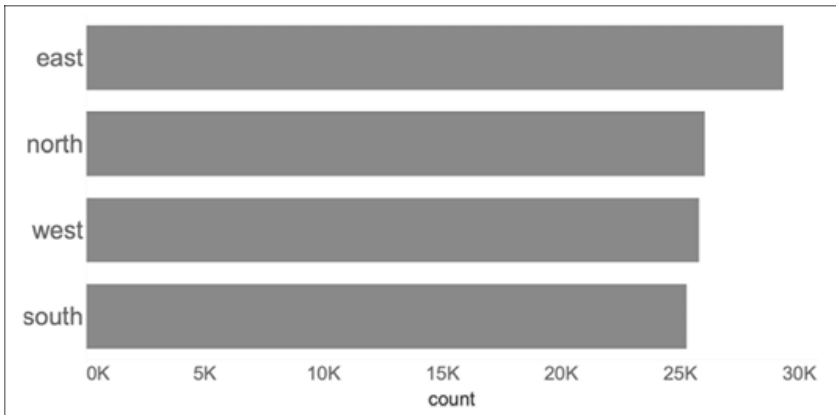


Figura 5.7: Frequência dos pontos cardeais mencionados nos relatos de avistamentos de OVNIs.

Na consulta anterior, ainda estamos permitindo que um registro que contenha mais de uma direção seja contado mais de uma vez. No entanto, não podemos mais ver as combinações específicas que existem. Quando necessário, podemos aumentar a complexidade da consulta para manipular esses casos com uma instrução como:

```
count(case when description ilike '%east%'
and description ilike '%north%' then 1 end) as east
```

A correspondência de padrões com LIKE, NOT LIKE e ILIKE é flexível e pode ser usada em vários locais de uma consulta SQL para filtrar, categorizar e agregar dados de modo a atender a diferentes necessidades de apresentação da saída. Esses operadores podem ser utilizados junto com as funções de parsing e transformação de texto que discutimos anteriormente para fornecer ainda mais flexibilidade. A seguir, discutiremos a manipulação de múltiplos elementos quando as correspondências são exatas antes de voltarmos a ver mais padrões em uma discussão sobre expressões regulares.

Correspondências exatas: IN, NOT IN

Antes de passarmos para uma correspondência de padrões mais complexa com as expressões regulares, seria interessante examinar dois operadores adicionais que são úteis na análise de texto. Embora não estejam estritamente relacionados à correspondência de padrões, eles costumam ser úteis na combinação com LIKE e seus pares para a criação de um grupo de regras que inclua o conjunto de resultados definitivamente

certo. Os operadores são IN e seu oposto, NOT IN. Eles permitem especificar uma lista de correspondências, resultando em um código mais compacto.

Suponhamos que quiséssemos categorizar os avistamentos de acordo com a primeira palavra da descrição. Podemos encontrar a primeira palavra usando a função `split_part`, com um caractere de espaço como delimitador. Muitos relatos começam com uma cor como primeira palavra. Poderíamos filtrar os registros para examinar os relatos que começam citando uma cor. Isso pode ser feito pela listagem de cada cor com uma estrutura OR:

```
SELECT first_word, description
FROM
(
    SELECT split_part(description,' ',1) as first_word
    ,description
    FROM ufo
) a
WHERE first_word = 'Red'
or first_word = 'Orange'
or first_word = 'Yellow'
or first_word = 'Green'
or first_word = 'Blue'
or first_word = 'Purple'
or first_word = 'White'
;
first_word  description
----------  ----------------------------------------------------
Blue        Blue Floating LightSaw blue light hovering...
White       White dot of light traveled across the sky, very...
Blue        Blue Beam project known seen from the high desert...
...         ...
```

Usar uma lista IN é mais compacto e geralmente menos propenso a erro, principalmente quando existem outros elementos na cláusula *WHERE*. IN recebe uma lista separada por vírgulas para fazer a comparação. O tipo de dado dos elementos deve coincidir com o da coluna. Se o tipo de dado for numérico, os elementos devem ser números; se o tipo for textual, os

elementos devem estar entre aspas como texto (mesmo se o elemento for um número):

```
SELECT first_word, description
FROM
(
    SELECT split_part(description,' ',1) as first_word
    ,description
    FROM ufo
) a
WHERE first_word in
  ('Red','Orange','Yellow','Green','Blue','Purple','White')
;
first_word  description
----------  ----------------------------------------------------
Red         Red sphere with yellow light in middleMy Grandson...
Blue        Blue light fireball shape shifted into several...
Orange      Orange lights.Strange orange-yellow hovering not...
...         ...
```

As duas formas têm resultados idênticos, e as frequências são mostradas na Figura 5.8.

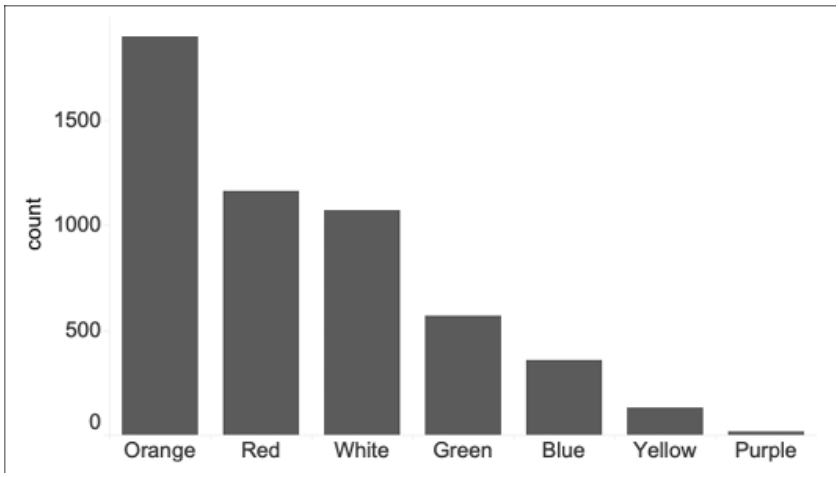


Figura 5.8: Frequência das cores de SELECT usadas como primeira palavra nas descrições de avistamentos de OVNIs.

O principal benefício dos operadores IN e NOT IN é que eles tornam o código mais compacto e legível. Isso pode ser útil para a criação de categorizações mais complexas na cláusula *SELECT*. Por exemplo,

suponhamos que quiséssemos categorizar e contar os registros pela primeira palavra de acordo com as cores, formas, movimentos ou outras palavras possíveis. Poderíamos criar algo como o código a seguir que combinasse elementos de parsing, transformações, correspondência de padrões, e listas IN:

```
SELECT
case when lower(first_word) in ('red','orange','yellow','green',
'blue','purple','white') then 'Color'
when lower(first_word) in ('round','circular','oval','cigar')
then 'Shape'
when first_word ilike 'triang%' then 'Shape'
when first_word ilike 'flash%' then 'Motion'
when first_word ilike 'hover%' then 'Motion'
when first_word ilike 'pulsat%' then 'Motion'
else 'Other'
end as first_word_type
,count(*)
FROM
(
    SELECT split_part(description,' ',1) as first_word
    ,description
    FROM ufo
) a
GROUP BY 1
ORDER BY 2 desc
;
first_word_type  count
---------------  -----
Other            85268
Color            6196
Shape            2951
Motion           1048
```

É claro que, dada a natureza desse conjunto de dados, provavelmente seriam necessárias muito mais linhas de código e regras para a categorização exata dos relatos pela primeira palavra. O SQL nos permite criar expressões complexas e mais diversificadas para lidarmos com dados

de texto. A seguir, examinaremos algumas maneiras ainda mais sofisticadas de trabalhar com dados de texto em SQL, usando expressões regulares.

Expressões regulares

Existem várias maneiras de procurar padrões em SQL. Um dos métodos mais poderosos, embora também confuso, é com o uso de expressões regulares (regex). Admito que acho as expressões regulares intimidantes, e por muito tempo evitei usá-las em minha carreira de analista de dados. Quando em apuros, felizmente eu tinha colegas que não se importavam em compartilhar trechos de código para fazer meu trabalho destravar. Só decidi que havia finalmente chegado a hora de aprender a usá-las quando recebi um grande projeto de análise de texto para executar.

Expressões regulares são sequências de caracteres, muitos deles com significados especiais, que definem padrões de busca. O principal desafio da aprendizagem de como usar a regex, e do uso e manutenção do código que a contém, é que a sintaxe não é particularmente intuitiva. Os trechos de código não se parecem em nada com a linguagem humana ou até mesmo com linguagens de computação como SQL ou Python. No entanto, se tivermos um conhecimento funcional dos caracteres especiais, o código pode ser escrito e decifrado. Como no código de todas as nossas consultas, é uma boa ideia começar com simplicidade, desenvolver complexidade quando necessário, e verificar os resultados no decorrer do processo. E escreva comentários à vontade, tanto para outros analistas quanto para você no futuro.

A regex é uma linguagem, porém ela só é usada dentro de outras linguagens. Por exemplo, as expressões regulares podem ser chamadas dentro de Java, Python e SQL, mas não há uma maneira independente de programar com elas. Todos os principais bancos de dados têm alguma implementação de regex. Nem sempre a sintaxe é a mesma, mas como ocorre com outras funções, uma vez que você conhecer as possibilidades, deve ser possível ajustar a sintaxe para o seu ambiente.

Uma explicação completa, e a apresentação de toda a sintaxe e das maneiras de usar a regex, não faz parte do escopo deste livro, mas mostrarei o suficiente para você começar a usá-la e executar várias tarefas

comuns em SQL. Se quiser ver uma introdução mais detalhada, *Learning Regular Expressions* (https://oreil.ly/5aYkb), de Ben Forta (O'Reilly), é uma boa opção. Aqui começarei introduzindo as maneiras de indicar para o banco de dados que você está usando uma regex e depois apresentarei a sintaxe, antes de passarmos para alguns exemplos de como a regex pode ser útil na análise dos relatos de avistamentos de OVNIs.

A regex pode ser usada em instruções SQL de duas maneiras. A primeira é com comparadores POSIX e a segunda com funções regex. POSIX é o acrônimo de Portable Operating System Interface, que representa um conjunto de padrões IEEE, mas você não precisa saber nada mais para usar comparadores POSIX em seu código SQL. O primeiro comparador é o símbolo ~ (til), que compara duas instruções e retorna TRUE se uma string estiver contida na outra. Como um exemplo simples, podemos verificar se a string "The data is about UFOs" contém a string "data":

```
SELECT 'The data is about UFOs' ~ 'data' as comparison;
comparison
----------
true
```

O valor de retorno é o BOOLEAN TRUE ou FALSE. Observe que, embora não contenha nenhuma sintaxe especial, "data" é uma regex. As expressões regulares também podem conter strings de texto comuns. Esse exemplo é semelhante ao que poderia ser feito com um operador LIKE. O comparador ~ diferencia maiúsculas de minúsculas. Para que ele não faça essa diferenciação, como ocorre com ILIKE, use ~* (til seguido de um asterisco):

```
SELECT 'The data is about UFOs' ~* 'DATA' as comparison;
comparison
----------
true
```

Para torná-lo um comparador de negação, insira um ! (ponto de exclamação) antes do til ou da combinação til-asterisco:

```
SELECT 'The data is about UFOs' !~ 'alligators' as comparison;
comparison
----------
true
```

A Tabela 5.1 resume os quatro comparadores POSIX.

Tabela 5.1: Comparadores POSIX

Sintaxe	O que ele faz	Diferencia maiúsculas de minúsculas?
`~`	Compara duas instruções e retorna TRUE se uma estiver contida na outra	Sim
`~*`	Compara duas instruções e retorna TRUE se uma estiver contida na outra	Não
`!~`	Compara duas instruções e retorna FALSE se uma estiver contida na outra	Sim
`!~*`	Compara duas instruções e retorna FALSE se uma estiver contida na outra	Não

Agora que temos uma maneira de introduzir a regex em nosso SQL, nos familiarizaremos com algumas das sintaxes especiais de correspondência de padrões que ela oferece. O primeiro caractere especial que precisamos conhecer é o símbolo . (ponto), um curinga que é usado na busca de qualquer caractere individual:

```
SELECT
'The data is about UFOs' ~ '. data' as comparison_1
,'The data is about UFOs' ~ '.The' as comparison_2
;
comparison_1  comparison_2
------------  ------------
true          false
```

Analisaremos por partes para entender o que está ocorrendo e melhorar nossa percepção de como a regex funciona. Na primeira comparação, o padrão tenta encontrar qualquer caractere (o que é indicado pelo ponto), um espaço e depois a palavra "data". Esse padrão encontra a string "e data" na frase do exemplo; logo, TRUE é retornado. Se parece contraintuitivo, já que há caracteres adicionais antes da letra "e" e depois da palavra "data," lembre-se de que o comparador só está procurando esse padrão dentro da string, semelhante ao que faz um operador LIKE. Na segunda comparação, o padrão tenta encontrar qualquer caractere seguido de

“The”. Já que na frase do exemplo “The” inicia a string e não há caracteres antes, o valor FALSE é retornado.

Para procurar vários caracteres, use o símbolo * (asterisco). Ele procurará zero ou mais caracteres, semelhante a quando o símbolo % (percentual) é usado em uma instrução LIKE. Esse uso do asterisco é diferente da sua inserção imediata após o til (~*), que faz com que a busca não diferencie maiúsculas de minúsculas. Observe, no entanto, que nesse caso “%” não é um curinga e é tratado como um caractere literal a ser procurado:

```
SELECT 'The data is about UFOs' ~ 'data *' as comparison_1
,'The data is about UFOs' ~ 'data %' as comparison_2
;
comparison_1  comparison_2
------------  ------------
true          false
```

Os próximos caracteres especiais que precisamos conhecer são [e] (colchetes esquerdo e direito). Eles são usados para enfeixar um conjunto de caracteres, sendo que um deles deve apresentar correspondência. Os colchetes procuram um único caractere, ainda que vários caracteres possam estar entre eles, mas veremos em breve como encontrá-lo mais de uma vez. Uma utilidade dos colchetes é fazer com que parte de um padrão não diferencie maiúsculas de minúsculas, pela inclusão das letras maiúscula e minúscula dentro deles (não use vírgula, porque isso também causaria a busca do caractere vírgula):

```
SELECT 'The data is about UFOs' ~ '[Tt]he' as comparison;
comparison
----------
true
```

Neste exemplo, o padrão procurará “the” ou “The”; já que essa string inicia a frase do exemplo, a instrução retorna o valor TRUE. Isso não é exatamente a mesma coisa que a busca com ~*, que diferencia maiúsculas de minúsculas, porque nesse caso variações como “tHe” e “THE” não coincidem com o padrão:

```
SELECT 'The data is about UFOs' ~ '[Tt]he' as comparison_1
,'the data is about UFOs' ~ '[Tt]he' as comparison_2
,'tHe data is about UFOs' ~ '[Tt]he' as comparison_3
```

```
,'THE data is about UFOs' ~ '[Tt]he' as comparison_4
;
comparison_1  comparison_2  comparison_3  comparison_4
------------  ------------  ------------  ------------
true          true          false         false
```

Outro uso da procura de correspondência com um conjunto de colchetes seria na busca de um padrão que incluísse um número, sendo permitida também a busca de qualquer número. Por exemplo, suponhamos que quiséssemos encontrar qualquer descrição que mencionasse "7 minutes", "8 minutes" ou "9 minutes". Isso poderia ser feito com uma instrução CASE com vários operadores LIKE, mas com a regex a sintaxe do padrão é mais compacta:

```
SELECT 'sighting lasted 8 minutes' ~ '[789] minutes' as comparison;
comparison
----------
true
```

Para encontrar qualquer número, poderíamos incluir todos os dígitos entre os colchetes:

```
[0123456789]
```

No entanto, a regex permite que um intervalo de caracteres seja inserido com o separador - (travessão). Podemos representar todos os números usando [0-9]. Um intervalo de números menor também pode ser usado, como [0-3] ou [4-9]. Esse padrão, com um intervalo, é equivalente ao do último exemplo, que listou cada número:

```
SELECT 'sighting lasted 8 minutes' ~ '[7-9] minutes' as comparison;
comparison
----------
true
```

Intervalos de letras podem ser procurados de maneira semelhante. A Tabela 5.2 resume os padrões de intervalo que são mais úteis na análise com SQL. Valores diferentes de números e letras também podem ser inseridos entre colchetes, como em `[$%@]`.

Tabela 5.2: Padrões de intervalo da regex

Padrão de	Finalidade

intervalo	
`[0-9]`	Encontra qualquer número
`[a-z]`	Encontra qualquer letra minúscula
`[A-Z]`	Encontra qualquer letra maiúscula
`[A-Za-z0-9]`	Encontra qualquer letra maiúscula ou minúscula, ou qualquer número
`[A-z]`	Encontra qualquer caractere ASCII; não costuma ser usado porque aceita qualquer coisa, inclusive símbolos

Se a correspondência de padrões desejada contiver mais de uma ocorrência de um valor ou tipo de valor específico, uma opção seria a inclusão de qualquer que seja o número de intervalos necessários, um após o outro. Por exemplo, podemos encontrar um número de três dígitos repetindo a notação de intervalo numérico três vezes:

```
SELECT 'driving on 495 south' ~ 'on [0-9][0-9][0-9]' as comparison;
comparison
----------
true
```

Outra alternativa seria usar uma das sintaxes opcionais especiais para a repetição de um padrão várias vezes. Isso pode ser útil quando você não souber exatamente quantas vezes o padrão se repetirá, mas certifique-se de verificar os resultados para garantir que não sejam retornadas acidentalmente mais ocorrências do que o desejado. Para procurar uma ou mais vezes, insira o símbolo + (adição) depois do padrão:

```
SELECT
'driving on 495 south' ~ 'on [0-9]+' as comparison_1
,'driving on 1 south' ~ 'on [0-9]+' as comparison_2
,'driving on 38east' ~ 'on [0-9]+' as comparison_3
,'driving on route one' ~ 'on [0-9]+' as comparison_4
;
comparison_1  comparison_2  comparison_3  comparison_4
------------  ------------  ------------  ------------
true          true          true          false
```

A Tabela 5.3 resume as outras opções para a indicação do número de vezes que um padrão deve ser repetido.

Tabela 5.3: Padrões de regex para a busca de um conjunto de caracteres várias vezes; em cada caso, o símbolo ou os símbolos são inseridos imediatamente após a expressão do conjunto

Símbolo	Finalidade
`+`	Procura o conjunto de caracteres uma ou mais vezes
`*`	Procura o conjunto de caracteres zero ou mais vezes
`?`	Procura o conjunto de caractere zero ou uma vez
`{ }`	Procura o conjunto de caracteres de acordo com o número de vezes especificado entre as chaves; por exemplo, `{3}` procura exatamente três vezes
`{ , }`	Procura o conjunto de caracteres qualquer número de vezes de um intervalo especificado pelos números separados por vírgulas existentes entre as chaves; por exemplo, `{3,5}` procura entre três e cinco vezes

Em vez de procurar a correspondência com um padrão, poderíamos querer encontrar os itens que *não* coincidem com o padrão. Isso pode ser feito com a inserção do símbolo ^ (circunflexo) antes do padrão, o que serve para negá-lo:

```
SELECT
'driving on 495 south' ~ 'on [0-9]+' as comparison_1
,'driving on 495 south' ~ 'on ^[0-9]+' as comparison_2
,'driving on 495 south' ~ '^on [0-9]+' as comparison_3
;
comparison_1  comparison_2  comparison_3
------------  ------------  ------------
true          false         false
```

Para procurar um padrão que inclua um dos caracteres especiais, precisamos de uma maneira de solicitar ao banco de dados que procure esse caractere literal sem tratá-lo como especial. Para fazê-lo, precisamos de um caractere de escape, que na regex é o símbolo \ (barra invertida):

```
SELECT
'"Is there a report?" she asked' ~ '\?' as comparison_1
,'it was filed under ^51.' ~ '^[0-9]+' as comparison_2
,'it was filed under ^51.' ~ '\^[0-9]+' as comparison_3
;
comparison_1  comparison_2  comparison_3
------------  ------------  ------------
```

```
true          false         true
```

Na primeira linha, omitir a barra invertida antes do ponto de interrogação faz o banco de dados retornar um erro "invalid regular expression", ou de expressão regular inválida (os termos exatos que indicam o erro podem ser diferentes dependendo do tipo de banco de dados). Na segunda linha, mesmo com `^` sendo seguido de um ou mais dígitos (`[0-9]+`), o banco de dados o interpretará na comparação `'^[0-9]+'` como uma negação e avaliará se a string não inclui os dígitos especificados. A terceira linha está escapando o acento circunflexo com uma barra invertida e agora o banco de dados o interpretará como o caractere `^` literal.

Os dados de texto costumam incluir caracteres de espaço em branco. Eles vão do espaço, que nossos olhos percebem, aos sutis e às vezes não exibidos caracteres de tabulação e nova linha. Veremos posteriormente como substituí-los com a expressão regular, mas por enquanto examinaremos como procurá-los com uma regex. As tabulações são procuradas com `\t`. Novas linhas são buscadas com `\r` para retorno de carro (carriage return) ou `\n` para line feed, e dependendo do sistema operacional, às vezes os dois são necessários: `\r\n`. Faça testes com seu ambiente executando algumas consultas simples para ver o que retorna o resultado desejado. Para procurar caracteres de espaço em branco, use \s, mas lembre-se de que o caractere de espaço também será encontrado:

```
SELECT
'spinning
flashing
and whirling' ~ '\n' as comparison_1
,'spinning
flashing
and whirling' ~ '\s' as comparison_2
,'spinning flashing' ~ '\s' as comparison_3
,'spinning' ~ '\s' as comparison_4
;
comparison_1  comparison_2  comparison_3  comparison_4
------------  ------------  ------------  ------------
true          true          true          false
```

 As ferramentas ou os analisadores de consulta SQL podem ter problemas para interpretar novas linhas digitadas diretamente neles e, portanto, podem retornar um erro. Se for esse o caso, tente copiar e colar o texto a partir da origem em vez de digitá-lo. No entanto, todas as ferramentas de consulta SQL devem ser capazes de trabalhar com novas linhas existentes em uma tabela de banco de dados.

Semelhante ao que acontece em expressões matemáticas, parênteses podem ser usados para reunir expressões que tenham de ser tratadas juntas. Por exemplo, poderíamos querer procurar um padrão um pouco complexo que se repetisse várias vezes:

```
SELECT
'valid codes have the form 12a34b56c' ~ '([0-9]{2}[a-z]){3}'
  as comparison_1
,'the first code entered was 123a456c' ~ '([0-9]{2}[a-z]){3}'
  as comparison_2
,'the second code entered was 99x66y33z' ~ '([0-9]{2}[a-z]){3}'
  as comparison_3
;
comparison_1  comparison_2  comparison_3
------------  ------------  ------------
true          false          true
```

Todas as três linhas usam o mesmo padrão de regex, `'([0-9]{2}[a-z]){3}'`, para a busca. O padrão dentro dos parênteses, `[0-9]{2}[a-z]`, procura dois dígitos seguidos de uma letra minúscula. Fora dos parênteses, `{3}` indica que o padrão inteiro deve ser repetido três vezes. A primeira linha segue esse padrão, já que contém a string `12a34b56c`. A segunda não atende ao padrão; ela tem dois dígitos seguidos de uma letra minúscula (`23a`), depois vêm mais dois dígitos (`23a45`), mas essa segunda repetição é seguida de um terceiro dígito em vez de outra letra minúscula (`23a456`), logo não há correspondência. A terceira linha tem um padrão correspondente, `99x66y33z`.

Como acabamos de ver, a regex pode ser usada em qualquer número de combinações com outras expressões, tanto regulares quanto de texto comum, para criar um código de correspondência de padrões. Além de especificar *o que* deve coincidir com o padrão, a regex pode ser utilizada

para especificar *onde* deve ocorrer a correspondência. Você deve usar o caractere especial `\y` para procurar um padrão começando no início ou no fim de uma palavra (em alguns bancos de dados, o caractere usado pode ser `\b`). Como exemplo, suponhamos que quiséssemos encontrar a palavra "car" nos relatos de avistamentos de OVNIs. Poderíamos escrever uma expressão como esta:

```
SELECT
'I was in my car going south toward my home' ~ 'car' as comparison;
comparison
----------
true
```

Ela procura "car" na string e retorna TRUE como esperado. No entanto, examinaremos mais algumas strings do conjunto de dados, procurando a mesma expressão:

```
SELECT
'I was in my car going south toward my home' ~ 'car'
  as comparison_1
,'UFO scares cows and starts stampede breaking' ~ 'car'
  as comparison_2
,'I''m a carpenter and married father of 2.5 kids' ~ 'car'
  as comparison_3
,'It looked like a brown boxcar way up into the sky' ~ 'car'
  as comparison_4
;
comparison_1  comparison_2  comparison_3  comparison_4
------------  ------------  ------------  ------------
true          true          true          true
```

Todas essas strings também contêm o padrão "car", ainda que "scares," "carpenter" e "boxcar" não sejam exatamente o que queríamos quando procuramos menções de carros (cars). Para corrigir isso, podemos adicionar \y ao início e ao fim do padrão "car" em nossa expressão:

```
SELECT
'I was in my car going south toward my home' ~ '\ycar\y'
  as comparison_1
,'UFO scares cows and starts stampede breaking' ~ '\ycar\y'
```

```
  as comparison_2
,'I''m a carpenter and married father of 2.5 kids' ~ '\ycar\y'
  as comparison_3
,'It looked like a brown boxcar way up into the sky' ~ '\ycar\y'
  as comparison_4
;
comparison_1  comparison_2  comparison_3  comparison_4
------------  ------------  ------------  ------------
true          false         false         false
```

É claro que, nesse exemplo simples, poderíamos ter simplesmente adicionado espaços antes e depois da palavra "car" e obteríamos o mesmo resultado. O benefício do padrão é que também são encontrados casos em que ele ocorre no começo de uma string e, portanto, não há um espaço inicial:

```
SELECT 'Car lights in the sky passing over the highway' ~* '\ycar\y'
as comparison_1
,'Car lights in the sky passing over the highway' ~* ' car '
as comparison_2
;
comparison_1  comparison_2
------------  ------------
true          false
```

O padrão `'\ycar\y'` faz uma busca que não diferencia maiúsculas de minúsculas quando "Car" é a primeira palavra, mas o padrão `' car '` não funciona assim. Para procurar uma correspondência no início de uma string inteira, use o caractere especial `\A`, e para procurar no fim da string, use `\Z`:

```
SELECT
'Car lights in the sky passing over the highway' ~* '\Acar\y'
  as comparison_1
,'I was in my car going south toward my home' ~* '\Acar\y'
  as comparison_2
,'An object is sighted hovering in place over my car' ~* '\ycar\Z'
  as comparison_3
,'I was in my car going south toward my home' ~* '\ycar\Z'
  as comparison_4
```

```
;
comparison_1  comparison_2  comparison_3  comparison_4
------------  ------------  ------------  ------------
true          false         true          false
```

Na primeira linha, o padrão encontra "Car" no início da string. A segunda linha começa com "I"; logo, não há correspondência de padrões. Na terceira linha, o padrão procura "car" no fim da string e o encontra. Para concluir, na quarta linha, a última palavra é "home", portanto não há correspondência de padrões.

Se essa for a primeira vez que você trabalha com expressões regulares, podem ser necessárias algumas pesquisas e testes em seu editor SQL para que se familiarize. Não há nada melhor do que trabalhar com exemplos reais para ajudar a consolidar o aprendizado; portanto, a seguir, aplicarei expressões regulares à nossa análise de avistamentos de OVNIs e introduzirei algumas funções SQL específicas de regex.

As implementações das expressões regulares variam muito dependendo do fornecedor do banco de dados. Os operadores POSIX desta seção funcionam no Postgres e em bancos de dados derivados do Postgres como o Amazon Redshift, mas não necessariamente em outros bancos de dados.

Uma alternativa ao operador ~ é a função `rlike` ou `regexp_like` (dependendo do banco de dados). Ela têm o seguinte formato:

```
regexp_like(string, pattern, optional_parameters)
```

O primeiro exemplo desta seção seria escrito assim:

```
SELECT regexp_like('The data is about UFOs','data')
as comparison;
```

Os parâmetros opcionais controlam o tipo de correspondência, como se ela não fosse diferenciar maiúsculas de minúsculas.

Muitos desses bancos de dados têm funções adicionais não abordadas aqui, como `regexp_substr` para a busca de substrings coincidentes e `regexp_count` que encontra quantas vezes um padrão está ocorrendo. O Postgres suporta o POSIX, mas infelizmente não suporta essas outras funções. Organizações que esperam executar muita análise de texto fariam bem em selecionar um tipo de banco de dados com um conjunto robusto de funções de expressão regular.

Busca e substituição com uma regex

Na seção anterior, discutimos as expressões regulares e como construir padrões com a regex para procurar partes de strings em nossos conjuntos de dados. Aplicaremos essa técnica ao conjunto de dados de avistamentos de OVNIs para ver como ela funciona na prática. No decorrer do processo, também introduzirei algumas funções SQL adicionais de regex.

Os relatos de avistamentos contêm vários detalhes, como o que o narrador estava fazendo na hora do avistamento e quando e onde ele estava fazendo essa atividade. Outro detalhe normalmente mencionado é a observação de várias luzes. Como primeiro exemplo, encontraremos as descrições que contêm um número e a palavra "light" ou "lights". Para facilitar a exibição neste livro, verificarei apenas os 100 primeiros caracteres, mas esse código também pode percorrer o campo de descrição inteiro:

```
SELECT left(description,50)
FROM ufo
WHERE left(description,50) ~ '[0-9]+ light[s ,.]'
;
left
--------------------------------------------------
Was walking outside saw 5 lights in a line changed
2 lights about 5 mins apart, goin from west to eas
Black triangular aircraft with 3 lights hovering a
...
```

O padrão da expressão regular está procurando qualquer número de dígitos ([0-9]+), seguido de um espaço, da string "light" e, para concluir, de uma letra "s", um espaço, uma vírgula ou um ponto. Além de procurar os registros relevantes, poderíamos separar apenas as partes referentes ao número e à palavra "lights". Para fazê-lo, usaremos a função de regex `regexp_matches`.

 O suporte a funções de regex varia muito dependendo do fornecedor do banco de dados e às vezes dependendo da versão do software de banco de dados. O SQL Server não suporta as funções, enquanto o MySQL tem um suporte mínimo. Bancos de dados analíticos como o Redshift, o Snowflake e o Vertica suportam várias funções úteis. O Postgres só tem funções de correspondência e substituição. Examine a documentação de seu banco de dados para ver a disponibilidade de funções específicas.

A função `regexp_matches` recebe dois argumentos: a string a ser procurada e o padrão de correspondência da regex. Ela retorna um array com as strings que atenderam ao padrão. Se não houver correspondências, um valor nulo será retornado. Já que o valor de retorno é um array, usaremos um índice igual a [1] para retornar um único valor como VARCHAR, o que permitirá manipulações adicionais de strings quando necessário. Se você está trabalhando em outro tipo de banco de dados, a função `regexp_substr` é semelhante a `regexp_matches`, mas retorna um valor VARCHAR; logo, não há necessidade de adicionar o índice [1].

 Um *array* é uma coleção de objetos armazenados juntos na memória do computador. Nos bancos de dados, os arrays ficam entre { } (chaves), e essa é uma boa maneira de detectarmos que algo que está contido no banco de dados não tem um dos tipos de dados regulares com os quais trabalhamos até agora. Os arrays apresentam algumas vantagens para o armazenamento e a recuperação de dados, mas não são tão fáceis de manipular em SQL, já que requerem sintaxe especial. Os elementos de qualquer array podem ser acessados com o uso da notação [ ] (de colchetes). Para o que queremos aqui, é suficiente saber que o primeiro elemento é encontrado com [1], o segundo com [2] e assim por diante.

Usando nosso exemplo, podemos executar o parsing do valor desejado, que é composto do número e da palavra "light(s)", a partir do campo de descrição e depois fazer o agrupamento por esse valor e pelas variações mais comuns:

```
SELECT (regexp_matches(description,'[0-9]+ light[s ,.]'))[1]
,count(*)
FROM ufo
WHERE description ~ '[0-9]+ light[s ,.]'
GROUP BY 1
ORDER BY 2 desc
```

```
;
regexp_matches  count
--------------  -----
3 lights        1263
2 lights        565
4 lights        549
...             ...
```

Os 10 principais resultados estão representados em gráfico na Figura 5.9.

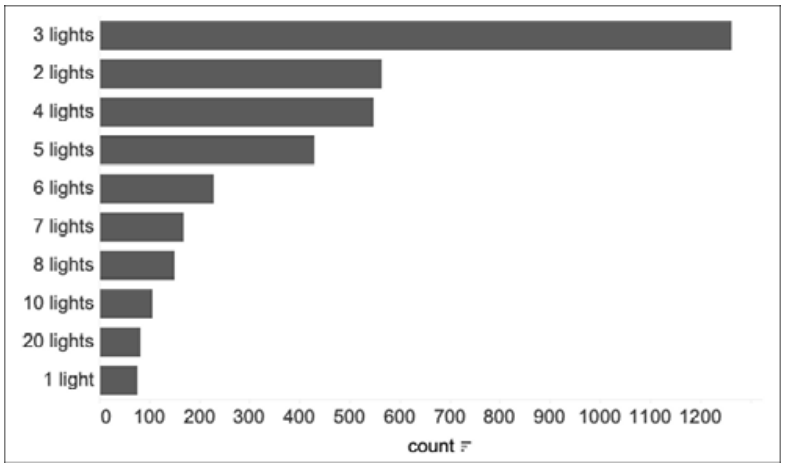


Figura 5.9: Número de luzes mencionado no começo das descrições de avistamentos de OVNIs.

Os relatos que mencionam três luzes são mais do que duas vezes mais comuns do que o segundo número de luzes mais mencionado, e os avistamentos de duas a seis luzes foram os mais observados. Para encontrar o intervalo completo do número de luzes, podemos fazer o parsing do texto que atendeu ao padrão e depois encontrar os valores `min` e `max`:

```
SELECT min(split_part(matched_text,' ',1)::int) as min_lights
,max(split_part(matched_text,' ',1)::int) as max_lights
FROM
(
    SELECT (regexp_matches(description
                           ,'[0-9]+ light[s ,.]')
                           )[1] as matched_text
    ,count(*)
    FROM ufo
```

```
    WHERE description ~ '[0-9]+ light[s ,.]'
    GROUP BY 1
) a
;
min_lights  max_lights
----------  -----
0           2000
```

Pelo menos um relato menciona duas mil luzes e o valor mínimo igual a zero também é mencionado. Poderíamos examinar melhor esses relatos para ver se há mais alguma coisa interessante ou incomum nesses valores extremos.

Além de encontrar correspondências, poderíamos substituir o texto que atendeu ao padrão por outro texto alternativo. Isso pode ser útil principalmente se quisermos tentar limpar um conjunto de dados de texto que tenha várias grafias para o mesmo item. A função `regexp_replace` pode fazer isso. Ela é semelhante à função `replace` discutida anteriormente no capítulo, mas recebe um argumento de expressão regular como padrão a ser procurado. A sintaxe é parecida com a da função `replace`:

```
regexp_replace(campo ou string, padrão, valor substituto)
```

Colocaremos essa função em ação para tentar limpar o campo `duration` da coluna `sighting_report` do qual fizemos o parsing anteriormente. Este parece ser um campo de entrada de texto livre, e há mais de oito mil valores diferentes. No entanto, a inspeção revela que existem temas comuns – a maioria se refere a alguma combinação de segundos, minutos e horas:

```
SELECT split_part(sighting_report,'Duration:',2) as duration
,count(*) as reports
FROM ufo
GROUP BY 1
;
duration    reports
--------    -------
10 minutes  4571
1 hour      1599
```

```
10 min      333
10 mins     150
>1 hour     113
...         ...
```

Dentro dessa amostra, as durações "10 minutes", "10 min" e "10 mins" representam o mesmo período de tempo, mas o banco de dados não sabe combiná-las porque as grafias usadas são um pouco diferentes. Poderíamos utilizar uma série de funções `replace` aninhadas para converter todas essas grafias diferentes. No entanto, também teríamos de levar em consideração outras variações, como as capitalizações. A regex é útil nessa situação, permitindo que criemos um código mais compacto. A primeira etapa é desenvolver um padrão que coincida com a string desejada, o que podemos fazer com a função `regexp_matches`. É uma boa ideia revisar essa etapa intermediária para verificar se estamos procurando o texto correto:

```
SELECT duration
,(regexp_matches(duration
                 ,'\m[Mm][Ii][Nn][A-Za-z]*\y')
                 )[1] as matched_minutes
FROM
(
    SELECT split_part(sighting_report,'Duration:',2) as duration
    ,count(*) as reports
    FROM ufo
    GROUP BY 1
) a
;
duration      matched_minutes
------------  ---------------
10 min.       min
10 minutes+   minutes
10 min        min
10 minutes +  minutes
10 minutes?   minutes
10 minutes    minutes
10 mins       mins
...           ...
```

Analisaremos esse código por partes. Na subconsulta, o valor de `duration` é separado do campo `sighting_report`. Em seguida, a função `regexp_matches` procura strings que coincidam com o padrão:

```
'\m[Mm][Ii][Nn][A-Za-z]*\y'
```

Esse padrão começa no início de uma palavra (`\m`) e depois procura qualquer sequência das letras "m", "i" e "n", independentemente de capitalização (`[Mm]` e assim por diante). Em seguida, ele procura zero ou mais ocorrências de qualquer outra letra minúscula ou maiúscula (`[A-Za-z]*`) e, para concluir, procura o fim de uma palavra (`\y`) para que só a palavra que contenha a variação de "minutes" seja incluída, e não o restante da string. Observe que os caracteres "+" e "?" não são procurados. Com esse padrão, agora podemos substituir todas essas variações pelo valor padronizado "min":

```
SELECT duration
,(regexp_matches(duration
                ,'\m[Mm][Ii][Nn][A-Za-z]*\y')
                )[1] as matched_minutes
,regexp_replace(duration
                ,'\m[Mm][Ii][Nn][A-Za-z]*\y'
                ,'min') as replaced_text
FROM
(
    SELECT split_part(sighting_report,'Duration:',2) as duration
    ,count(*) as reports
    FROM ufo
    GROUP BY 1
) a
;
duration      matched_minutes  replaced_text
-----------   ---------------  -------------
10 min.       min              10 min.
10 minutes+   minutes          10 min+
10 min        min              10 min
10 minutes +  minutes          10 min +
10 minutes?   minutes          10 min?
10 minutes    minutes          10 min
```

```
10 mins       mins             10 min
...           ...              ...
```

Os valores da coluna `replaced_text` estão muito mais padronizados agora. Os caracteres de ponto, sinal de adição e ponto de interrogação também poderiam ser substituídos se melhorássemos a regex. Do ponto de vista analítico, entretanto, podemos querer considerar como representar a incerteza indicada pelos sinais de adição e ponto de interrogação. As funções `regexp_replace` podem ser aninhadas para fazermos a substituição de diferentes partes ou tipos de strings. Por exemplo, podemos padronizar tanto os minutos quanto as horas:

```
SELECT duration
,(regexp_matches(duration
                ,'\m[Hh][Oo][Uu][Rr][A-Za-z]*\y')
                )[1] as matched_hour
,(regexp_matches(duration
                ,'\m[Mm][Ii][Nn][A-Za-z]*\y')
                )[1] as matched_minutes
,regexp_replace(
        regexp_replace(duration
                       ,'\m[Mm][Ii][Nn][A-Za-z]*\y'
                       ,'min')
        ,'\m[Hh][Oo][Uu][Rr][A-Za-z]*\y'
        ,'hr') as replaced_text
FROM
(
    SELECT split_part(sighting_report,'Duration:',2) as duration
    ,count(*) as reports
    FROM ufo
    GROUP BY 1
) a
;
duration             matched_hour  matched_minutes  replaced_text
-------------------  ------------  ---------------  -------------
1 Hour 15 min        Hour          min              1 hr 15 min
1 hour & 41 minutes  hour          minutes          1 hr & 41 min
1 hour 10 mins       hour          mins             1 hr 10 min
```

```
 1 hour 10 minutes    hour          minutes          1 hr 10 min
 ...                  ...           ...              ...
```

A regex das horas é semelhante à dos minutos, procurando correspondências de "hour" sem diferenciação entre maiúsculas e minúsculas no início de uma palavra, seguidas de zero ou mais caracteres alfabéticos antes do fim da palavra. As correspondências de hora e minutos intermediárias podem não ser necessárias no resultado final, mas acho útil revisá-las enquanto desenvolvo meu código SQL para evitar erros posteriormente. Provavelmente uma limpeza completa da coluna `duration` envolveria muito mais linhas de código, e seria fácil nos perdermos e introduzir um erro de digitação.

A função `regexp_replace` pode ser aninhada um número ilimitado de vezes ou combinada com a função básica `replace`. Outro uso para `regexp_replace` seria em instruções CASE, para uma substituição direcionada quando as condições da instrução forem atendidas. A regex é uma ferramenta poderosa e flexível do SQL que, como vimos, pode ser usada de várias maneiras dentro de qualquer consulta SQL.

Nesta seção, introduzi várias maneiras de procurar, encontrar e substituir elementos específicos dentro de textos mais longos, usando desde a correspondência por meio de curingas com LIKE às listas IN e a uma busca mais complexa de padrões com as regex. Todos esses recursos, junto com as funções de parsing e transformação de texto introduzidas anteriormente, nos permitem criar conjuntos de regras personalizados com qualquer que seja o nível de complexidade necessário para a manipulação dos conjuntos de dados que tivermos em mãos. No entanto, devemos nos lembrar de manter o equilíbrio entre a complexidade e a sobrecarga na manutenção. Em uma análise feita uma única vez em um conjunto de dados, pode valer a pena criar conjuntos de regras complexos que limpem perfeitamente os dados. Em um caso de geração de relatórios e monitoramento contínuos, geralmente é preferível explorar opções para que os dados cheguem limpos das fontes de dados. A seguir, examinaremos diversas maneiras de construir novas strings de texto com SQL: com o uso de constantes, de strings existentes e de strings que passaram por parsing.

Construindo e remodelando o texto

Vimos como é feito o parsing, a transformação, a busca e a substituição de elementos de strings para a execução de várias tarefas de limpeza e análise com SQL. No entanto, o SQL também pode ser usado para gerar novas combinações de texto. Nesta seção, primeiro discutiremos a *concatenação*, que permite que diferentes campos e tipos de dados sejam consolidados em um único campo. Em seguida, abordarei a alteração da forma do texto com funções que combinam múltiplas colunas em uma única linha, e também o oposto: a divisão de uma única string em várias linhas.

Concatenação

Um texto novo pode ser criado com SQL por meio da concatenação. Qualquer combinação de texto constante ou embutido em código, campos do banco de dados, e cálculos feitos nesses campos, pode ser obtida. Existem algumas maneiras de concatenar. A maioria dos bancos de dados suporta a função `concat`, que recebe como argumentos os campos ou valores a serem concatenados:

```
concat(value1, value2)
concat(value1, value2, value3...)
```

Alguns bancos de dados suportam a função `concat_ws` (concatenate with separator, concatenação com separador), que recebe o valor do separador como primeiro argumento, seguido da lista de valores a serem concatenados. Isso será útil se quisermos reunir múltiplos valores usando uma vírgula, um travessão ou um elemento semelhante para separá-los:

```
concat_ws(separator, value1, value2...)
```

Para concluir, o operador || (pipe duplo) pode ser usado em muitos bancos de dados para a concatenação de strings (o SQL Server usa o operador +):

```
value1 || value2
```

Se algum dos valores de uma concatenação for nulo, o banco de dados retornará nulo. Certifique-se de usar `coalesce` ou CASE para substituir valores nulos por um padrão se suspeitar que eles podem ocorrer.

A concatenação pode reunir um campo e uma string constante. Por

exemplo, suponhamos que quiséssemos rotular as formas como tal e adicionar a palavra "reports" à contagem de relatos de cada forma. A subconsulta faz o parsing do nome da forma a partir do campo `sighting_report` e conta o número de registros. A consulta externa concatena as formas com a string `' (shape)'` e os relatos com a string `' reports'`:

```
SELECT concat(shape, ' (shape)') as shape
,concat(reports, ' reports') as reports
FROM
(
    SELECT split_part(
                split_part(sighting_report,'Duration',1)
                ,'Shape: ',2) as shape
    ,count(*) as reports
    FROM ufo
    GROUP BY 1
) a
;
Shape             reports
----------------  ------------
Changing (shape)  2295 reports
Chevron (shape)   1021 reports
Cigar (shape)     2119 reports
...               ...
```

Também podemos combinar dois campos, opcionalmente com um separador de string. Por exemplo, poderíamos reunir os valores da forma e do local no mesmo campo:

```
SELECT concat(shape,' - ',location) as shape_location
,reports
FROM
(
    SELECT
    split_part(split_part(sighting_report,'Shape',1)
      ,'Location: ',2) as location
    ,split_part(split_part(sighting_report,'Duration',1)
       ,'Shape: ',2) as shape
```

```
    ,count(*) as reports
    FROM ufo
    GROUP BY 1,2
) a
;
shape_location           reports
-----------------------  -------
Light - Albuquerque, NM  58
Circle - Albany, OR      11
Fireball - Akron, OH     8
...                      ...
```

As 10 principais combinações estão representadas em gráfico na Figura 5.10.

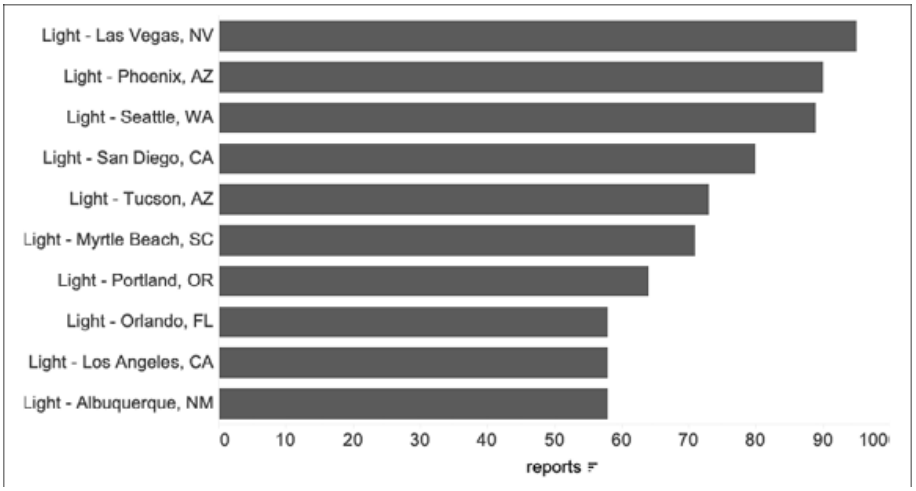


Figura 5.10: Principais combinações de forma e local nos avistamentos de OVNIs.

Vimos anteriormente que "light" é a forma mais comum, logo não é surpresa ela aparecer em cada um dos principais resultados. Phoenix é o local mais comum, enquanto Las Vegas é o segundo local mais comum no ranking geral.

Nesse caso, já que tivemos muito trabalho para executar o parsing dos diferentes campos, pode não fazer tanto sentido concatená-los novamente. No entanto, pode ser útil reorganizar o texto ou combinar valores em um único campo para exibição em outra ferramenta. Combinando vários campos e texto, também podemos gerar frases que funcionem como resumos de dados, para uso em emails ou relatórios automatizados. Neste

exemplo, a subconsulta `a` faz o parsing dos campos `occurred` e `shape`, como vimos anteriormente, e conta os registros. Em seguida, na subconsulta `aa`, os valores `min` e `max` de `occurred` são calculados, junto com o número total de relatos, e os resultados são agrupados pela forma. Linhas com campos `occurred` com menos de oito caracteres são excluídas, para removermos as que não estejam com as datas na forma apropriada e evitarmos erros nos cálculos de `min` e `max`. Para concluir, na consulta externa, o texto final é montado com a função `concat`. O formato das datas é alterado para ser exibido com datas longas (April 9, 1957) no primeiro e no último avistamento:

```
SELECT
concat('There were '
       ,reports
       ,' reports of '
       ,lower(shape)
       ,' objects. The earliest sighting was '
       ,trim(to_char(earliest,'Month'))
        , ' '
       , date_part('day',earliest)
       , ', '
       , date_part('year',earliest)
       ,' and the most recent was '
       ,trim(to_char(latest,'Month'))
        , ' '
       , date_part('day',latest)
       , ', '
       , date_part('year',latest)
       ,'.'
       )
FROM
(
    SELECT shape
    ,min(occurred::date) as earliest
    ,max(occurred::date) as latest
    ,sum(reports) as reports
    FROM
```

```
    (
        SELECT split_part(
                    split_part(
                          split_part(sighting_report,' (Entered',1)
                          ,'Occurred : ',2)
                    ,'Reported',1) as occurred
        ,split_part(
               split_part(sighting_report,'Duration',1)
               ,'Shape: ',2) as shape
        ,count(*) as reports
        FROM ufo
        GROUP BY 1,2
    ) a
    WHERE length(occurred) >= 8
    GROUP BY 1
) aa
;
concat
------------------------------------------------------------------
 -
There were 820 reports of teardrop objects. The earliest sighting
 was
April 9, 1957 and the most recent was October 3, 2020.
There were 7331 reports of fireball objects. The earliest sighting
 was
June 30, 1790 and the most recent was October 5, 2020.
There were 1020 reports of chevron objects. The earliest sighting
 was
July 15, 1954 and the most recent was October 3, 2020.
```

Poderíamos ser ainda mais criativos formatando o número de relatos ou incluindo instruções `coalesce` ou CASE para manipular nomes de formas em branco, por exemplo. Embora essas frases sejam repetitivas e, portanto, não demandem redatores humanos (ou IA), elas serão dinâmicas se a fonte de dados for atualizada com frequência e poderão ser úteis para a aplicação em relatórios.

Além das funções e operadores para a criação de texto novo com a concatenação, o SQL tem algumas funções especiais para a remodelagem

de texto, que examinaremos a seguir.

Remodelando o texto

Como vimos no Capítulo 2, às vezes é útil alterar a forma dos dados – pivotando-os de linhas para colunas ou fazendo o contrário, alterando-os de colunas para linhas. Vimos como fazer isso com *GROUP BY* e agregações, ou com instruções *UNION*. No entanto, em SQL existem algumas funções especiais para a remodelagem de texto.

Um caso de uso para o qual a remodelagem de texto pode ser útil é se existirem várias linhas com diferentes valores textuais para uma entidade e quisermos combiná-los em um único valor. É claro que a combinação dos valores pode torná-los difíceis de analisar, mas o caso de uso pode demandar um único registro por entidade na saída. A combinação dos valores individuais no mesmo campo permite reter os detalhes. A função `string_agg` recebe dois argumentos, um campo ou uma expressão, e um separador, que normalmente é uma vírgula, mas pode ser qualquer caractere separador desejado. A função só agrega valores que não sejam nulos, e a ordem pode ser controlada com uma cláusula *ORDER BY* dentro da função quando necessário:

```
SELECT location
,string_agg(shape,', ' order by shape asc) as shapes
FROM
(
    SELECT
    case when split_part(
                    split_part(sighting_report,'Duration',1)
                    ,'Shape: ',2) = '' then 'Unknown'
         when split_part(
                    split_part(sighting_report,'Duration',1)
                    ,'Shape: ',2) = 'TRIANGULAR' then 'Triangle'
         else split_part(
                    split_part(sighting_report,'Duration',1),'Shape:
  ',2)
         end as shape
    ,split_part(
            split_part(sighting_report,'Shape',1)
```

```
            ,'Location: ',2) as location
    ,count(*) as reports
    FROM ufo
    GROUP BY 1,2
) a
GROUP BY 1
;
location        shapes
--------------  ---------------------------------
Macungie, PA    Fireball, Formation, Light, Unknown
Kingsford, MI   Circle, Light, Triangle
Olivehurst, CA  Changing, Fireball, Formation, Oval
...             ...
```

Já que `string_agg` é uma função de agregação, ela requer uma cláusula *GROUP BY* nos outros campos da consulta. No MySQL, uma função equivalente é `group_concat`, e bancos de dados analíticos como o Redshift e o Snowflake têm uma função semelhante chamada `listagg`.

Outro caso de uso seria fazermos exatamente o oposto de `string_agg`, dividindo um campo individual em várias linhas. Há muita inconsistência em como isso é implementado nos diferentes bancos de dados, e até mesmo em se existe uma função para essa operação. O Postgres tem uma função chamada `regexp_split_to_table`, enquanto outros bancos de dados têm uma função `split_to_table` que opera de maneira semelhante (verifique a documentação para ver a disponibilidade e a sintaxe em seu banco de dados). A função `regexp_split_to_table` recebe dois argumentos, um valor de string e um delimitador. O delimitador pode ser uma expressão regular, mas lembre-se de que uma regex também pode ser uma string simples como um caractere de vírgula ou espaço. A função então divide os valores em linhas:

```
SELECT
regexp_split_to_table('Red, Orange, Yellow, Green, Blue, Purple'
                      ,', ');
regexp_split_to_table
--------------------
Red
Orange
```

```
Yellow
Green
Blue
Purple
```

A string a ser dividida pode incluir qualquer coisa e não precisa ser uma lista. Podemos usar a função para dividir qualquer string, inclusive frases. Logo, podemos utilizá-la para encontrar as palavras mais usadas nos campos de texto, uma ferramenta potencialmente útil para o trabalho de análise de texto. Examinaremos as palavras mais usadas nas descrições dos relatos de avistamentos de OVNIs:

```
SELECT word, count(*) as frequency
FROM
(
    SELECT regexp_split_to_table(lower(description),'\s+') as word
    FROM ufo
) a
GROUP BY 1
ORDER BY 2 desc
;
word  frequency
----  ---------
the   882810
and   477287
a     450223
```

Primeiro a subconsulta transforma a descrição para letras minúsculas, já que variações nas caixas das letras não são interessantes para esse exemplo. Em seguida, a string é dividida com o uso da regex `'\s+'`, que faz a separação em um ou mais caracteres de espaço em branco.

As palavras mais usadas não surpreendem; no entanto, elas não são muito úteis já que são apenas palavras normalmente usadas. Para obter uma lista mais significativa, podemos remover as assim chamadas *palavras vazias (stop words)*. Elas são simplesmente as palavras mais usadas em um idioma. Alguns bancos de dados têm listas internas no que eles chamam de dicionários, mas as implementações não são padrão. Também não há uma lista definitiva e unânime de palavras vazias, e é comum que um ajuste

seja feito para a criação de uma lista específica para a aplicação desejada; contudo, existem várias listas de palavras vazias comuns na internet. Para esse exemplo, carreguei uma lista de 421 palavras comuns em uma tabela que chamei de `stop_words`, disponível no *site do GitHub* referente ao livro (*https://oreil.ly/94jIS*). As palavras vazias são removidas do conjunto de resultados com uma *LEFT JOIN* para a tabela `stop_words`, sendo filtradas para resultados que não estejam nessa tabela:

```
SELECT word, count(*) as frequency
FROM
(
    SELECT regexp_split_to_table(lower(description),'\s+') as word
    FROM ufo
) a
LEFT JOIN stop_words b on a.word = b.stop_word
WHERE b.stop_word is null
GROUP BY 1
ORDER BY 2 desc
;
word    frequency
------  ---------
light   97071
lights  89537
object  80785
...     ...
```

A 10 palavras mais comuns estão representadas em gráfico na Figura 5.11.

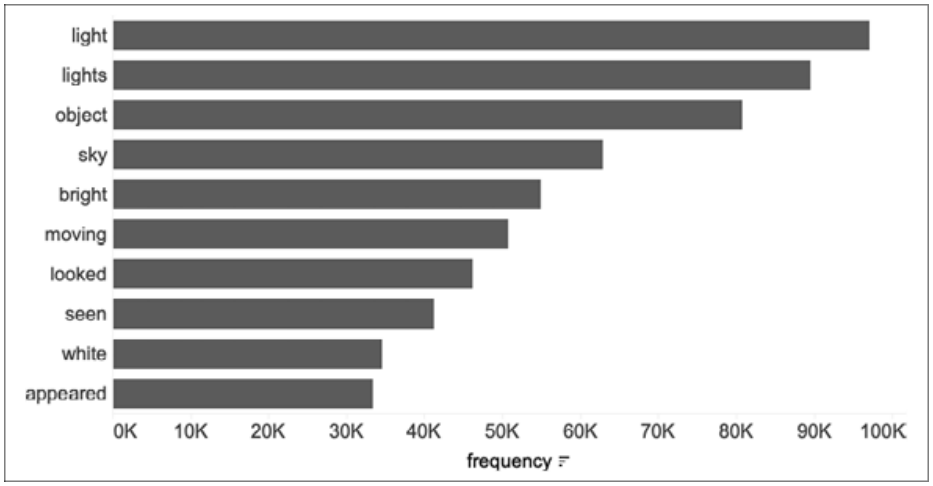


Figura 5.11: Palavras mais comuns nas descrições de avistamentos de OVNIs, sem contar as palavras vazias.

Poderíamos continuar trabalhando para obter algo mais sofisticado com a inclusão de mais palavras comuns na tabela `stop_words` ou fazendo a junção dos resultados com as descrições para marcá-las com as palavras interessantes que elas contêm. É bom ressaltar que `regexp_split_to_table` e funções semelhantes de outros bancos de dados podem ser lentas, dependendo do tamanho e da quantidade dos registros analisados.

A construção e a remodelagem de texto com SQL podem ser feitas de maneira simples ou complexa, dependendo da necessidade. Funções de concatenação, agregação de strings e divisão de strings podem ser usadas individualmente, em conjunto e com outras funções e operadores SQL para a obtenção da saída de dados desejada.

Conclusão

Embora sem sempre o SQL seja a primeira ferramenta mencionada quando se trata de análise de texto, ele tem muitas funções e operadores poderosos para a execução de várias tarefas. Usando desde parsing e transformações, à busca e substituição e também à construção e remodelagem de texto, o SQL pode ser usado tanto para limpar quanto para preparar dados de texto, assim como na execução da análise.

No próximo capítulo, examinaremos o uso do SQL para a detecção de anomalias, outro tópico em que nem sempre o SQL é a primeira ferramenta mencionada, mas para o qual ele tem recursos surpreendentes.

---

1 N.T.: De um modo geral, podemos definir flat file como um arquivo no formato de texto plano (plain text) composto de linhas e colunas. Para indicar o início e o fim das colunas, podemos utilizar caracteres delimitadores ou determinar uma largura fixa.

2 N.T.: O *pipeline ETL* é um conjunto de processos que incluem extrair dados de uma fonte e transformá-los.

3 Já que o conjunto de dados foi criado nos Estados Unidos, ele está no formato mm/dd/aaaa. Em outros países o formato dd/mm/aaaa é usado. É sempre útil verificar o país de origem e ajustar o código quando necessário.

CAPÍTULO 6

Detecção de anomalias

Uma *anomalia* é algo que é diferente de outros membros do mesmo grupo. No caso dos dados, uma anomalia é um registro, uma observação ou um valor que difere dos outros pontos de dados de uma maneira que gera dúvidas ou suspeitas. As anomalias recebem vários nomes, que incluem valores discrepantes (outliers), *novidades*, *ruídos*, *desvios* e *exceções*, para citar alguns. Usarei os termos *anomalia* e *valor discrepante* de maneira intercambiável no decorrer deste capítulo, e talvez você também veja os outros termos usados em discussões desse tópico. A detecção de anomalias pode ser o objetivo final de uma análise ou uma etapa dentro de um projeto de análise mais amplo.

Normalmente as anomalias têm uma entre duas fontes: eventos reais que são extremos ou incomuns, ou erros introduzidos durante a coleta ou o processamento dos dados. Embora muitas das etapas usadas na detecção de valores discrepantes sejam as mesmas independentemente da fonte, a maneira de escolhermos como manipular uma anomalia específica vai depender da causa raiz. Como resultado, conhecer a causa raiz e diferenciar os dois tipos de causas é importante para o processo de análise.

Eventos reais podem gerar valores discrepantes por várias razões. Dados anômalos podem indicar fraude, invasão da rede, defeitos estruturais em um produto, brechas em políticas, ou uso de um produto não pretendido ou planejado pelos desenvolvedores. A detecção de anomalias é muito usada na erradicação de fraudes financeiras, e a cibersegurança também faz uso desse tipo de análise. Às vezes os dados anômalos não ocorrem porque alguém mal-intencionado está tentando explorar um sistema, e sim porque um cliente está usando um produto de maneira inesperada. Por exemplo, conheci uma pessoa que usava uma aplicação de monitoramento de atividade física, que tinha sido projetada para corridas, andar de bicicleta, caminhadas e atividades semelhantes, para registrar

dados de suas corridas em uma pista automotiva. Ela não encontrou uma opção melhor e não pensou no quanto os valores da velocidade e da distância de um carro em uma pista seriam anômalos em comparação com os registrados para corridas de bicicleta ou a pé. Quando as anomalias puderem ser rastreadas em um processo real, decidir o que fazer com elas vai exigir um bom conhecimento da análise a ser feita, da área afetada, de regras de utilização e às vezes do sistema jurídico que regula o produto.

Os dados também podem conter anomalias devido a erros na coleta ou no processamento. Sabemos que dados inseridos manualmente geram erros de digitação e dados incorretos. Alterações em formulários, campos, ou regras de validação podem introduzir valores inesperados, inclusive valores nulos. É comum fazermos o rastreamento do comportamento de aplicações web e móveis; no entanto, qualquer alteração em como e quando esse logging é obtido pode introduzir anomalias. Passei tantas horas diagnosticando alterações em métricas que aprendi que devia perguntar antecipadamente se algum logging tinha sido alterado recentemente. O processamento dos dados pode introduzir valores discrepantes quando alguns valores são filtrados erroneamente, as etapas do processamento não são concluídas, ou os dados são carregados várias vezes, criando duplicidades. Quando as anomalias são geradas pelo processamento dos dados, geralmente podemos ficar mais confiantes ao corrigir ou descartar esses valores. É claro que corrigir a entrada ou o processamento de dados em uma etapa inicial é sempre uma boa ideia, se possível, para evitar futuros problemas de qualidade.

Neste capítulo, primeiro discutiremos algumas das razões para o uso do SQL para esse tipo de análise e situações nas quais ele não tem um desempenho tão bom. Em seguida, introduzirei o conjunto de dados de terremotos que será usado nos exemplos do restante do capítulo. Apresentarei então as ferramentas básicas que temos à nossa disposição em SQL para a detecção de valores discrepantes. Depois discutirei os diversos tipos de valores discrepantes que ao aplicar as ferramentas podemos encontrar. Uma vez que detectarmos e entendermos as anomalias, a próxima etapa é decidir o que fazer com elas. Nem sempre as anomalias são problemáticas, como o são na detecção de fraudes, na

identificação de ciberataques e no monitoramento de sistemas de saúde. As técnicas deste capítulo também podem ser usadas para detectar clientes ou campanhas de marketing anormalmente bons ou mudanças positivas no comportamento de clientes. O objetivo da detecção pode ser passar as anomalias adiante para que outras pessoas ou máquinas lidem com elas, mas geralmente essa é uma etapa de uma análise mais ampla; logo, encerrarei o assunto com várias opções para a correção de anomalias.

Recursos e limites do SQL para a detecção de anomalias

O SQL é uma linguagem versátil e poderosa para muitas tarefas de análise de dados, porém não é capaz de fazer tudo. Ele fornece várias vantagens para a detecção de anomalias, assim como apresenta algumas desvantagens que tornam outras linguagens ou ferramentas opções melhores para algumas tarefas.

É útil considerar o uso de SQL quando o conjunto de dados já estiver em um banco de dados, como vimos anteriormente com as análises de série temporal e de texto nos capítulos 3 e 5, respectivamente. O SQL se beneficia do poder computacional do banco de dados para fazer cálculos com muitos registros rapidamente. Principalmente no caso de grandes tabelas de dados, é demorado fazer a transferência de um banco de dados para outra ferramenta. Faz ainda mais sentido trabalhar dentro de um banco de dados quando a detecção de anomalias é uma etapa de uma análise maior que será feita em SQL. O código escrito em SQL pode ser examinado para entendermos por que registros específicos foram marcados como valores discrepantes, e o SQL continuará consistente com o passar do tempo mesmo quando os dados que estiverem fluindo para o banco de dados mudarem.

Um aspecto negativo do SQL é que ele não tem a sofisticação estatística que está disponível em pacotes desenvolvidos para linguagens como R e Python. O SQL tem várias funções estatísticas padrão, mas outros cálculos estatísticos mais complexos podem ser lentos ou intensos demais para alguns bancos de dados. Para casos de uso que demandem uma resposta muito rápida, como a detecção de fraude ou invasão, pode não ser

apropriado analisar dados em um banco de dados, já que geralmente há um atraso no carregamento dos dados, principalmente para bancos de dados analíticos. Em um fluxo de trabalho comum, o SQL é usado para a execução da análise inicial e a determinação dos valores mínimo, máximo e médio, e depois é feito o desenvolvimento de um monitoramento em tempo real com o uso de um serviço de streaming ou de armazenamentos de dados especiais de tempo real. No entanto, detectar o tipo de padrão dos valores discrepantes e, em seguida, fazer a implementação em serviços de streaming ou armazenamentos de dados especiais de tempo real pode ser uma opção. Para concluir, o código SQL é baseado em regras, como vimos no Capítulo 5. Ele é muito bom para a manipulação de um conjunto de condições ou critérios conhecido, mas não se ajusta automaticamente aos tipos de padrões inconstantes vistos com adversários que mudam rapidamente. As abordagens de machine learning, e as linguagens associadas a elas, geralmente são uma opção melhor para essas aplicações.

Agora que discutimos as vantagens do SQL e quando devemos usá-lo em vez de empregar outra linguagem ou ferramenta, vamos examinar os dados que utilizaremos nos exemplos deste capítulo antes de passar para o código propriamente dito.

Conjunto de dados

Os dados dos exemplos deste capítulo são de um conjunto de registros de todos os terremotos documentados pelo USGS (US Geological Survey, Serviço Geológico dos Estados Unidos) de 2010 a 2020. O USGS fornece os dados em vários formatos, inclusive em feeds em tempo real, em *https://earthquake.usgs.gov/earthquakes/feed*.

O conjunto de dados contém aproximadamente 1,5 milhão de registros. Cada registro representa um único terremoto e inclui informações como o timestamp, o local, a magnitude, a profundidade, e a fonte das informações. Uma amostra dos dados é mostrada na Figura 6.1. Um *dicionário de dados* completo está disponível no site do USGS (*https://oreil.ly/NjgCt*).

*	time	latitude	longitude	depth	mag	net	place	type	status
1	2011-03-11 05:46:24	38.297	142.373	29	9.1	official	2011 Great Tohoku Earthquake, Japan	earthquake	reviewed
2	2010-02-27 06:34:11	-36.122	-72.898	22.9	8.8	official	offshore Bio-Bio, Chile	earthquake	reviewed
3	2012-04-11 08:38:36	2.327	93.063	20	8.6	official	off the west coast of northern Sumatra	earthquake	reviewed
4	2015-09-16 22:54:32	-31.5729	-71.6744	22.44	8.3	us	48km W of Illapel, Chile	earthquake	reviewed
5	2013-05-24 05:44:48	54.892	153.221	598.1	8.3	us	Sea of Okhotsk	earthquake	reviewed
6	2012-04-11 10:43:10	0.802	92.463	25.1	8.2	us	off the west coast of northern Sumatra	earthquake	reviewed
7	2017-09-08 04:49:19	15.0222	-93.8993	47.39	8.2	us	101km SSW of Tres Picos, Mexico	earthquake	reviewed
8	2014-04-01 23:46:47	-19.6097	-70.7691	25	8.2	us	94km NW of Iquique, Chile	earthquake	reviewed
9	2018-08-19 00:19:40	-18.1125	-178.153	600	8.2	us	286km NNE of Ndoi Island, Fiji	earthquake	reviewed
10	2019-05-26 07:41:15	-5.8119	-75.2697	122.57	8	us	78km SE of Lagunas, Peru	earthquake	reviewed
11	2013-02-06 01:12:25	-10.799	165.114	24	8	us	76km W of Lata, Solomon Islands	earthquake	reviewed
12	2011-03-11 06:15:40	36.281	141.111	42.6	7.9	us	near the east coast of Honshu, Japan	earthquake	reviewed
13	2017-01-22 04:30:22	-6.2464	155.1718	135	7.9	us	35km WNW of Panguna, Papua New Guinea	earthquake	reviewed
14	2018-01-23 09:31:40	56.0039	-149.1658	14.06	7.9	us	280km SE of Kodiak, Alaska	earthquake	reviewed
15	2016-12-17 10:51:10	-4.5049	153.5216	94.54	7.9	us	54km E of Taron, Papua New Guinea	earthquake	reviewed
16	2014-06-23 20:53:09	51.8486	178.7352	109	7.9	us	19km SE of Little Sitkin Island, Alaska	earthquake	reviewed
17	2018-09-06 15:49:18	-18.4743	179.3502	670.81	7.9	us	102km ESE of Suva, Fiji	earthquake	reviewed
18	2016-12-08 17:38:46	-10.6812	161.3273	40	7.8	us	69km WSW of Kirakira, Solomon Islands	earthquake	reviewed
19	2016-11-13 11:02:56	-42.7373	173.054	15.11	7.8	us	54km NNE of Amberley, New Zealand	earthquake	reviewed
20	2015-05-30 11:23:02	27.8386	140.4931	664	7.8	us	189km WNW of Chichi-shima, Japan	earthquake	reviewed
21	2020-07-22 06:12:44	55.0715	-158.596	28	7.8	us	99 km SSE of Perryville, Alaska	earthquake	reviewed
22	2010-04-06 22:15:01	2.383	97.048	31	7.8	us	northern Sumatra, Indonesia	earthquake	reviewed

Figura 6.1: Amostra dos dados de terremotos.

Os terremotos são causados por deslizamentos repentinos ao longo de falhas nas placas tectônicas que existem na superfície externa da Terra. Certos locais nas bordas dessas placas vivenciam terremotos em maior quantidade e de maior magnitude do que os que ocorrem em outros locais. O conhecido Círculo de Fogo é uma região ao longo da orla do Oceano Pacífico na qual ocorrem muitos terremotos. Vários locais dentro dessa região, incluindo a Califórnia, o Alasca, o Japão e a Indonésia, aparecerão com frequência em nossa análise.

A *magnitude* mede o tamanho de um terremoto na sua origem, como avaliado por suas ondas sísmicas. Ela é registrada em escala logarítmica, o que significa que a amplitude de um terremoto de magnitude 5 é 10 vezes maior do que a de um terremoto de magnitude 4. É fascinante a forma como os terremotos são medidos, mas esse assunto que não faz parte do escopo deste livro. O *site do USGS* (*https://earthquake.usgs.gov*) é um bom ponto de partida se você quiser saber mais.

Detectando valores discrepantes

Embora a ideia de uma anomalia ou valor discrepante (outlier) – um ponto de dados que é muito diferente dos outros – pareça algo simples, encontrá-la em um conjunto de dados específico traz certos desafios. O primeiro está relacionado a sabermos quando um valor ou ponto de

dados é comum ou raro, e o segundo é definir um limite que marque os valores em cada lado dessa linha divisória. À medida que avançarmos usando os dados de `earthquakes`, criaremos perfis das profundidades e magnitudes para desenvolver um conhecimento dos valores que são normais e dos que são incomuns.

Geralmente, quanto maior ou mais completo é o conjunto de dados, mais fácil é decidir o que é realmente anômalo. Em alguns casos, podemos ter valores rotulados ou valores de referência reais ao quais recorrer. Um rótulo costuma ser uma coluna do conjunto de dados que indica se o registro é normal ou um valor discrepante. Os valores de referência reais podem ser obtidos de fontes industriais ou científicas ou de análises passadas e podem informar, por exemplo, que qualquer terremoto com magnitude maior do que 7 é uma anomalia. Em outros casos, devemos examinar os dados propriamente ditos e usar o bom senso. No restante do capítulo, presumiremos que temos um conjunto de dados suficientemente grande para fazer exatamente isso, embora, claro, existam referências externas que poderíamos consultar sobre as magnitudes típicas e extremas dos terremotos.

Nossas ferramentas para a detecção de valores discrepantes usando o conjunto de dados se enquadram em algumas categorias. Em primeiro lugar, podemos fazer uma classificação, ou usar *ORDER BY*, pelos valores dos dados. Opcionalmente isso pode ser combinado com várias cláusulas *GROUP BY* para encontrarmos os valores discrepantes pela frequência. Em segundo lugar, podemos usar as funções estatísticas do SQL para encontrar valores extremos em cada extremidade de um intervalo de valores. Para concluir, podemos representar os dados em gráfico e inspecioná-los visualmente.

Ordenando para encontrar anomalias

Uma das ferramentas básicas que temos para encontrar valores discrepantes é a ordenação dos dados, o que é feito com a cláusula *ORDER BY*. O comportamento padrão de *ORDER BY* é a classificação ascendente (*ASC*). Para classificar na ordem descendente, adicione *DESC* após a coluna. Uma cláusula *ORDER BY* pode incluir uma ou mais colunas, e cada coluna pode ser classificada de maneira ascendente ou

descendente, independentemente das outras colunas. A classificação começa com a primeira coluna especificada. Se uma segunda coluna for especificada, os resultados da primeira classificação serão então classificados pela segunda coluna (retendo a primeira classificação) e assim por diante passando por todas as colunas da cláusula.

Já que a ordenação ocorre após o banco de dados ter calculado o restante da consulta, muitos bancos de dados permitem referenciar as colunas da consulta pelo número em vez de pelo nome. O SQL Server é uma exceção; ele requer o nome completo. Prefiro a sintaxe de numeração porque ela gera um código mais compacto, principalmente quando as colunas da consulta incluem uma sintaxe longa para cálculos ou funções.

Por exemplo, podemos classificar a tabela `earthquakes` por `mag`, a magnitude:

```
SELECT mag
FROM earthquakes
ORDER BY 1 desc
;
mag
------
(null)
(null)
(null)
...
```

Esse código retorna várias linhas de nulos. É preciso lembrar que o conjunto de dados pode conter valores nulos para a magnitude – eles próprios sendo possíveis valores discrepantes. Podemos excluir os valores nulos:

```
SELECT mag
FROM earthquakes
WHERE mag is not null
ORDER BY 1 desc
;
mag
---
9.1
```

```
8.8
8.6
8.3
```

Há apenas um valor maior do que 9 e somente mais dois valores maiores do que 8.5. Em muitos contextos, esses valores não pareceriam particularmente altos. No entanto, com algum conhecimento na área de terremotos, é possível reconhecer que, na verdade, os valores são altos e incomuns. O USGS fornece uma lista dos 20 maiores terremotos que ocorreram no mundo (*https://oreil.ly/gHUhy*). Todos eles têm magnitude maior ou igual a 8.4, e apenas cinco têm magnitude maior ou igual a 9.0, sendo que três ocorreram entre 2010 e 2020, o período de tempo abordado por nosso conjunto de dados.

Outra maneira de considerarmos se os valores são anomalias dentro de um conjunto de dados é calculando sua frequência. Podemos contar o campo `id` e fazer o agrupamento por `mag` para encontrar o número de terremotos por magnitude. Esse número será então dividido pelo número total de terremotos, que pode ser calculado com o uso de uma função de janela `sum`. Todas as funções de janela demandam uma cláusula *OVER* com uma cláusula *PARTITION BY* e/ou *ORDER BY*. Já que o denominador deve contar todos os registros, adicionei um particionamento por 1, que é uma maneira de forçar o banco de dados a usar uma função de janela, mas mesmo assim fazer a leitura na tabela inteira. Para concluir, o conjunto de resultados é ordenado pela magnitude:

```
SELECT mag
,count(id) as earthquakes
,round(count(id) * 100.0 / sum(count(id)) over (partition by 1),8)
as pct_earthquakes
FROM earthquakes
WHERE mag is not null
GROUP BY 1
ORDER BY 1 desc
;
mag  earthquakes  pct_earthquakes
---  -----------  ---------------
9.1  1            0.00006719
8.8  1            0.00006719
```

```
8.6  1           0.00006719
8.3  2           0.00013439
...  ...         ...
6.9  53          0.00356124
6.8  45          0.00302370
6.7  60          0.00403160
...  ...         ...
```

Temos apenas um terremoto para cada magnitude maior do que 8.5, mas dois terremotos registraram 8.3. Até o valor 6.9, o número de terremotos tem dois dígitos, mas eles representam um percentual muito pequeno dos dados. Em nossa investigação, também devemos verificar a outra extremidade da classificação, a dos valores menores, classificando em ordem ascendente em vez de descendente:

```
SELECT mag
,count(id) as earthquakes
,round(count(id) * 100.0 / sum(count(id)) over (partition by 1),8)
as pct_earthquakes
FROM earthquakes
WHERE mag is not null
GROUP BY 1
ORDER BY 1
;
mag   earthquakes  pct_earthquakes
---    -----------  ---------------
-9.99  258          0.01733587
-9     29           0.00194861
-5     1            0.00006719
-2.6   2            0.00013439
...    ...          ...
```

Na extremidade de valores baixos, -9.99 e -9 ocorrem com mais frequência do que o esperado. Embora não possamos calcular o logaritmo de zero ou de um número negativo, um logaritmo pode ser negativo quando o argumento é maior do que zero e menor do que 1. Por exemplo, log(0,5) é igual a aproximadamente -0,301. Os valores -9.99 e -9 representam magnitudes extremamente baixas, e isso faz pensar se terremotos tão pequenos poderiam realmente ser detectados. Dada a frequência desses

valores, suspeito que eles representem um valor desconhecido em vez de um terremoto minúsculo e, portanto, podemos considerá-los anomalias.

Além da classificação dos dados gerais, pode ser útil fazermos o agrupamento por um ou mais campos de atributos para a busca de anomalias dentro de subconjuntos de dados. Por exemplo, poderíamos verificar as magnitudes mais altas e mais baixas registradas para regiões geográficas específicas no campo `place`:

```
SELECT place, mag, count(*)
FROM earthquakes
WHERE mag is not null
and place = 'Northern California'
GROUP BY 1,2
ORDER BY 1,2 desc
;
place                mag   count
-------------------  ----  -----
Northern California  5.61
Northern California  4.73  1
Northern California  4.51  1
...                  ...   ...
Northern California  -1.1  7
Northern California  -1.2  2
Northern California  -1.6  1
```

"Northern California" (norte da Califórnia) é a região mais comum do conjunto de dados e inspecionando apenas seu subconjunto podemos ver que os valores altos e baixos não são tão extremos quanto os do conjunto de dados como um todo. Terremotos com magnitude acima de 5.0 não são incomuns no conjunto total, mas são valores discrepantes para o norte da Califórnia.

Calculando percentis e desvios-padrão para encontrar anomalias

Classificar e opcionalmente agrupar dados e depois recuperar os resultados visualmente é uma abordagem útil para a detecção de anomalias, principalmente quando os dados têm valores que são muito extremos. Sem conhecimento da área, entretanto, pode não ser óbvio que

um terremoto de magnitude 9.0 seja uma anomalia. Quantificar a extremidade dos pontos de dados adiciona outra camada de rigor à análise. Há duas maneiras de fazê-lo: com percentis ou com desvios-padrão.

Os percentis representam a proporção de valores de uma distribuição que são menores do que um valor específico. A mediana de uma distribuição é o valor em que metade da população tem um valor mais baixo e metade tem um valor mais alto. Ela é tão usada que tem sua própria função SQL, `median`, em muitos bancos de dados, mas não em todos. Outros percentis também podem ser calculados. Por exemplo, podemos encontrar o 25° percentil, em que 25% dos valores sejam mais baixos e 75% sejam mais altos, ou o 89º percentil, em que 89% dos valores sejam mais baixos e 11% sejam mais altos. Geralmente os percentis são usados em contextos acadêmicos, como nos testes padronizados, mas podem ser aplicados a qualquer área.

O SQL tem uma função de janela, `percent_rank`, que retorna o percentil de cada linha de uma partição. Como ocorre com todas as funções de janela, a direção da classificação é controlada com uma instrução *ORDER BY*. Semelhante à função `rank`, `percent_rank` não recebe nenhum argumento; ela opera em todas as linhas retornadas pela consulta. A forma básica é:

```
percent_rank() over (partition by ... order by ...)
```

Tanto *PARTITION BY* quanto *ORDER BY* são opcionais, mas a função requer algo na cláusula *OVER*, e especificar a ordem é sempre uma boa ideia. Para encontrar o percentil das magnitudes de cada terremoto para cada região, primeiro podemos calcular `percent_rank` para cada linha da subconsulta e depois contar as ocorrências de cada magnitude na consulta externa. Lembre-se de que é importante calcular `percent_rank` primeiro, antes de executar qualquer agregação, para que valores repetidos sejam levados em consideração no cálculo:

```
SELECT place, mag, percentile
,count(*)
FROM
(
    SELECT place, mag
    ,percent_rank() over (partition by place order by mag) as
```

```
  percentile
     FROM earthquakes
     WHERE mag is not null
     and place = 'Northern California'
) a
GROUP BY 1,2,3
ORDER BY 1,2 desc
;
place                mag   percentile             count
-------------------  ----  ---------------------  -----
Northern California  5.6   1.0                    1
Northern California  4.73  0.9999870597065141     1
Northern California  4.51  0.9999741194130283     1
...                  ...   ...                    ...
Northern California  -1.1  3.8820880457568775E-5  7
Northern California  -1.2  1.2940293485856258E-5  2
Northern California  -1.6  0.0                    1
```

No norte da Califórnia, o terremoto de magnitude 5.6 tem percentil igual a 1, ou 100%, o que indica que todos os outros valores são menores. O terremoto de magnitude -1.6 tem percentil igual a 0, ou seja, nenhum outro ponto de dados é menor.

Além de encontrar o percentil exato de cada linha, o SQL pode distribuir o conjunto de dados em um número especificado de buckets e retornar o bucket ao qual cada linha pertence com uma função chamada `ntile`. Por exemplo, poderíamos distribuir o conjunto de dados em 100 buckets:

```
SELECT place, mag
,ntile(100) over (partition by place order by mag) as ntile
FROM earthquakes
WHERE mag is not null
and place = 'Central Alaska'
ORDER BY 1,2 desc
;
place           mag  ntile
--------------  ----  -----
Central Alaska  5.4   100
Central Alaska  5.3   100
```

```
Central Alaska  5.2   100
...             ...   ...
Central Alaska  1.5   79
...             ...   ...
Central Alaska  -0.5  1
Central Alaska  -0.5  1
Central Alaska  -0.5  1
```

Se examinarmos os resultados de "Central Alaska", veremos que os três terremotos com magnitude maior do que 5 estão no 100º percentil, os de magnitude 1.5 estão no 79º percentil, e os valores menores iguais a -0.5 estão no primeiro percentil. Após calcular esses valores, podemos encontrar os limites de cada bucket (ntile), usando `max` e `min`. Neste exemplo, usaremos quatro percentis para simplificar a exibição, mas qualquer inteiro positivo é permitido no argumento de `ntile`:

```
SELECT place, ntile
,max(mag) as maximum
,min(mag) as minimum
FROM
(
    SELECT place, mag
    ,ntile(4) over (partition by place order by mag) as ntile
    FROM earthquakes
    WHERE mag is not null
    and place = 'Central Alaska'
) a
GROUP BY 1,2
ORDER BY 1,2 desc
;
place           ntile  maximum  minimum
--------------  -----  -------  -------
Central Alaska  4      5.4      1.4
Central Alaska  3      1.4      1.1
Central Alaska  2      1.1      0.8
Central Alaska  1      0.8      -0.5
```

O bucket de valor mais alto, 4, que representa do 75º ao 100º percentis, tem o intervalo mais amplo, que vai de 1.4 a 5.4. Por outro lado, os valores

intermediários, que representam 50 por cento dos valores e incluem os buckets 2 e 3, vão apenas de 0.8 a 1.4.

Além de encontrar o percentil ou executar a função ntile para cada linha, podemos calcular percentis específicos ao longo do conjunto de resultados inteiro de uma consulta. Para fazê-lo, use a função `percentile_cont` ou `percentile_disc`. Ambas são funções de janela, mas com uma sintaxe um pouco diferente das de outras funções de janela discutidas anteriormente porque demandam uma cláusula *WITHIN GROUP*. A forma da função é:

```
percentile_cont(numeric) within group (order by field_name) over
  (partition by field_name)
```

O argumento numérico é um valor entre 0 e 1 que representa o percentil a ser retornado. Por exemplo, 0.25 retorna o 25º percentil. A cláusula *ORDER BY* especifica o campo do qual será retornado o percentil, assim como também representa a ordem. Opcionalmente, *ASC* ou *DESC* podem ser adicionados, com *ASC* sendo o padrão, como em todas as cláusulas *ORDER BY* em SQL. A cláusula *OVER* (*PARTITION BY*...) é opcional (e alguns bancos de dados não a suportam, o que pode causar confusão; logo, verifique sua documentação se ocorrerem erros).

A função `percentile_cont` retornará um valor interpolado (calculado) correspondente ao percentil exato, mas que pode não existir no conjunto de dados. Por outro lado, a função `percentile_disc` (percentil descontínuo) retorna o valor do conjunto que estiver mais próximo do percentil solicitado. Para conjuntos de dados grandes, ou para os que apresentem valores relativamente contínuos, na prática geralmente há pouca diferença entre a saída das duas funções, mas você deve considerar qual é a mais apropriada para sua análise. Examinaremos um exemplo para ver como funciona. Calcularemos o 25º, o 50º (ou a mediana) e o 75º percentis de todas as magnitudes não nulas da região central do Alasca:

```
SELECT
percentile_cont(0.25) within group (order by mag) as pct_25
,percentile_cont(0.5) within group (order by mag) as pct_50
,percentile_cont(0.75) within group (order by mag) as pct_75
FROM earthquakes
WHERE mag is not null
```

```
and place = 'Central Alaska'
;
pct_25  pct_50  pct_75
------  ------  ------
0.8     1.1     1.4
```

A consulta retorna os percentis solicitados, resumidos ao longo do conjunto de dados. Observe que os números correspondem aos valores máximos dos buckets 1, 2 e 3 calculados no exemplo anterior. Percentis de diferentes campos podem ser calculados dentro da mesma consulta com a alteração do campo na cláusula *ORDER BY*:

```
SELECT
percentile_cont(0.25) within group (order by mag) as pct_25_mag
,percentile_cont(0.25) within group (order by depth) as pct_25_depth
FROM earthquakes
WHERE mag is not null
and place = 'Central Alaska'
;
pct_25_mag  pct_25_depth
----------  ------------
0.8         7.1
```

Quando mais campos estão presentes na consulta, ao contrário de outras funções de janela, `percentile_cont` e `percentile_disc` demandam uma cláusula *GROUP BY* no nível da consulta. Por exemplo, se quisermos considerar duas áreas do Alasca, e, portanto, incluir o campo `place`, a consulta também deve incluí-lo em *GROUP BY*, e os percentis serão calculados por área:

```
SELECT place
,percentile_cont(0.25) within group (order by mag) as pct_25_mag
,percentile_cont(0.25) within group (order by depth) as pct_25_depth
FROM earthquakes
WHERE mag is not null
and place in ('Central Alaska', 'Southern Alaska')
GROUP BY place
;
place            pct_25_mag  pct_25_depth
---------------  ----------  ------------
```

```
Central Alaska   0.8          7.1
Southern Alaska  1.2          10.1
```

Com essas funções, podemos encontrar qualquer percentil necessário para a análise. Já que o valor da mediana é calculado com frequência, vários bancos de dados implementaram uma função `median` que só tem um argumento, o campo para o qual será calculada a mediana. Essa é uma sintaxe útil e certamente muito mais simples, mas lembre-se de que podemos fazer o mesmo com `percentile_cont` se uma função `median` não estiver disponível.

As funções `percentile` e `median` podem ser lentas e usar muito poder computacional em conjuntos de dados grandes. Isso ocorre porque o banco de dados deve classificar e ordenar todos os registros, geralmente na memória. Alguns fornecedores de bancos de dados implementaram versões aproximadas das funções, como `approximate_percentile`, que são muito mais rápidas e retornam resultados bem próximos dos da função que calcula o conjunto de dados inteiro.

Encontrar os percentis ou os ntiles de um conjunto de dados nos permite adicionar alguma quantificação às anomalias. Veremos posteriormente no capítulo que esses valores também fornecem ferramentas para a manipulação de anomalias em conjuntos de dados. No entanto, já que os percentis estão sempre em uma escala entre 0 e 100, eles não dão uma ideia do quanto certos valores são incomuns. Para saber isso podemos usar funções estatísticas adicionais suportadas pelo SQL.

Para medir o quanto os valores de um conjunto de dados são extremos, podemos usar o *desvio-padrão*. O desvio-padrão mede a variação existente em um conjunto de valores. Um valor mais baixo significa menos variação, enquanto um número mais alto revela mais variação. Quando os dados estão distribuídos normalmente com base na mediana, cerca de 68% dos valores ficam dentro de +/– um desvio-padrão em relação à mediana e aproximadamente 95% ficam dentro de dois desvios-padrão. O desvio-padrão é calculado como a raiz quadrada da soma das diferenças em relação à mediana, dividida pelo número de observações:

$$\sqrt{\Sigma(x_i - \mu)^2/N}$$

Nessa fórmula, *xi* é uma observação, μ é a média de todas as observações,

$\sum$ indica que todos os valores devem ser somados e N é o número de observações. Consulte algum texto ou recurso online interessante de estatística[1] para obter mais informações sobre como o desvio-padrão é derivado.

A maioria dos bancos de dados tem três funções de desvio-padrão. A função `stddev_pop` encontra o desvio-padrão de uma população. Se o conjunto de dados representar a população inteira, como costuma ocorrer com um conjunto de dados de clientes, use `stddev_pop`. A função `stddev_samp` encontra o desvio-padrão de uma amostra e difere da fórmula anterior por fazer a divisão por N - 1 em vez de N. Isso tem o efeito de aumentar o desvio-padrão, refletindo a perda de precisão quando apenas uma amostra da população inteira é usada. A função `stddev` disponível em muitos bancos de dados é idêntica à função `stddev_samp` e pode ser usada simplesmente por ser mais curta. Se você estiver trabalhando com dados de uma amostra, como os de uma pesquisa ou de um estudo de uma população maior, use `stddev_samp` ou `stddev`. Na prática, no trabalho com conjuntos de dados maiores, geralmente há pouca diferença entre os resultados de `stddev_pop` e `stddev_samp`. Por exemplo, a tabela `earthquakes` tem 1,5 milhão de registros e os valores só divergem após cinco casas decimais:

```
SELECT stddev_pop(mag) as stddev_pop_mag
,stddev_samp(mag) as stddev_samp_mag
FROM earthquakes
;
stddev_pop_mag        stddev_samp_mag
--------------------  --------------------
1.273605805569390395  1.273606233458381515
```

Essas diferenças são tão pequenas que, na maioria dos casos de aplicação prática, não importa que função de desvio-padrão está sendo usada.

Com essa função, agora podemos calcular o número de desvios-padrão em relação à média para cada valor do conjunto de dados. Esse valor é conhecido como *z-score* e é uma maneira de padronizar os dados. Valores que estão acima da média têm z-score positivo e os que estão abaixo têm z-score negativo. A Figura 6.2 mostra como os z-scores e os desvios-padrão estão relacionados com a distribuição normal.

Para encontrar os z-scores dos terremotos, primeiro calcule a média e o desvio-padrão do conjunto de dados inteiro em uma subconsulta. Em seguida, faça a junção com o conjunto de dados usando uma *JOIN* Cartesiana, para que os valores da média e do desvio-padrão sejam reunidos em cada linha dos terremotos. Isso é feito com a sintaxe `1 = 1`, já que a maioria dos bancos de dados requer que alguma condição de *JOIN* seja especificada.

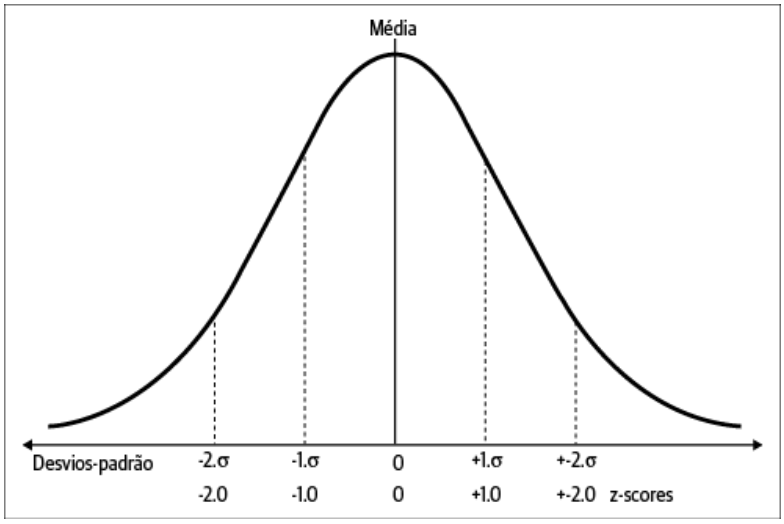


Figura 6.2: Desvios-padrão e z-scores para uma distribuição normal.

Na consulta externa, subtraia a magnitude média de cada magnitude individual e divida pelo desvio-padrão:

```
SELECT a.place, a.mag
,b.avg_mag, b.std_dev
,(a.mag - b.avg_mag) / b.std_dev as z_score
FROM earthquakes a
JOIN
(
    SELECT avg(mag) as avg_mag
    ,stddev_pop(mag) as std_dev
    FROM earthquakes
    WHERE mag is not null
) b on 1 = 1
WHERE a.mag is not null
ORDER BY 2 desc
;
place                                    mag  avg_mag  std_dev  z_sco
```

```
  re
------------------------------------  ---  -------  -------  -----
  --
2011 Great Tohoku Earthquake,
  Japan     9.1   1.6251   1.2736  5.8691
offshore Bio-Bio,
  Chile                 8.8   1.6251   1.2736  5.6335
off the west coast of northern
  Sumatra  8.6   1.6251   1.2736  5.4765
...                                     ...   ...      ...     ...
Nevada                                  -2.5  1.6251   1.2736  -3.23
  89
Nevada                                  -2.6  1.6251   1.2736  -3.31
  74
Nevada                                  -2.6  1.6251   1.2736  -3.31
  74
```

Os terremotos maiores têm um z-score que quase chega a 6, enquanto os menores (sem contar os de valores -9 e -9.99 que parecem ser anomalias na entrada de dados) têm z-scores próximos de 3. Podemos concluir que os terremotos maiores são valores discrepantes mais extremos do que os da extremidade baixa.

Representação gráfica para a busca de anomalias visualmente

Além da classificação dos dados e do cálculo de percentis e desvios-padrão para a busca de anomalias, visualizar os dados em um dos vários formatos gráficos também pode ajudar a encontrar anomalias. Como vimos em capítulos anteriores, uma vantagem dos gráficos é sua capacidade de resumir e apresentar muitos pontos de dados de forma compacta. Inspecionando os gráficos, geralmente podemos detectar padrões e valores discrepantes que talvez não víssemos se considerássemos apenas a saída bruta. Para concluir, os gráficos ajudam na tarefa de descrever os dados, e qualquer possível problema que eles tenham relacionado a anomalias, para outras pessoas.

Nesta seção, apresentarei três tipos de gráficos que são úteis para a detecção de anomalias: gráficos de barras, diagramas de dispersão e diagramas de caixa. O SQL necessário para a geração da saída desses

gráficos é simples, mas você pode ter de empregar estratégias de pivotagem discutidas em capítulos anteriores, dependendo dos recursos e das limitações do software usado para criar os gráficos. Qualquer uma das principais ferramentas de BI ou softwares de planilha, ou linguagens como Python ou R, consegue produzir esses tipos de gráficos. Os gráficos desta seção foram criados com o uso de Python com o Matplotlib.

O *gráfico de barras* é usado para exibir um histograma ou a distribuição dos valores de um campo e é útil tanto para a caracterização dos dados quanto para a detecção de valores discrepantes. O intervalo total de valores é representado ao longo de um eixo e o número de ocorrências de cada valor é exibido no outro eixo. Valores excepcionalmente altos ou baixos são de interesse, assim como a forma do diagrama. Podemos determinar rapidamente se a distribuição é aproximadamente normal (simétrica em relação a um pico ou ao valor da média), se é de outro tipo ou se há picos em valores específicos.

Para gerar um histograma das magnitudes dos terremotos, primeiro crie um conjunto de dados que agrupe as magnitudes e conte os terremotos. Em seguida, exiba a saída, como na Figura 6.3.

```
SELECT mag
,count(*) as earthquakes
FROM earthquakes
GROUP BY 1
ORDER BY 1
;
mag    earthquakes
-----  -----------
-9.99  258
-9     29
-5     1
...    ...
```

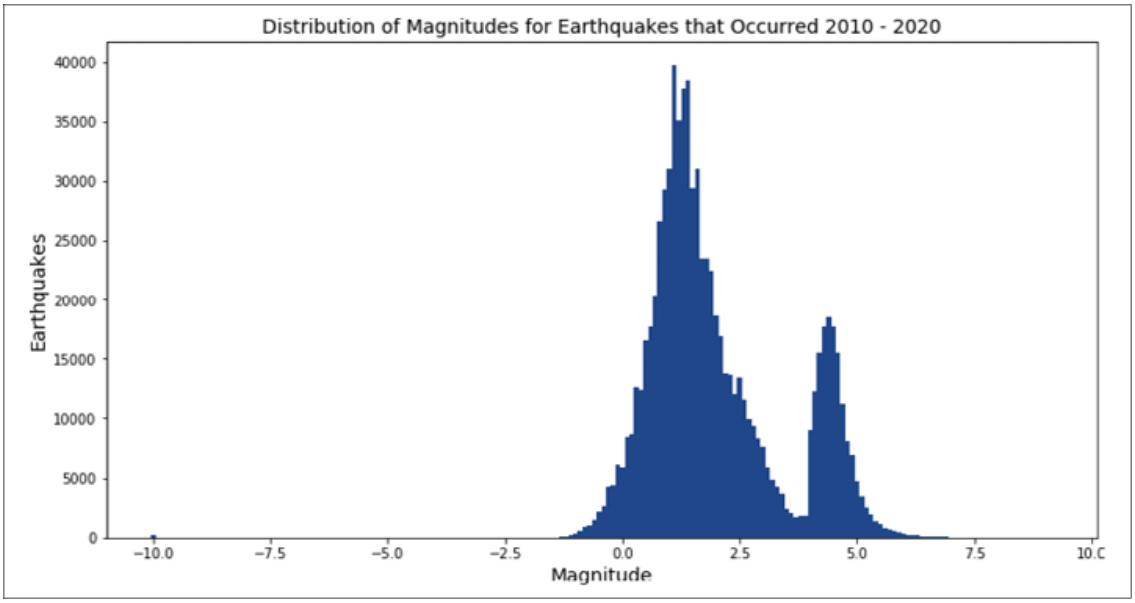


Figura 6.3: Distribuição das magnitudes dos terremotos.

O gráfico se estende de -10.0 a +10.0, o que faz sentido, dada nossa exploração anterior dos dados. Ele atinge o pico e é razoavelmente simétrico ao redor de um valor que fica no intervalo entre 1.1 a 1.4 com quase 40.000 terremotos de cada magnitude, mas apresenta um segundo pico de quase 20.000 terremotos perto do valor 4.4. Examinaremos o motivo da existência desse segundo pico na próxima seção sobre tipos de anomalias. Os valores extremos são difíceis de detectar; logo, seria útil dar um zoom em uma subseção do gráfico, como na Figura 6.4.

Aqui as frequências desses terremotos de intensidade muito alta são mais fáceis de ver, assim como a diminuição na frequência de mais de 10 para somente 1 conforme a exibição passa dos valores mais baixos de 7 para acima de 8. Felizmente esses tremores são extremamente raros.

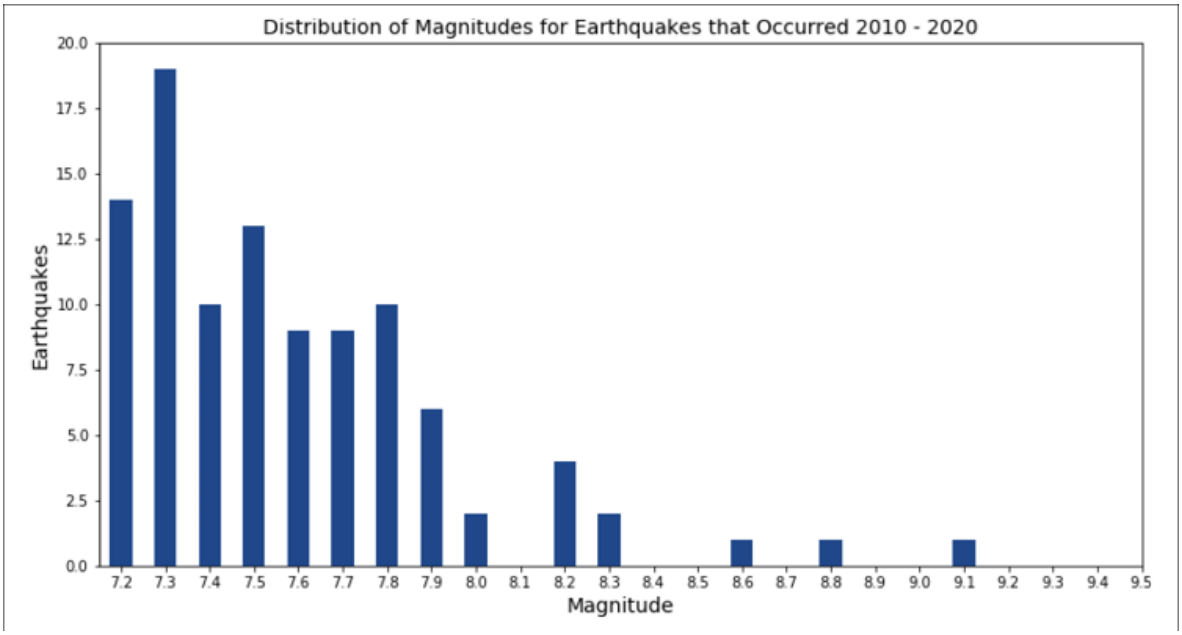


Figura 6.4: Uma visualização com zoom da distribuição das magnitudes dos terremotos, concentrada nas magnitudes mais altas.

Um segundo tipo de gráfico que pode ser usado na caracterização de dados e na identificação de valores discrepantes é o *diagrama de dispersão*. Um diagrama de dispersão é apropriado quando o conjunto de dados contém pelo menos dois valores numéricos de interesse.

O eixo x exibe o intervalo de valores do primeiro campo de dados, o eixo y exibe o intervalo de valores do segundo campo de dados e um ponto é exibido para cada par de valores x e y do conjunto de dados. Por exemplo, poderíamos representar a magnitude versus a profundidade dos terremotos no conjunto de dados. Primeiro, consulte os dados para criar um conjunto de dados de cada par de valores. Em seguida, exiba a saída, como na Figura 6.5:

```
SELECT mag, depth
,count(*) as earthquakes
FROM earthquakes
GROUP BY 1,2
ORDER BY 1,2
;
mag    depth  earthquakes
-----  -----  -----------
-9.99  -0.59  1
```

```
-9.99  -0.35  1
-9.99  -0.11  1
...    ...    ...
```

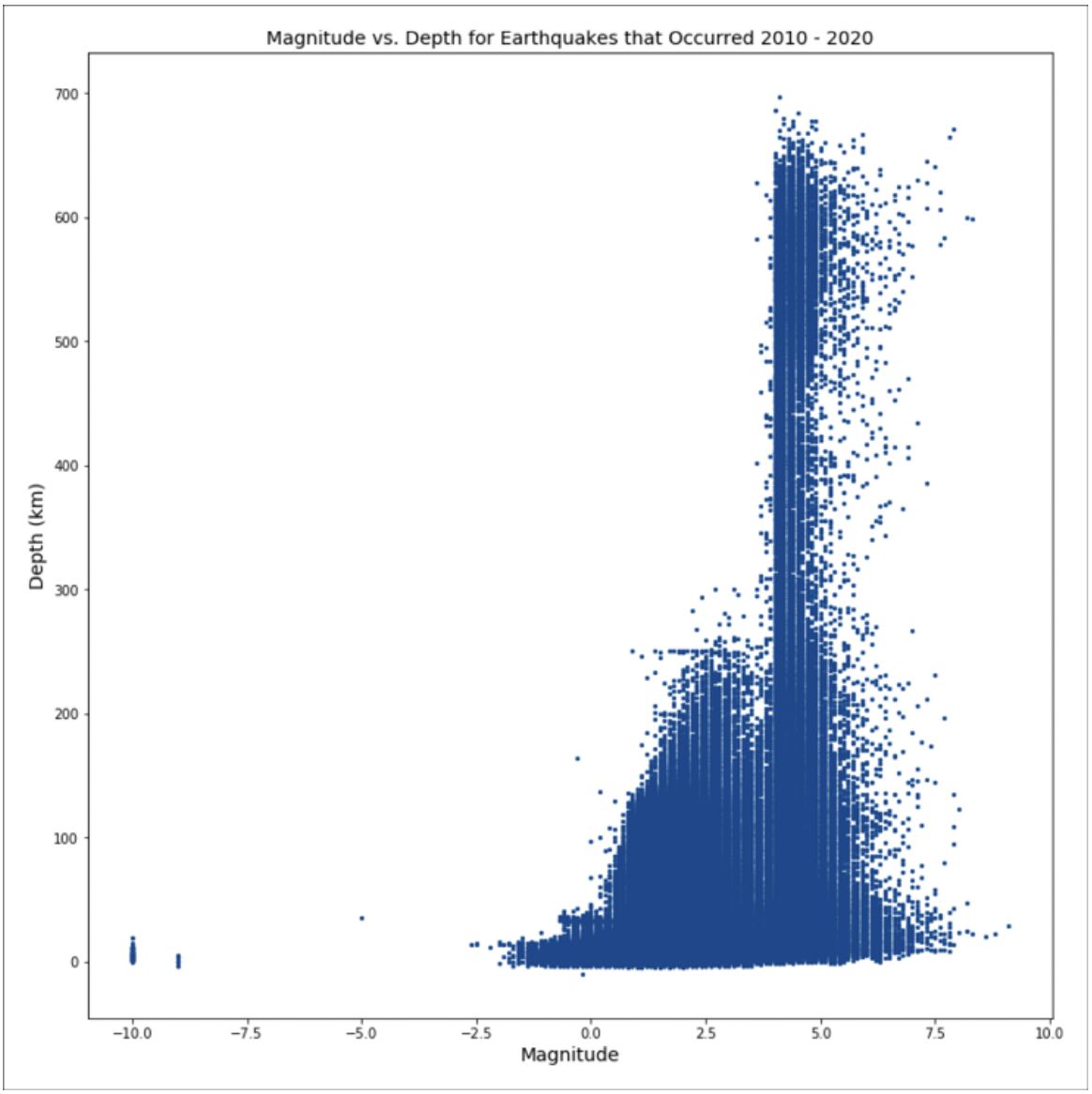


Figura 6.5: Diagrama de dispersão da magnitude e da profundidade dos terremotos.

Nesse gráfico, podemos ver o mesmo intervalo de magnitudes, agora representado em relação às profundidades, que variam de imediatamente abaixo de zero a cerca de 700 quilômetros. O interessante é que os valores de profundidade altos, acima de 300, correspondem a magnitudes iguais a aproximadamente 4 e mais altas. Talvez esses terremotos profundos só possam ser detectados após alcançarem uma magnitude mínima. Observe

que, devido ao volume de dados, tomei um atalho e agrupei os valores por uma combinação de magnitude e profundidade, em vez de exibir 1,5 milhão de pontos de dados. A contagem dos terremotos pode ser usada no dimensionamento de cada círculo da dispersão, como na Figura 6.6, que dá um zoom no intervalo de magnitudes de 4.0 a 7.0 e nas profundidades de 0 a 50 km.

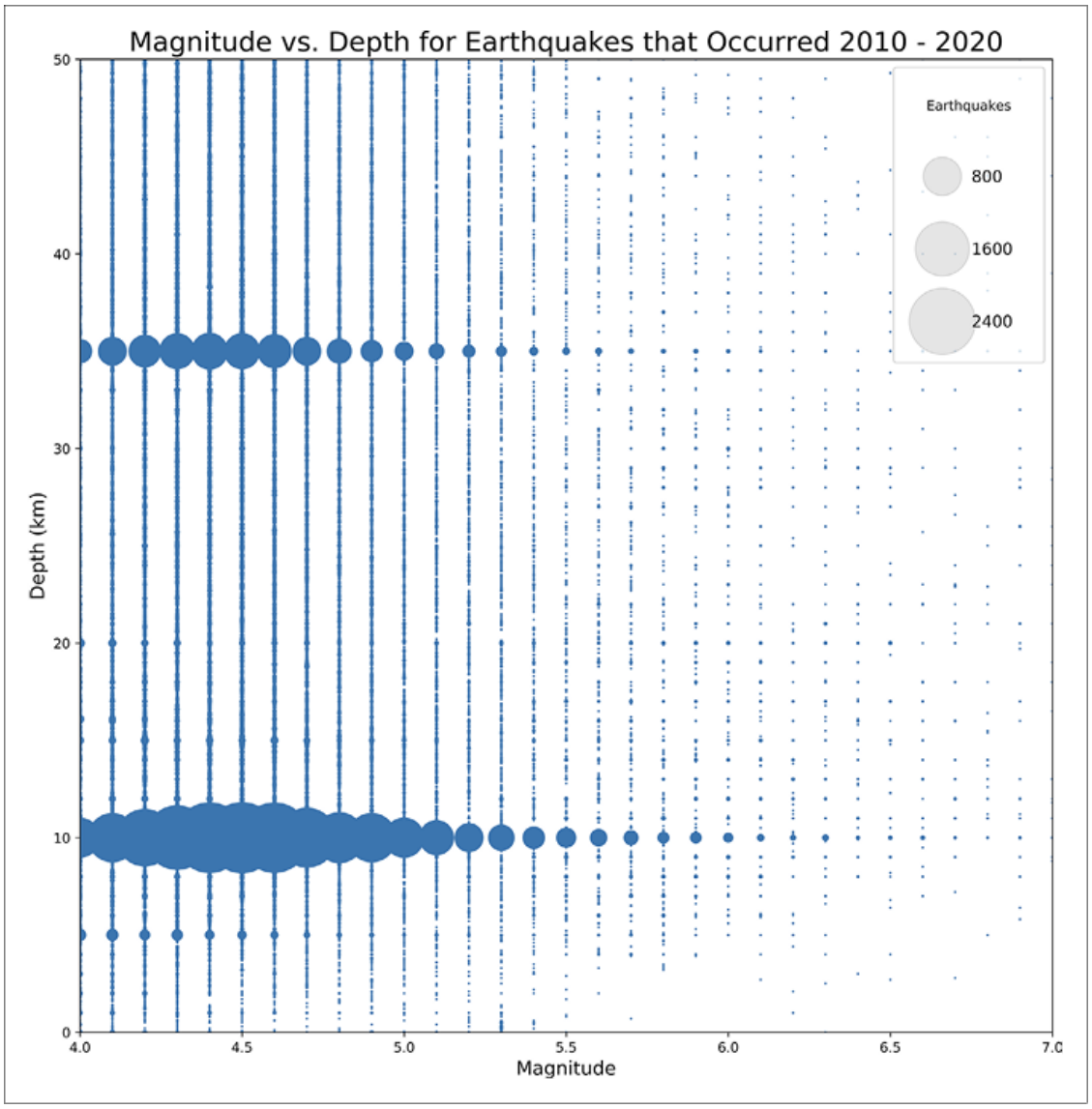


Figura 6.6: Diagrama de dispersão da magnitude e da profundidade dos terremotos, com zoom e com círculos dimensionados pelo número de terremotos.

Um terceiro tipo de gráfico útil na busca e na análise de valores discrepantes é o *diagrama de caixa*, também conhecido como *diagrama de caixa e bigodes (box-and-whisker plot)*. Esses gráficos resumem os dados do

meio do intervalo de valores e retêm os valores discrepantes. O nome desse tipo de gráfico vem da caixa, ou retângulo, exibida no meio. A linha que forma a base do retângulo fica localizada no valor do 25º percentil, a linha que forma o topo fica no valor do 75º percentil e a linha que passa pelo meio fica no valor do 50º percentil, ou da mediana. Os percentis já nos são familiares porque foram discutidos na seção anterior. Os "bigodes" do diagrama de caixa são linhas que se estendem para fora da caixa, normalmente até 1 vez e meia mais que o *intervalo interquartil*. O intervalo interquartil é simplesmente a diferença entre o valor do 75º percentil e o do 25º percentil. Qualquer valor que ultrapasse os bigodes é exibido no gráfico como valor discrepante.

Qualquer software ou linguagem de programação que você usar para criar diagramas de caixa se encarregará dos cálculos dos percentis e do intervalo interquartil. Muitos também oferecem opções para a exibição dos bigodes de acordo com desvios-padrão a partir da mediana ou com percentis mais distantes como o 10º e o 90º. O cálculo sempre será simétrico ao redor do ponto intermediário (como um desvio-padrão acima e abaixo da média), mas o tamanho do bigode superior e inferior pode diferir dependendo dos dados.

Normalmente, o diagrama de caixa exibe todos os valores. Já que o conjunto de dados é muito grande, nesse exemplo examinaremos o subconjunto de 16.036 terremotos que incluem "Japan" no campo `place`. Para começar, criaremos o conjunto de dados com um código SQL, que será composto de um simples *SELECT* de todos os valores de `mag` que atendem aos critérios de filtragem:

```
SELECT mag
FROM earthquakes
WHERE place like '%Japan%'
ORDER BY 1
;
mag
---
2.7
3.1
3.2
...
```

Em seguida, crie um diagrama de caixa no software de geração de gráficos selecionado, como mostrado na Figura 6.7.

Embora geralmente o software de criação de gráficos forneça essas informações, também podemos encontrar os principais valores do diagrama de caixa com SQL:

```
SELECT ntile_25, median, ntile_75
,(ntile_75 - ntile_25) * 1.5 as iqr
,ntile_25 - (ntile_75 - ntile_25) * 1.5 as lower_whisker
,ntile_75 + (ntile_75 - ntile_25) * 1.5 as upper_whisker
FROM (
    SELECT
    percentile_cont(0.25) within group (order by mag) as ntile_25
    ,percentile_cont(0.5) within group (order by mag) as median
    ,percentile_cont(0.75) within group (order by mag) as ntile_75
    FROM earthquakes
    WHERE place like '%Japan%'
) a
;
ntile_25  median  ntile_75  iqr  lower_whisker   upper_whisker
--------  ------  --------  ----  -------------  -------------
4.3       4.5     4.7       0.60  3.70           5.30
```

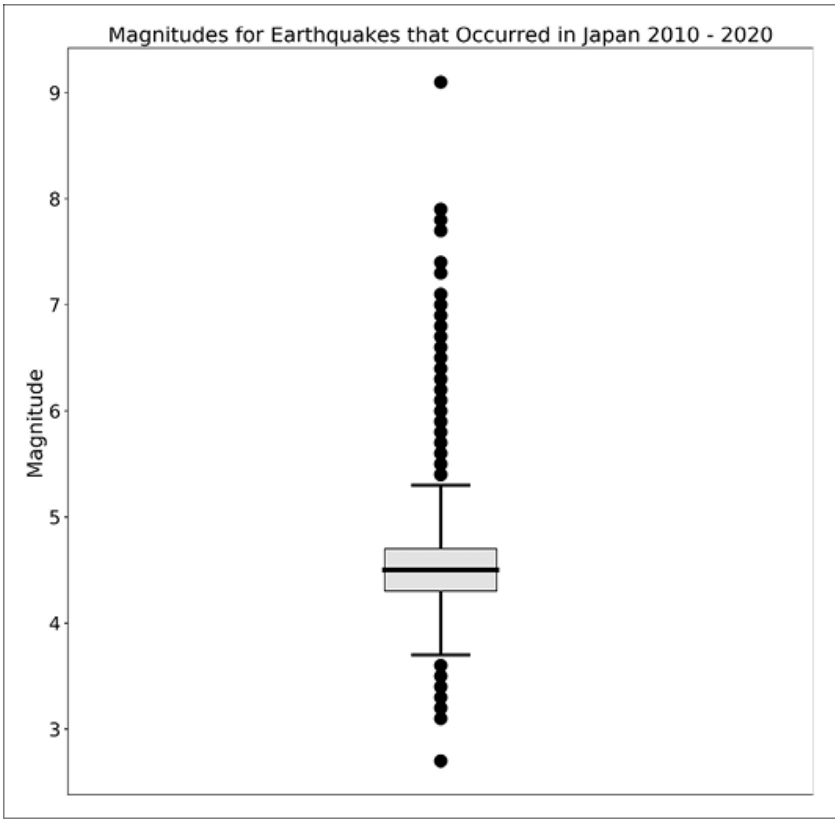


Figura 6.7: Diagrama de caixa exibindo a distribuição de magnitudes dos terremotos no Japão.

O terremoto japonês que representa a mediana teve magnitude igual a 4.5, e os bigodes se estendem de 3.7 a 5.3. Os círculos exibidos representam terremotos anômalos, tanto pequenos quanto grandes. O Grande Terremoto de Tohoku de 2011, com magnitude de 9.1, é obviamente um valor discrepante, mesmo entre os maiores terremotos ocorridos no Japão.

Pelo que vivenciei em minha experiência, acho os diagramas de caixa uma das visualizações mais difíceis de explicar para quem não conhece estatística ou não passa o dia todo criando e examinando visualizações. O intervalo interquartil é um conceito particularmente confuso, embora a noção de valores discrepantes pareça fazer sentido para a maioria das pessoas. Se não tiver certeza de que seu público-alvo saberá interpretar um diagrama de caixa, tente explicá-lo com termos claros, mas não excessivamente técnicos. Eu mantenho um desenho como o da Figura 6.8 que explica as partes de um diagrama de caixa e o envio junto com meu trabalho "apenas para o caso" de meu público-alvo precisar de alguma referência para consulta.

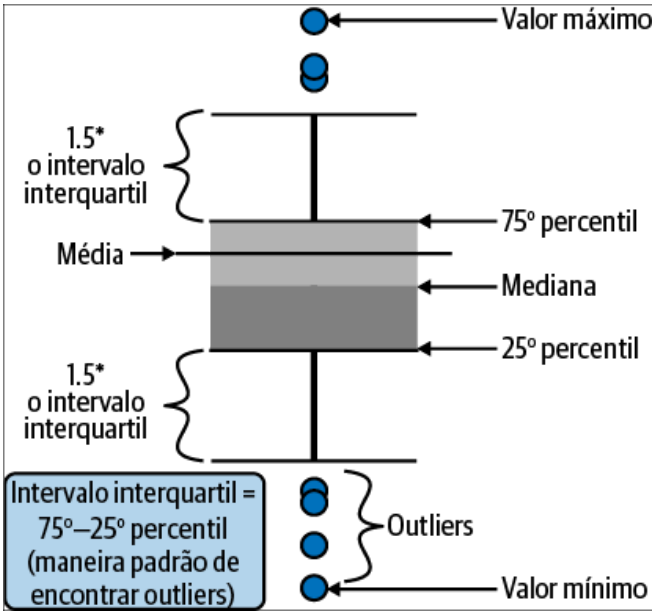


Figura 6.8: Representação gráfica das partes de um diagrama de caixa.

Os diagramas de caixa também podem ser usados na comparação de agrupamentos de dados para identificação e diagnóstico posterior de onde ocorrem valores discrepantes. Por exemplo, poderíamos comparar os terremotos do Japão em diferentes anos. Primeiro adicione o ano do campo `time` à saída SQL e depois exiba o diagrama, como na Figura 6.9:

```
SELECT date_part('year',time)::int as year
,mag
FROM earthquakes
WHERE place like '%Japan%'
ORDER BY 1,2
;
year  mag
----  ---
2010  3.6
2010  3.7
2010  3.7
...   ...
```

Embora a mediana e o intervalo das caixas flutuem um pouco de um ano para o outro, eles ficam consistentemente entre 4 e 5. O Japão vivenciou grandes terremotos anômalos em todos os anos, com pelo menos um com magnitude maior do que 6.0, e em seis anos ocorreu pelo menos um

terremoto com magnitude maior ou igual a 7.0. O Japão é sem dúvida uma região com muita atividade sísmica.

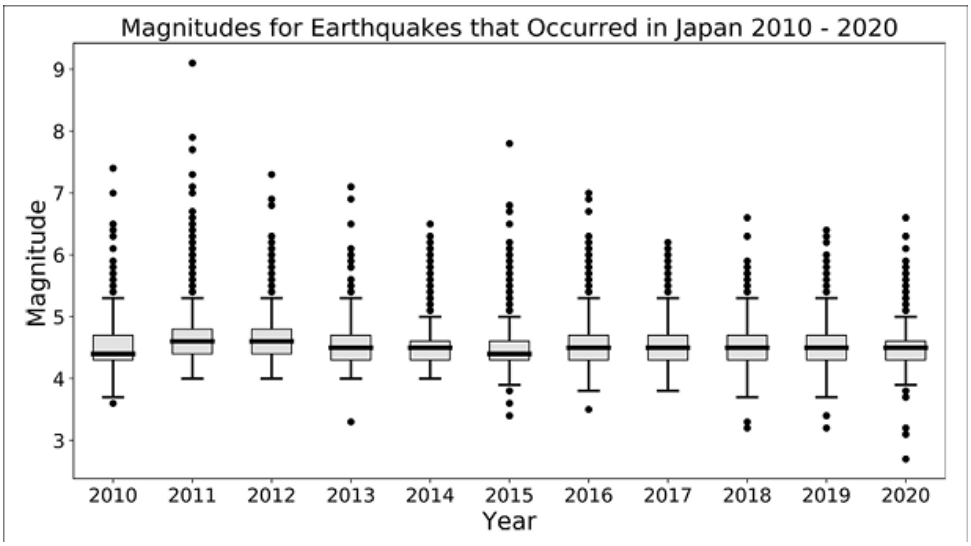


Figura 6.9: Diagrama de caixa das magnitudes dos terremotos do Japão, por ano.

Os gráficos de barras, os diagramas de dispersão e os diagramas de caixa costumam ser usados na detecção e na caracterização de valores discrepantes em conjuntos de dados. Eles nos permitem absorver rapidamente a complexidade de grandes quantidades de dados e contar a história existente por trás delas. Junto com a classificação, os percentis e os desvios-padrão, os gráficos são uma parte importante do kit de ferramentas de detecção de anomalias. Com essas ferramentas em mãos, estamos prontos para discutir as diversas formas que as anomalias podem assumir além das que vimos até agora.

Tipos de anomalias

As anomalias podem ter várias formas e tamanhos. Nesta seção, abordarei três categorias gerais de anomalias: valores, contagens de frequências, e presença ou ausência. Estes são pontos de partida para a investigação de qualquer conjunto de dados, seja como exercício de criação de perfis ou porque haja suspeita de anomalias. Os valores discrepantes e outros valores incomuns costumam ser específicos de uma área em particular; logo, geralmente quanto mais sabemos sobre como e por que os dados foram gerados, melhor. No entanto, esses padrões e técnicas para a

detecção de anomalias são bons pontos de partida para a investigação.

Valores anômalos

Talvez o tipo mais comum de anomalia, e a primeira coisa que vem à mente quando se trata desse tópico, seja quando valores individuais são valores discrepantes extremamente altos ou baixos, ou quando os valores do meio da distribuição são incomuns.

Na última seção, examinamos várias maneiras de encontrar valores discrepantes, por meio da classificação, de percentis e desvios-padrão, e de gráficos. Descobrimos que o conjunto de dados de terremotos tem valores de magnitude anormalmente grandes e alguns valores que parecem ser excepcionalmente pequenos. As magnitudes também contêm diferentes quantidades de *dígitos significativos*, ou dígitos à direita da vírgula ou do ponto decimal. Por exemplo, poderíamos inspecionar um subconjunto de valores próximos de 1 para encontrar um padrão que se repita em todo o conjunto de dados:

```
SELECT mag, count(*)
FROM earthquakes
WHERE mag > 1
GROUP BY 1
ORDER BY 1
limit 100
;
mag         count
----------  -----
...         ...
1.08        3863
1.08000004  1
1.09        3712
1.1         39728
1.11        3674
1.12        3995
....        ...
```

De vez em quando aparece um valor com 8 dígitos significativos. Muitos valores têm dois dígitos significativos, porém é mais comum que haja

apenas um dígito significativo. Isso ocorre provavelmente devido aos diferentes níveis de precisão dos instrumentos que estão coletando os dados da magnitude. O banco de dados também não exibe um segundo dígito significativo quando ele é zero; logo, "1.10" aparece simplesmente como "1.1". No entanto, o grande número de registros em "1.1" indica que essa não é apenas uma questão de exibição. Dependendo da finalidade da análise, pode ou não ser interessante ajustar os valores para que todos tenham o mesmo número de dígitos significativos por meio do arredondamento.

Geralmente, além de encontrarmos valores anômalos, é útil entender por que eles ocorreram ou conhecer outros atributos relacionados às anomalias. É nesse momento que a criatividade e o trabalho de detetive de dados entram em cena. Por exemplo, 1.215 registros do conjunto de dados têm valores de profundidade muito altos de mais de 600 quilômetros. Poderíamos querer saber onde esses valores discrepantes ocorreram ou como foram coletados. Examinaremos a fonte, que pode ser encontrada no campo `net` (que representa rede):

```
SELECT net, count(*)
FROM earthquakes
WHERE depth > 600
GROUP BY 1
;
net  count
---  -----
us   1215
```

O site do USGS indica que a fonte é o *USGS National Earthquake Information Center, PDE* (*https://earthquake.usgs.gov/data/comcat/contributor/us*). Contudo, isso não é muito informativo, então verificaremos os valores de `place`, que contêm os locais dos terremotos:

```
SELECT place, count(*)
FROM earthquakes
WHERE depth > 600
GROUP BY 1
;
place                              count
```

```
----------------------------  -----
100km NW of Ndoi Island, Fiji   1
100km SSW of Ndoi Island, Fiji  1
100km SW of Ndoi Island, Fiji   1
...                             ...
```

A inspeção visual sugere que muitos desses terremotos mais profundos ocorreram ao redor da ilha de Ndoi em Fiji. No entanto, o local inclui um componente de distância e direção, como "100km NW of", que dificulta a sumarização. Podemos fazer um parsing de texto para nos concentrarmos no local propriamente dito e obter insights melhores. Para locais que contenham valores, depois apresentem "of", e então sejam exibidos mais valores, faça a divisão na string "of" e pegue a segunda parte:

```
SELECT
case when place like '% of %' then split_part(place,' of ',2)
     else place end as place_name
,count(*)
FROM earthquakes
WHERE depth > 600
GROUP BY 1
ORDER BY 2 desc
;
place_name         count
-----------------  -----
Ndoi Island, Fiji  487
Fiji region        186
Lambasa, Fiji      140
...                ...
```

Agora podemos dizer com mais certeza que a maioria dos valores de profundidade muito altos é registrada para terremotos ocorridos em algum lugar das ilhas Fiji, com uma concentração específica ao redor da pequena ilha vulcânica de Ndoi. A análise poderia continuar para tornar-se mais complexa, por exemplo, com o parsing do texto para agruparmos todos os terremotos registrados na região maior, o que revelaria que, depois de Fiji, outros terremotos muito profundos foram registrados ao redor de Vanuatu e das Filipinas.

Anomalias podem surgir na forma de grafias incorretas, variações na

capitalização ou outros erros no texto. A facilidade da busca vai depender do número de valores distintos, ou da *cardinalidade*, do campo. Diferenças na capitalização podem ser detectadas tanto pela simples contagem dos valores distintos quanto pela contagem desses valores quando uma função `lower` ou `upper` é aplicada:

```
SELECT count(distinct type) as distinct_types
,count(distinct lower(type)) as distinct_lower
FROM earthquakes
;
distinct_types  distinct_lower
--------------  --------------
25              24
```

Há 24 valores distintos no campo `type`, mas 25 formas diferentes. Para encontrar os tipos específicos, podemos usar um cálculo e marcar os valores cuja forma minúscula não coincidir com o valor real. Incluir a contagem de registros de cada forma nos ajudará a contextualizar para depois decidir como manipulá-los:

```
SELECT type
,lower(type)
,type = lower(type) as flag
,count(*) as records
FROM earthquakes
GROUP BY 1,2,3
ORDER BY 2,4 desc
;
type       lower      flag   records
---------  ---------  -----  -------
...        ...        ...    ...
explosion  explosion  true   9887
ice quake  ice quake  true   10136
Ice Quake  ice quake  false  1
...        ...        ...    ...
```

O valor anômalo de "Ice quake" é fácil de identificar, já que ele é o único valor para o qual o cálculo da flag retorna `false`. Como há apenas um registro com esse valor, em comparação com os 1.136 com a forma minúscula, podemos presumir que ele pode estar agrupado com os outros

registros. Outras funções de texto podem ser aplicadas, como `trim` se suspeitarmos que os valores contêm espaços iniciais ou finais adicionais, ou `replace` se desconfiarmos que certas grafias têm várias formas, como o número "2" e a palavra "dois".

Grafias incorretas podem ser mais difíceis de descobrir do que outras variações. Se existir um conjunto conhecido de valores e grafias corretos, ele pode ser usado na validação dos dados por meio de uma *OUTER JOIN* para uma tabela contendo os valores ou com uma instrução CASE combinada com uma lista IN. Nos dois casos, o objetivo é marcar valores que sejam inesperados ou inválidos. Sem esse conjunto de valores corretos, geralmente as opções são aplicar o conhecimento que temos da área ou fazer suposições baseadas na experiência. Na tabela `earthquakes`, podemos examinar os valores de `type` com apenas alguns registros e tentar determinar se há outro valor mais comum que possa ser substituído:

```
SELECT type, count(*) as records
FROM earthquakes
GROUP BY 1
ORDER BY 2 desc
;
type                        records
--------------------------  -------
...                         ...
landslide                   15
mine collapse               12
experimental explosion      6
building collapse           5
...                         ...
meteorite                   1
accidental explosion        1
collapse                    1
induced or triggered event  1
Ice Quake                   1
rockslide                   1
```

Examinamos "Ice Quake" anteriormente e decidimos que deve ser a mesma coisa que "ice quake". Há apenas um registro para "rockslide", que poderia ser considerado semelhante a outro valor, "landslide", que tem

15 registros. “Collapse” é mais ambíguo, já que o conjunto de dados inclui tanto “mine collapse” quanto “building collapse”. O que faremos com esses valores, ou se faremos algo, vai depender do objetivo da análise, como discutirei posteriormente em “Manipulando anomalias”, na página 300.

Contagens ou frequências anômalas

Às vezes as anomalias não ocorrem na forma de valores individuais, e sim na forma de padrões ou grupos de atividade nos dados. Por exemplo, um cliente gastando 100 dólares em um site de e-commerce pode não ser incomum, mas esse mesmo cliente gastando 100 dólares por hora no decorrer de 48 horas quase certamente seria uma anomalia.

Há várias dimensões nas quais os grupos de atividade podem indicar anomalias e muitas delas dependem do contexto dos dados. Tanto o tempo quanto o local são comuns em vários conjuntos de dados e são características do conjunto de dados `earthquakes`; portanto, irei usá-los para ilustrar as técnicas desta seção. Lembre-se de que geralmente essas técnicas também podem ser aplicadas a outros atributos.

Eventos ocorrendo com frequência incomum durante um curto período de tempo podem indicar atividade anômala. Isso pode ser bom, como quando uma celebridade promove um produto inesperadamente, levando a um aumento nas vendas desse produto. No entanto, também pode ser ruim, como quando picos incomuns indicam uso fraudulento de um cartão de crédito ou tentativas de deixar um site inativo com inundação de tráfego. Para entender esses tipos de anomalias e saber se há desvios em relação à tendência normal, primeiro aplique as agregações apropriadas e depois use as técnicas introduzidas anteriormente neste capítulo, junto com as técnicas de análise de séries temporais discutidas no Capítulo 3.

Nos exemplos a seguir, percorrerei uma série de etapas e consultas que nos ajudarão a conhecer os padrões normais e a procurar os incomuns. Esse é um processo iterativo que usa perfis de dados, conhecimento da área e insights de resultados de consultas anteriores para guiar cada etapa. Começaremos nossa jornada verificando as contagens de terremotos por ano, o que podemos fazer truncando o campo `time` para o nível anual e contando os registros. Para bancos de dados que não suportem

`date_trunc`, considere usar `extract` ou `trunc`:

```
SELECT date_trunc('year',time)::date as earthquake_year
,count(*) as earthquakes
FROM earthquakes
GROUP BY 1
;
earthquake_year  earthquakes
---------------  -----------
2010-01-01       122322
2011-01-01       107397
2012-01-01       105693
2013-01-01       114368
2014-01-01       135247
2015-01-01       122914
2016-01-01       122420
2017-01-01       130622
2018-01-01       179304
2019-01-01       171116
2020-01-01       184523
```

Podemos ver que 2011 e 2012 tiveram poucos terremotos em comparação com outros anos. Também houve um nítido aumento nos registros em 2018 que se sustentou em 2019 e 2020. Isso parece incomum, e poderíamos supor que de repente a Terra tornou-se sismicamente mais ativa, que há um erro nos dados, como a duplicação de registros, ou que algo mudou no processo de coleta de dados. Desceremos ao nível mensal para ver se essa tendência persiste em um nível de tempo mais granular:

```
SELECT date_trunc('month',time)::date as earthquake_month
,count(*) as earthquakes
FROM earthquakes
GROUP BY 1
;
earthquake_month  earthquakes
----------------  -----------
2010-01-01        9651
2010-02-01        7697
2010-03-01        7750
```

... ...

A saída pode ser vista na Figura 6.10. Podemos ver que, embora o número de terremotos varie de um mês para o outro, parece haver um aumento geral a partir de 2017. Também podemos ver que há três meses anômalos, em abril de 2010, julho de 2018 e julho de 2019.

Desse ponto em diante podemos continuar verificando os dados em períodos de tempo mais granulares, podendo opcionalmente filtrar o conjunto de resultados por um intervalo de datas para dar destaque a esses períodos anômalos. Após fazer o direcionamento para os dias específicos ou até mesmo para horas do dia para detectar quando os picos ocorreram, poderíamos detalhar ainda mais os dados por outros atributos do conjunto de dados. Isso pode ajudar a explicar as anomalias ou pelo menos restringir as condições nas quais elas ocorreram.

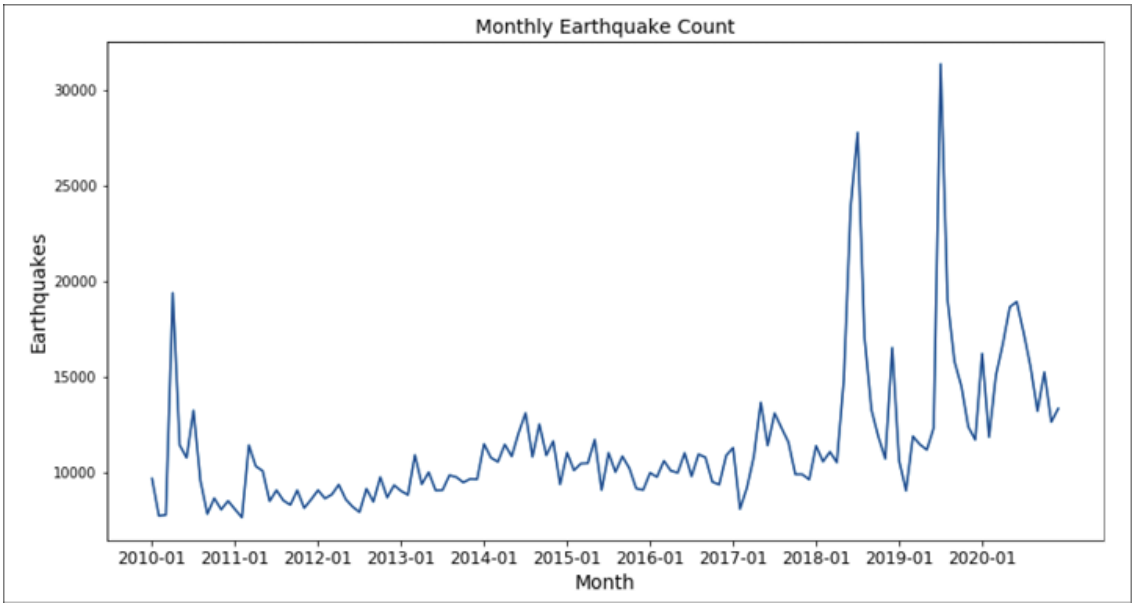


Figura 6.10: Número de terremotos por mês.

Por exemplo, o aumento nos terremotos a partir de 2017 pode ser pelo menos parcialmente explicado pelo campo `status`. O status indica se o evento foi revisado por um humano ("reviewed") ou postado diretamente pelo sistema sem revisão ("automatic"):

```
SELECT date_trunc('month',time)::date as earthquake_month
,status
,count(*) as earthquakes
```

```
FROM earthquakes
GROUP BY 1,2
ORDER BY 1
;
earthquake_month  status     earthquakes
----------------  --------   -----------
2010-01-01        automatic  620
2010-01-01        reviewed   9031
2010-02-01        automatic  695
...               ...        ...
```

As tendências dos status “automatic” e “reviewed” estão representadas em gráfico na Figura 6.11.

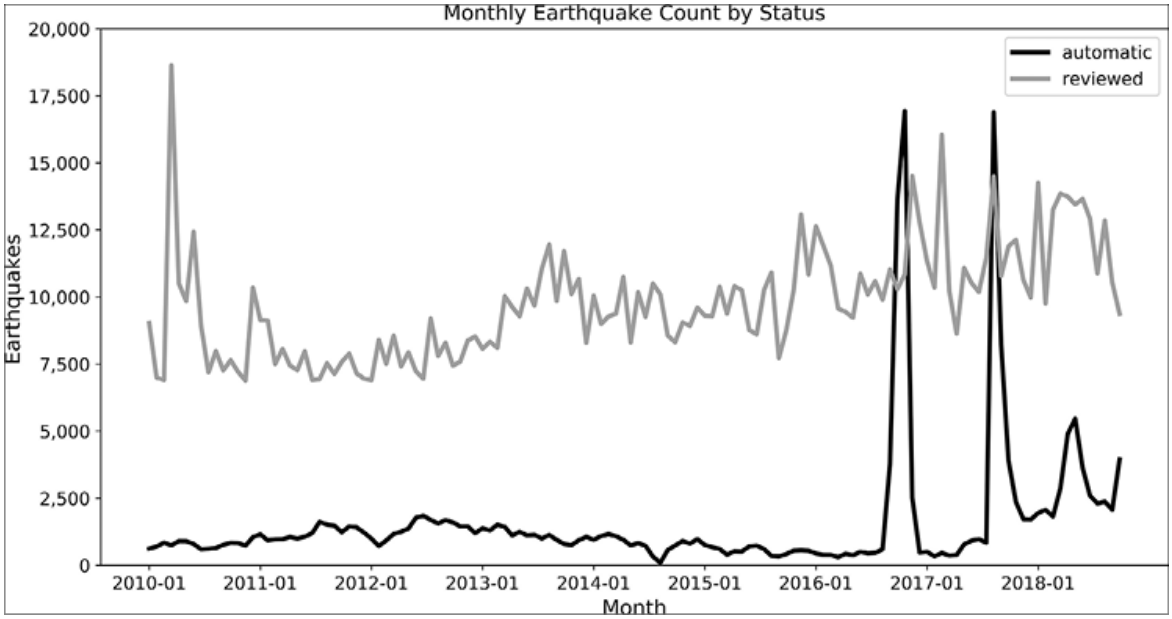


Figura 6.11: Número de terremotos por mês, divididos por status.

Podemos ver no gráfico que as contagens de valores discrepantes em julho de 2018 e em julho de 2019 ocorrem devido a grandes aumentos no número de terremotos com status “automatic”, enquanto o pico de abril de 2010 foi de terremotos de status “reviewed”. Um novo tipo de equipamento de registro automático pode ter sido adicionado ao conjunto de dados em 2017 ou ainda pode não ter havido tempo suficiente para a revisão de todos os registros.

A análise do local em conjuntos de dados que tenham essa informação pode ser outra maneira útil de encontrar e entender anomalias. A tabela

`earthquakes` contém informações sobre milhares de terremotos bem pequenos, o que pode obscurecer a visualização dos terremotos maiores e mais significativos. Examinaremos os locais em que ocorreram os terremotos maiores, de magnitude 6 ou mais alta, e veremos onde eles estão agrupados geograficamente:

```
SELECT place, count(*) as earthquakes
FROM earthquakes
WHERE mag >= 6
GROUP BY 1
ORDER BY 2 desc
;
place                                  earthquakes
-----------------------------------  -----------
near the east coast of Honshu, Japan  52
off the east coast of Honshu, Japan   34
Vanuatu                               28
...                                   ...
```

Ao contrário do tempo, que consultamos em níveis progressivamente mais granulares, os valores de `place` já são tão granulares que fica um pouco mais difícil captar o cenário geral, embora a região de Honshu no Japão claramente se destaque. Podemos aplicar algumas das técnicas de análise de texto do Capítulo 5 para fazer o parsing e depois agrupar as informações geográficas. Nesse caso, usaremos `split_part` para remover o texto relacionado à direção (como "near the coast of" ou "100km N of") que geralmente aparece no início do campo `place`:

```
SELECT
case when place like '% of %' then split_part(place,' of ',2)
     else place
     end as place
,count(*) as earthquakes
FROM earthquakes
WHERE mag >= 6
GROUP BY 1
ORDER BY 2 desc
;
place                  earthquakes
```

```
--------------------  -----------
Honshu, Japan         89
Vanuatu               28
Lata, Solomon Islands 28
...                   ...
```

A região ao redor de Honshu no Japão teve 89 terremotos, o que a torna não só o local do maior terremoto do conjunto de dados, mas também um valor discrepante pelo número de terremotos muito grandes registrados. Poderíamos dar continuidade no parsing, na limpeza e no agrupamento dos valores de `place` para obter um retrato mais refinado de onde ocorreram os maiores terremotos do planeta.

Geralmente encontrar contagens, somas ou frequências anômalas nos dados é um exercício que envolve várias rodadas sucessivas de consultas com diferentes níveis de granularidade. É comum que comece com um escopo mais amplo, aumente a granularidade, diminua o zoom novamente para comparar tendências de linha de base e mais uma vez aumente o zoom em certas divisões ou dimensões dos dados. Felizmente, o SQL é uma ótima ferramenta para esse tipo de iteração rápida. A combinação de técnicas, principalmente da análise de séries temporais, discutida no Capítulo 3, e da análise de texto, discutida no Capítulo 5, trará ainda mais sofisticação para a análise.

Anomalias pela ausência de dados

Vimos que frequências de eventos anormalmente altas podem sinalizar anomalias. No entanto, lembre-se de que a ausência de registros também pode indicar anomalias. Por exemplo, o batimento cardíaco de um paciente que está sendo operado é monitorado. A ausência de batimento em algum momento gerará um alerta, e o mesmo ocorrerá no caso de irregularidades no batimento. Em muitos contextos, entretanto, será difícil detectar a ausência de dados se você não estiver procurando por ela. Nem sempre os clientes anunciam que vão mudar de fornecedor. Eles simplesmente param de usar o produto ou serviço e abandonam silenciosamente o conjunto de dados.

Uma maneira de assegurar que as ausências nos dados sejam notadas seria com o uso de técnicas da análise de coorte, discutida no Capítulo 4.

Especificamente, uma *JOIN* para uma série de datas ou uma dimensão de dados, para assegurar que exista um registro para cada entidade não importando se ele estava ou não presente nesse período de tempo, facilita a detecção de ausências.

Outra maneira de detectar a ausência seria procurando lacunas, ou o tempo passado desde a última visualização. Algumas regiões estão mais propensas à ocorrência de grandes terremotos devido à maneira como as placas tectônicas estão dispostas ao redor do globo. Também detectamos algo assim nos dados de nossos exemplos anteriores. Os terremotos são notoriamente difíceis de prever, mesmo quando temos uma noção de onde eles devem ocorrer. Isso não impede que algumas pessoas especulem sobre o próximo terremoto "big one" (gigante) simplesmente devido ao período de tempo passado desde que o último ocorreu. Podemos usar SQL para encontrar as lacunas entre os grandes terremotos e o tempo passado desde o mais recente:

```
SELECT place
,extract('days' from '2020-12-31 23:59:59' - latest)
as days_since_latest
,count(*) as earthquakes
,extract('days' from avg(gap)) as avg_gap
,extract('days' from max(gap)) as max_gap
FROM
(
    SELECT place
    ,time
    ,lead(time) over (partition by place order by time) as next_time
    ,lead(time) over (partition by place order by time) - time as
  gap
    ,max(time) over (partition by place) as latest
    FROM
    (
        SELECT
        replace(
          initcap(
          case when place ~ ', [A-Z]' then split_part(place,', ',2)
               when place like '% of %' then split_part(place,' of
```

```
  ',2)
               else place end
        )
        ,'Region','')
        as place
        ,time
        FROM earthquakes
        WHERE mag > 5
    ) a
) a
GROUP BY 1,2
;
place             days_since_latest  earthquakes  avg_gap  max_gap
---------------   -----------------  -----------  -------  -------
Greece            62.0               109          36.0     256.0
Nevada            30.0               9            355.0    1234.0
Falkland Islands  2593.0             3            0.0      0.0
...               ...                ...          ...      ...
```

Na subconsulta mais interna, o campo `place` passa por parsing e é limpo, retornando regiões ou países maiores, junto com a hora de cada terremoto, para todos os terremotos de magnitude 5 ou maior. A segunda subconsulta usa uma função `lead` para calcular o momento da ocorrência do próximo terremoto, caso haja, para cada local e hora, e a lacuna entre cada terremoto e a ocorrência do próximo. A função de janela `max` retorna o terremoto mais recente de cada local. A consulta externa calcula os dias passados desde o último terremoto de magnitude maior do que 5 no conjunto de dados, usando a função `extract` para retornar apenas os dias do intervalo que é retornado quando duas datas são subtraídas. Já que o conjunto de dados só inclui registros até o fim de 2020, o timestamp "2020-12-31 23:59:59" é usado, embora seja apropriado usar `current_timestamp` ou uma expressão equivalente se os dados forem atualizados continuamente. Os dias são extraídos de maneira semelhante da média e do limite máximo do valor de `gap`.

Na prática, o tempo passado desde o último grande terremoto ocorrido em um local pode ter pouco poder preditivo, mas, em muitas áreas, as métricas das lacunas e do tempo desde a última visualização têm

aplicações práticas. O conhecimento das lacunas existentes entre as ações define uma linha de base com a qual a lacuna atual pode ser comparada. Se a lacuna atual estiver dento do intervalo de valores históricos, podemos deduzir que um cliente continua fiel, mas, se ela estiver muito distante, o risco de o cliente ter mudado de fornecedor aumentará. O conjunto de resultados de uma consulta que retorne lacunas históricas pode tornar-se ele próprio o tema de uma análise de detecção de anomalias, respondendo a perguntas como o período de tempo mais longo durante o qual um cliente esteve ausente antes de voltar.

Manipulando anomalias

Anomalias podem aparecer em conjuntos de dados por várias razões e podem assumir muitas formas, como acabamos de ver. Após detectá-las, a próxima etapa é manipulá-las de algum modo. Como isso será feito vai depender tanto da origem da anomalia – problema no processo subjacente ou de qualidade de dados – quanto do objetivo final do conjunto de dados ou da análise. As opções incluem a investigação sem alterações, a remoção, a substituição, o reescalonamento, e a correção em uma etapa inicial.

Investigação

Geralmente descobrir, ou tentar descobrir, a causa de uma anomalia é a primeira etapa da decisão do que fazer com ela. Essa parte do processo pode ser divertida e frustrante – divertida no sentido de que investigar e resolver um mistério nos fará usar nossas habilidades e a criatividade, mas frustrante porque quase sempre trabalhamos sob a pressão do tempo e rastrear anomalias pode dar a sensação de se estar em um labirinto interminável, levando-nos a nos perguntar se não houve uma falha durante a análise inteira.

Quando investigo anomalias, geralmente meu processo envolve uma série de consultas que se alternam entre a busca de padrões e a verificação de exemplos específicos. É fácil identificar um valor anômalo real. Nesses casos, costumo procurar na linha inteira que contém o valor discrepante pistas do timing, da origem e de qualquer outro atributo que esteja

disponível. Em seguida, verifico registros que compartilham desses atributos para ver se eles têm valores que pareçam incomuns. Por exemplo, eu poderia verificar se outros registros do mesmo dia têm valores normais ou incomuns. O tráfego de um site específico ou as compras de determinado produto podem revelar outras anomalias.

Após investigar a origem e os atributos das anomalias ao trabalhar com dados produzidos internamente em minha organização, entro em contato com os stakeholders ou os proprietários do produto. Às vezes há um bug ou falha conhecido, mas geralmente existe um problema real em um processo ou sistema que precisa ser resolvido, e informações sobre o contexto são úteis. Para conjuntos de dados externos ou públicos, pode não haver uma oportunidade de descobrir qual é a causa raiz. Nesses casos, tento coletar informações suficientes para decidir qual das opções discutidas a seguir é apropriada.

Remoção

Uma opção para o tratamento de anomalias de dados seria simplesmente as remover do conjunto de dados. Se houver alguma razão que leve à suspeita de que há um erro na coleção de dados que possa afetar o registro inteiro, a remoção será apropriada. A remoção também será uma boa opção se o conjunto de dados for suficientemente grande a ponto de a eliminação de alguns registros não afetar as conclusões. Outra boa razão para o uso da remoção seria se os valores discrepantes fossem tão extremos que distorcessem os resultados a ponto de serem geradas conclusões totalmente inapropriadas.

Vimos anteriormente que o conjunto de dados `earthquakes` contém vários registros com magnitude –9.99 e alguns com magnitude –9. Já que os terremotos correspondentes a esses valores são muito pequenos, poderíamos suspeitar que há valores incorretos ou que eles foram inseridos quando a magnitude real não era conhecida. É fácil remover os registros que contêm esses valores na cláusula *WHERE*:

```
SELECT time, mag, type
FROM earthquakes
WHERE mag not in (-9,-9.99)
limit 100
```

```
;
time                  mag   type
--------------------  ----  ----------
2019-08-11 03:29:20   4.3   earthquake
2019-08-11 03:27:19   0.32  earthquake
2019-08-11 03:25:39   1.8   earthquake
```

No entanto, antes de remover os registros, poderíamos querer saber se a inclusão dos valores discrepantes faz diferença para a saída. Por exemplo, poderíamos querer saber se remover os valores discrepantes afetará a magnitude média, já que as médias podem ser facilmente distorcidas por eles. Podemos fazer isso calculando a média no conjunto de dados inteiro, e sem os valores extremamente baixos, usando uma instrução CASE para excluí-los:

```
SELECT avg(mag) as avg_mag
,avg(case when mag > -9 then mag end) as avg_mag_adjusted
FROM earthquakes
;
avg_mag               avg_mag_adjusted
------------------    ------------------
1.6251015161530643    1.6273225642983641
```

As médias só são diferentes no terceiro dígito significativo (1.625 versus 1.627), o que é uma diferença bem pequena. Contudo, se fizermos a filtragem apenas para Yellowstone National Park, onde ocorrem muitos dos valores -9.99, a diferença será mais relevante:

```
SELECT avg(mag) as avg_mag
,avg(case when mag > -9 then mag end) as avg_mag_adjusted
FROM earthquakes
WHERE place = 'Yellowstone National Park, Wyoming'
;
avg_mag                 avg_mag_adjusted
----------------------  ----------------------
0.40639347873981053095  0.92332793709528214616
```

Embora esses ainda sejam valores pequenos, a diferença entre uma média de 0.46 e de 0.92 é suficientemente grande; logo, pode ser preferível remover os valores discrepantes.

Observe que há duas opções para se fazer isso: na cláusula *WHERE*, que remove os valores discrepantes de todos os resultados, ou em uma instrução CASE, que os remove somente de cálculos específicos. A seleção da opção a ser usada vai depender do contexto da análise, assim como de se será importante preservar as linhas, para a retenção de contagens totais, ou valores úteis de outros campos.

Substituição por valores alternativos

Geralmente os valores anômalos podem ser manipulados pela sua substituição por outros valores em vez de pela remoção de registros inteiros. Um valor alternativo pode ser um padrão, um valor substituto, o valor numérico mais próximo dentro de um intervalo ou uma síntese estatística como a média ou a mediana.

Vimos anteriormente que valores nulos podem ser substituídos por um padrão com o uso da função `coalesce`. Quando os valores não forem necessariamente nulos, mas forem problemáticos por alguma outra razão, uma instrução CASE pode ser usada para fazer a substituição por um valor padrão. Por exemplo, em vez de relatar todos os diversos eventos sísmicos, poderíamos agrupar os tipos que *não* são terremotos no valor único "Other":

```
SELECT
case when type = 'earthquake' then type
     else 'Other'
     end as event_type
,count(*)
FROM earthquakes
GROUP BY 1
;
event_type  count
----------  -------
earthquake  1461750
Other       34176
```

É claro que isso reduz a quantidade de detalhes dos dados, mas também pode ser uma maneira de resumir um conjunto de dados que tenha muitos valores anômalos para `type`, como já vimos. Quando sabemos que

os valores anômalos estão incorretos, e conhecemos o valor correto, substituí-los com uma instrução CASE também é uma solução que preserva a linha no conjunto de dados geral. Por exemplo, um 0 adicional poderia ter sido incluído no fim de um registro, ou um valor poderia ter sido registrado em polegadas em vez de milhas.

Outra opção para a manipulação de valores discrepantes numéricos seria substituir os valores extremos pelo valor alto ou baixo mais próximo que não seja extremo. Essa abordagem mantém grande parte do intervalo de valores, mas evita as médias que podem ser geradas por valores discrepantes extremos. A *winsorização* é uma técnica própria para isso, na qual os valores discrepantes são configurados com um percentil específico dos dados. Por exemplo, se tivéssemos valores acima do 95º percentil e abaixo do 5º percentil, poderíamos configurar aqueles com o valor do 95º percentil e estes com o valor do 5º percentil. Para fazer esse cálculo em SQL, primeiro encontraremos os valores do 5º e do 95º percentis:

```
SELECT percentile_cont(0.95) within group (order by mag)
as percentile_95
,percentile_cont(0.05) within group (order by mag)
as percentile_05
FROM earthquakes
;
percentile_95  percentile_05
-------------  -------------
4.5            0.12
```

Podemos inserir esse cálculo em uma subconsulta e então usar uma instrução CASE para manipular a configuração de valores para valores discrepantes abaixo do 5º percentil e acima do 95º percentil. Oberve a *JOIN* Cartesiana que nos permite comparar os valores dos percentis com cada magnitude individual:

```
SELECT a.time, a.place, a.mag
,case when a.mag > b.percentile_95 then b.percentile_95
      when a.mag < b.percentile_05 then b.percentile_05
      else a.mag
      end as mag_winsorized
FROM earthquakes a
```

```
JOIN
(
    SELECT percentile_cont(0.95) within group (order by mag)
     as percentile_95
    ,percentile_cont(0.05) within group (order by mag)
     as percentile_05
    FROM earthquakes
) b on 1 = 1
;
time                 place                        mag   mag_winsoriz
  e
-------------------  ---------------------------  ----  ------------
  -
2014-01-19 06:31:50  5 km SW of Volcano, Hawaii   -9    0.12
2012-06-11 01:59:01  Nevada                       -2.6  0.12
...                  ...                          ...   ...
2020-01-27 21:59:01  31km WNW of Alamo, Nevada    2     2.0
2013-07-07 08:38:59  54km S of Fredonia, Arizona  3.5   3.5
...                  ...                          ...   ...
2013-09-25 16:42:43  46km SSE of Acari, Peru      7.1   4.5
2015-04-25 06:11:25  36km E of Khudi, Nepal       7.8   4.5
...                  ...                          ...   ...
```

O valor do 5º percentil é 0.12, enquanto o do 95º percentil é 4.5. Valores abaixo e acima desses limites serão alterados para o limite do campo `mag_winsorize`. Valores entre esses limites continuarão os mesmos. Não há um limite para o percentil definido na winsorização. O 1º e o 99º percentis ou até mesmo os percentis 0,01 e 99,9 podem ser usados dependendo dos requisitos da análise e de quanto os valores discrepantes forem predominantes e extremos.

Reescalonamento

Em vez da exclusão de registros por filtragem ou da alteração dos valores de valores discrepantes, o reescalonamento fornece um caminho que retém todos os valores, mas facilita a análise e a criação de gráficos.

Já abordamos o z-score, mas gostaria de destacar que ele pode ser usado para o reescalonamento de valores. O z-score é útil porque pode ser usado

com valores positivos e negativos.

Outra transformação comum é a conversão para a escala logarítmica (de log). O benefício da transformação de valores para a escala de log é que eles retêm a mesma ordem, mas números pequenos ficam mais espalhados. As transformações em log também podem ser convertidas novamente para a escala original, facilitando a interpretação. Uma desvantagem é que a transformação em log não pode ser usada em números negativos. No conjunto de dados `earthquakes`, vimos que a magnitude já está expressa na escala de log. O Grande Terremoto de Tohoku de magnitude 9.1 é extremo, mas o valor pareceria ainda mais extremo se não estivesse na escala de log!

O campo `depth` é medido em quilômetros. Aqui consultaremos não só a profundidade, mas também a profundidade com a função `log` aplicada, e depois exibiremos a saída em gráfico nas figuras 6.12 e 6.13 para demonstrar a diferença. A função `log` usa a base 10 como padrão. Para reduzirmos o conjunto de resultados a fim de facilitar a exibição em gráfico, a profundidade também é arredondada para um único dígito significativo com o uso da função `round`. A tabela é filtrada para excluir valores menores do que 0.05, já que eles seriam arredondados para valores menores ou iguais a zero:

```
SELECT round(depth,1) as depth
,log(round(depth,1)) as log_depth
,count(*) as earthquakes
FROM earthquakes
WHERE depth >= 0.05
GROUP BY 1,2
;
depth  log_depth            earthquakes
-----  -------------------  -----------
0.1    -1.0000000000000000  6994
0.2    -0.6989700043360188  6876
0.3    -0.5228787452803376  7269
...    ...                  ...
```

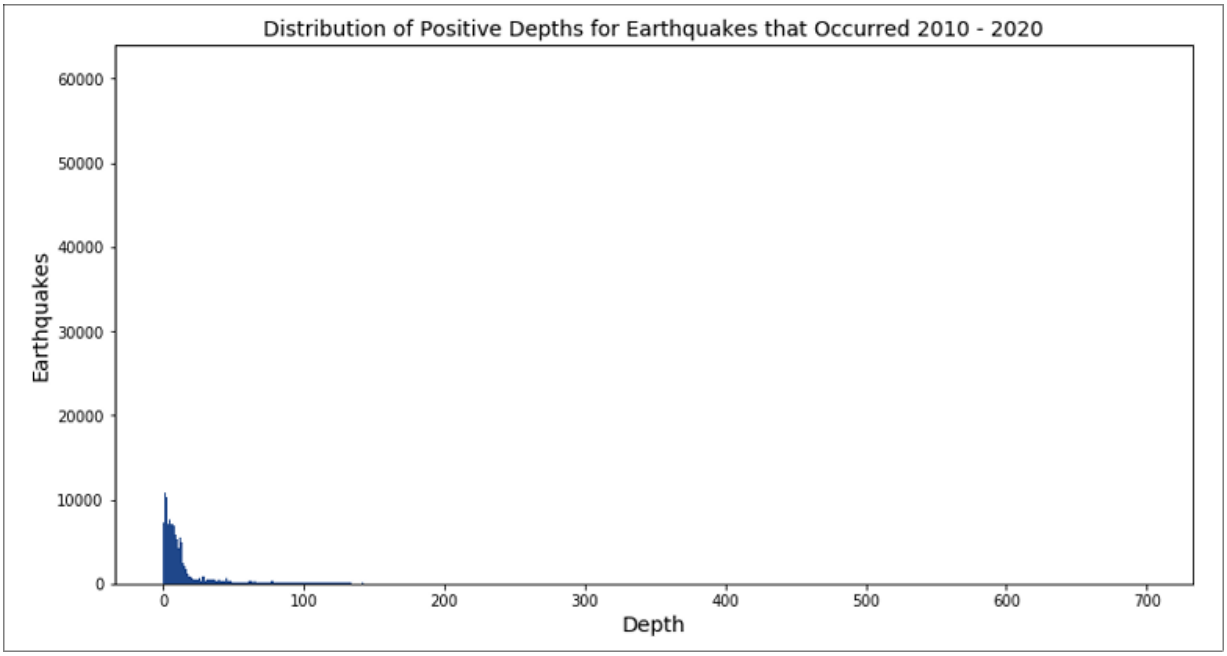


Figura 6.12: Distribuição de terremotos por profundidade, com profundidades não ajustadas.

Na Figura 6.12, fica claro que há um grande número de terremotos entre 0.05 e talvez 20, mas além desses valores é difícil ver a distribuição, já que o eixo x se estende até 700 para capturar o intervalo dos dados. No entanto, quando a profundidade é transformada para uma escala de log na Figura 6.13, é muito mais fácil ver a distribuição dos valores menores. Particularmente, o pico em 1.0, que corresponde a uma profundidade de 10 quilômetros, se destaca.

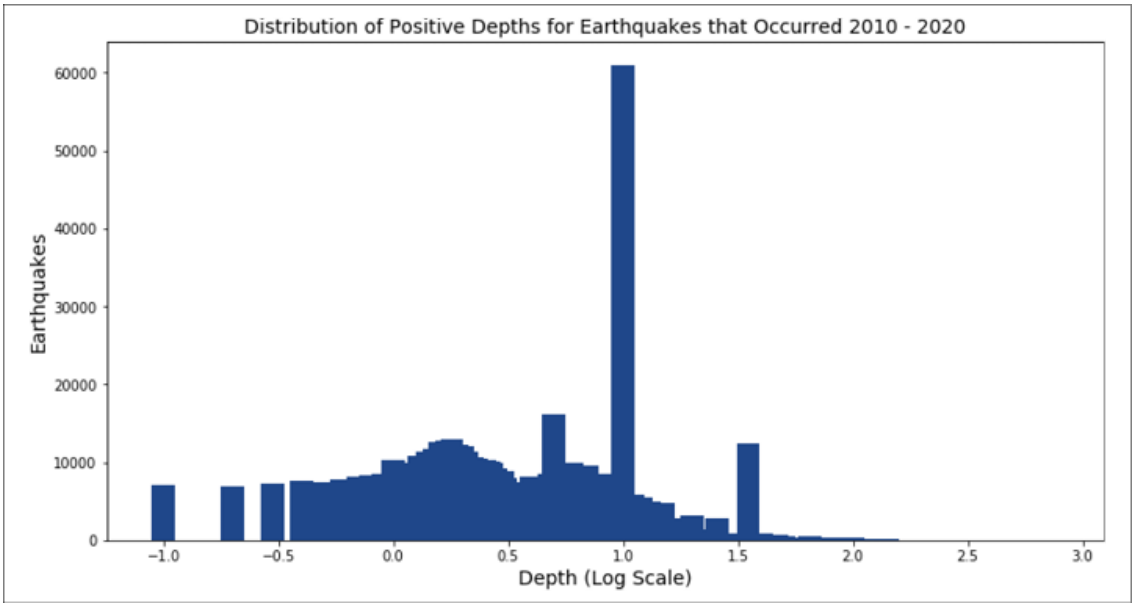


Figura 6.13: Distribuição de terremotos por profundidade em escala de log.

Outros tipos de transformações de escala, embora não necessariamente apropriados para a remoção de valores discrepantes, podem ser obtidos com SQL. Os mais comuns são:

- Raiz quadrada: use a função `sqrt`
- Raiz cúbica: use a função `cbrt`
- Transformação recíproca: 1 / `field_name`

Alteração das unidades, como de polegadas para pés ou de libras para quilogramas: multiplique ou divida pelo fator de conversão apropriado com * ou /.

O reescalonamento pode ser feito em código SQL ou com frequência também é feito alternativamente no software ou na linguagem de codificação usada no gráfico. A transformação para log é particularmente útil quando há uma grande distribuição de valores positivos e os padrões cuja detecção é importante estão nos valores menores.

Como em todas as análises, a decisão de como manipular as anomalias vai depender da finalidade e das informações do contexto ou da área que você tiver sobre o conjunto de dados. Remover valores discrepantes é o método mais simples, mas para a retenção de todos os registros, técnicas como a winsorização e o reescalonamento funcionam bem.

Conclusão

A detecção de anomalias é uma prática comum na análise. O objetivo pode ser detectar os valores discrepantes, ou manipulá-los a fim de preparar um conjunto de dados para a análise posterior. Nos dois casos, as ferramentas básicas da classificação, cálculo de percentis e criação de gráficos com a saída de consultas SQL pode ajudar a encontrar valores discrepantes com eficiência. Há vários tipos de anomalias, com os valores extremos, os picos de atividade inesperados e as ausências anormais sendo os mais comuns. O conhecimento da área quase sempre é útil no decorrer do processo de encontrar e coletar informações sobre as causas de anomalias. As opções para o tratamento de anomalias incluem a investigação, a remoção, a substituição por valores alternativos e o reescalonamento dos dados. A opção escolhida vai depender do objetivo, mas qualquer um desses caminhos pode ser utilizado com SQL. No próximo capítulo, voltaremos nossa atenção para a experimentação, na qual o objetivo é descobrir se um grupo de participantes inteiro difere da norma do grupo de controle.

---

1 *https://www.mathsisfun.com/data/standard-deviation-formulas.html* tem uma boa explicação.

CAPÍTULO 7

Análise experimental

A *experimentação*, também conhecida como *teste A/B* ou *teste de separação (split testing)*, é considerada o padrão ouro para o estabelecimento de causalidade. Grande parte do trabalho de análise de dados envolve o estabelecimento de correlações: é mais provável que uma coisa ocorra quando outra coisa também ocorrer, seja uma ação, um atributo ou um padrão sazonal. No entanto, você já deve ter ouvido o ditado "a correlação não implica causa", e é exatamente esse problema da análise de dados que a experimentação tenta resolver.

Todos os experimentos começam com uma *hipótese*: um palpite sobre a mudança comportamental que resultará de alguma alteração em um produto, processo ou mensagem. A alteração pode ser em uma interface de usuário, no fluxo de integração de um novo usuário, em um algoritmo que dê recomendações, nas mensagens ou no timing de uma campanha de marketing ou em qualquer outra área. Se a organização a construiu ou tem controle sobre ela, pelo menos teoricamente poderemos usá-la em experimentos. Geralmente as hipóteses são baseadas em outro trabalho de análise de dados. Por exemplo, poderíamos descobrir que um percentual alto de pessoas saiu da fila de pagamento de um produto e especular que mais pessoas poderiam concluir o processo de pagamento se o número de etapas fosse reduzido.

O segundo elemento necessário para qualquer experimento é uma *métrica de sucesso*. A mudança comportamental que previmos pode estar relacionada ao preenchimento de um formulário, à conversão em compras, à taxa de cliques, à retenção, ao engajamento ou qualquer outro comportamento que seja importante para a missão da organização. A métrica de sucesso deve quantificar esse comportamento, ser razoavelmente fácil de medir e ser suficientemente sensível para detectar uma mudança. A taxa de cliques, a finalização da compra e o tempo de conclusão de um processo geralmente são boas métricas de sucesso. A

retenção e a satisfação do cliente costumam ser métricas de sucesso menos adequadas, apesar de serem muito importantes, porque com frequência são influenciadas por vários fatores além daquele que está sendo testado em um experimento individual e, portanto, são menos sensíveis às mudanças que gostaríamos de verificar. Normalmente boas métricas de sucesso são as que você já rastreia como parte do processo de conhecimento da saúde empresarial ou organizacional.

Você deve estar se perguntando por que um experimento pode ter várias métricas de sucesso. Certamente com SQL podemos gerar muitos cálculos e métricas diferentes. No entanto, é preciso tomar cuidado com o problema das múltiplas comparações. Não darei uma explicação completa aqui, mas essencialmente significa que, quanto maior o número de locais nos quais você procurar uma alteração significativa, mais provável será que encontre uma. Calcule uma métrica e você pode ou não encontrar uma alteração significativa em uma das variantes do experimento. Calcule 20 métricas e haverá uma boa chance de pelo menos uma exibir relevância, tenha ou não o experimento algo a ver com essa métrica. Como regra geral, é preciso que haja uma ou talvez duas métricas de sucesso principais. Uma a cinco métricas adicionais podem ser usadas para a proteção contra inconveniências (downside protection). Elas costumam ser chamadas de *métricas de guardrail*. Por exemplo, você poderia querer assegurar que um experimento não prejudique o tempo de carregamento das páginas, ainda que o objetivo não seja melhorá-lo.

O terceiro elemento da experimentação seria um sistema que atribuísse aleatoriamente entidades a um grupo de controle ou a um grupo experimental de variantes e alterasse a experiência de acordo. Esse tipo de sistema também é chamado de *sistema de coorte*. Vários fornecedores de software oferecem ferramentas de análise experimental e coorte, embora algumas organizações prefiram construí-las internamente para ter mais flexibilidade. De qualquer forma, para a execução da análise experimental com SQL, os dados de atribuição no nível de entidade devem fluir para uma tabela do banco de dados que também contenha dados comportamentais.

As discussões de experimentos deste capítulo se referem especificamente aos experimentos online, nos quais a atribuição de variantes ocorre por meio de um sistema de computador e o comportamento é rastreado digitalmente. É claro que há muitos tipos de experimentos executados para disciplinas

> científicas e de ciências sociais. Uma diferença essencial é que as métricas de sucesso e os comportamentos que são examinados em experimentos online geralmente já são acompanhados para outras finalidades, enquanto em muitos estudos científicos, o comportamento resultante é rastreado especificamente para o experimento e somente enquanto ele durar. No caso dos experimentos online, às vezes precisamos ser criativos para encontrar métricas que sejam boas substitutas e causem impacto quando uma medição direta não for possível.

Com uma hipótese, uma métrica de sucesso e um sistema de coorte de variantes definidos, você pode executar experimentos, coletar os dados, e analisar os resultados usando SQL.

Vantagens e limites da análise experimental com SQL

O uso do SQL é útil para a análise de experimentos. Em muitos casos de análise experimental, os dados de coorte e os dados comportamentais do experimento já estão fluindo para o banco de dados, o que torna o SQL uma escolha natural. Geralmente as métricas de sucesso já fazem parte do vocabulário de criação de relatórios e análise de uma organização, com as consultas SQL já desenvolvidas. Adicionar dados de atribuição de variantes à lógica de consulta existente costuma ser relativamente fácil.

O SQL é uma boa opção para a automação dos relatórios de resultados dos experimentos. A mesma consulta pode ser executada para cada experimento, com a substituição do nome ou do identificador do experimento na cláusula *WHERE*. Muitas organizações com altos volumes de experimentos criaram relatórios padronizados para acelerar as leituras e simplificar o processo de interpretação.

Embora o SQL seja útil para muitas das etapas envolvidas na análise experimental, ele apresenta uma grande deficiência: não consegue calcular a significância estatística. Vários bancos de dados permitem que os desenvolvedores estendam a funcionalidade do SQL com *UDFs* (user-defined functions, funções definidas pelo usuário). As UDFs podem usar testes estatísticos de linguagens como Python, mas não fazem parte do escopo deste livro. Uma boa opção seria calcular a síntese estatística em SQL e depois usar uma calculadora online como a fornecida em *Evanmiller.org* (*https://oreil.ly/3uspA*) para determinar se o resultado do experimento é estatisticamente significativo.

Por que correlação não é causa: como os valores podem estar relacionados uns com os outros

É mais fácil provar que dois valores estão correlacionados (eles aumentam ou diminuem juntos, ou um existe principalmente na presença do outro) do que provar que um *causa* o outro. Por que isso ocorre? Embora nossos cérebros estejam estruturados para detectar causalidade, na verdade há cinco maneiras pelas quais dois valores, X e Y, podem estar relacionados um com o outro:

1. *X causa Y:* É claro que é isso que todos nós tentamos encontrar. Por meio de alguns mecanismos, Y é o resultado de X.
2. *Y causa X:* O relacionamento está lá, mas a direção da causalidade está invertida. Por exemplo, os guarda-chuvas não causam chuva, mas a presença de chuva faz as pessoas usarem guarda-chuvas.
3. *X e Y têm uma causa comum:* Os valores estão relacionados porque há uma terceira variável que os explica. Tanto as vendas de sorvetes quanto o uso de aparelhos de ar-condicionado aumentam no verão, mas um não causa o outro. As temperaturas mais altas causam os dois aumentos.
4. *Existe um loop de feedback entre X e Y:* Quando Y aumenta, X aumenta para compensar, o que por sua vez leva ao aumento de Y e assim por diante. Isso pode ocorrer quando um cliente está no processo de mudar de fornecedor de um serviço. Menos interações levam a menos itens para serem sugeridos ou lembrados, o que leva a menos interações e assim por diante. A falta de recomendações causa menos engajamento ou é o contrário?
5. *Não há relacionamento; é apenas aleatório:* Se você procurar bastante, encontrará métricas que estão correlacionadas ainda que não haja um relacionamento real entre elas.

Conjunto de dados

Neste capítulo, usaremos um conjunto de dados de um jogo para celular da empresa fictícia Tanimura Studios. Há quatro tabelas. A tabela `game_users` contém registros das pessoas que baixaram o jogo, junto com a data e o país. Uma amostra dos dados é exibida na Figura 7.1.

*	user_id	created	country
1	1000	2020-01-01	Canada
2	1001	2020-01-01	United States
3	1002	2020-01-01	Canada
4	1003	2020-01-01	Australia
5	1004	2020-01-01	Germany
6	1005	2020-01-01	United States
7	1006	2020-01-01	United States
8	1007	2020-01-01	Canada
9	1008	2020-01-01	Australia
10	1009	2020-01-01	United States

Figura 7.1: Amostra da tabela `game_users`.

A tabela `game_actions` contém registros do que os usuários fizeram no jogo. Uma amostra dos dados é exibida na Figura 7.2.

*	user_id	action	action_date
1	1000	email_optin	2020-01-01
2	1000	onboarding complete	2020-01-01
3	1001	email_optin	2020-01-01
4	1001	onboarding complete	2020-01-01
5	1002	onboarding complete	2020-01-01
6	1003	onboarding complete	2020-01-01
7	1003	email_optin	2020-01-01
8	1004	onboarding complete	2020-01-01
9	1005	onboarding complete	2020-01-01
10	1005	email_optin	2020-01-01

Figura 7.2: Amostra da tabela `game_actions`.

A tabela `game_purchases` rastreia as compras de moeda em dólares americanos no jogo. Uma amostra dos dados é exibida na Figura 7.3.

*	user_id	purch_date	amount
1	1009	2020-01-10	50.00
2	1009	2020-01-02	2.99
3	1009	2020-01-02	10.00
4	1010	2020-01-16	25.00
5	1010	2020-01-22	25.00
6	1010	2020-01-09	50.00
7	1022	2020-01-07	25.00
8	1035	2020-01-07	10.00
9	1035	2020-01-02	2.99
10	1035	2020-01-01	2.99

Figura 7.3: Amostra da tabela `game_purchases`.

Para concluir, a tabela `exp_assignment` contém registros das variantes às

quais os usuários foram atribuídos para um experimento específico. Uma amostra dos dados é exibida na Figura 7.4.

*	exp_name	user_id	exp_date	variant
1	Onboarding	1000	2020-01-01	control
2	Onboarding	1001	2020-01-01	variant 1
3	Onboarding	1002	2020-01-01	control
4	Onboarding	1003	2020-01-01	variant 1
5	Onboarding	1004	2020-01-01	control
6	Onboarding	1005	2020-01-01	variant 1
7	Onboarding	1006	2020-01-01	control
8	Onboarding	1007	2020-01-01	variant 1
9	Onboarding	1008	2020-01-01	control
10	Onboarding	1009	2020-01-01	control

Figura 7.4: Amostra da tabela `exp_assignment`.

Todos os dados dessas tabelas são fictícios, criados com geradores de números aleatórios, embora a estrutura seja semelhante à que você veria no banco de dados de uma empresa de jogos digitais real.

Tipos de experimentos

Existe um amplo grupo de experimentos. Se você puder alterar algo que um usuário, cliente, constituinte ou outra entidade vivenciará, teoricamente poderá testar essa alteração. Do ponto de vista da análise, há dois tipos principais de experimentos: os com resultados binários e os com resultados contínuos.

Experimentos com resultados binários: o teste qui-quadrado

Como era de se esperar, um experimento com resultado binário só tem dois resultados: uma ação é ou não executada. Um usuário conclui ou não um fluxo de registro. Um consumidor clica ou não em um site. Um aluno se gradua ou não. Para esses tipos de experimentos, calculamos a proporção de cada variante que conclui a ação. O numerador é a quantidade de participantes que concluíram a ação, enquanto o denominador são todas as unidades que foram expostas. Essa métrica também é descrita como uma taxa: taxa de conclusão, taxa de cliques, taxa de graduação e assim por diante.

Para determinar se as taxas das variantes são estatisticamente diferentes,

podemos usar o *teste qui-quadrado*, que é um teste estatístico para variáveis categóricas.[1] Geralmente os dados de um teste qui-quadrado são exibidos na forma de uma *tabela de contingência*, que mostra a frequência de observações na interseção de dois atributos. Ela parecerá uma tabela dinâmica para quem estiver familiarizado com esse tipo de tabela.

Examinaremos um exemplo, usando nosso conjunto de dados de jogo para celular. Um gerente de produto introduziu uma nova versão do fluxo de integração (onboarding), uma série de telas que ensinam a um novo jogador como o jogo funciona. O gerente de produto espera que a nova versão aumente o número de jogadores que concluirão a integração e que começarão sua primeira sessão do jogo. A nova versão foi introduzida em um experimento chamado "Onboarding" que atribuiu usuários ao grupo control ou variant 1, como registrado na tabela `exp_assignment`. Um evento chamado "onboarding complete" da tabela `game_actions` indica se um usuário concluiu o fluxo de integração.

A tabela de contingência mostra a frequência na interseção da atribuição de variantes (control ou variant 1) e se a integração foi ou não concluída. Podemos usar uma consulta para encontrar os valores da tabela. Aqui contaremos o número de usuários com ou sem uma ação "onboarding complete" e faremos o agrupamento por `variant`:

```
SELECT a.variant
,count(case when b.user_id is not null then a.user_id end) as
  completed
,count(case when b.user_id is null then a.user_id end) as
  not_completed
FROM exp_assignment a
LEFT JOIN game_actions b on a.user_id = b.user_id
and b.action = 'onboarding complete'
WHERE a.exp_name = 'Onboarding'
GROUP BY 1
;
variant    completed  not_completed
---------  ---------  -------------
control    36268      13629
variant 1  38280      11995
```

A inclusão de totais para cada linha e coluna transforma essa saída em uma tabela de contingência, como na Figura 7.5.

	Integração concluída?		
Variante	Sim	Não	Total
Controle	36.268	13.629	49.897
Variante 1	38.280	11.995	50.275
Total	74.548	25.624	100.172

Figura 7.5: Tabela de contingência de conclusões de integração.

Para fazer uso de uma das calculadoras de significância online, precisaremos do número de sucessos, ou das vezes em que a ação foi executada, e do número total de integrantes da coorte de cada variante. O código SQL para o cálculo dos pontos de dados requeridos é simples. A variante atribuída e a contagem de usuários dessa variante são consultadas na tabela `exp_assignment`. Depois usamos *LEFT JOIN* com a tabela `game_actions` para encontrar a contagem de usuários que concluíram a integração. A *LEFT JOIN* é necessária porque o esperado é que nem todos os usuários concluam a ação relevante. Por fim, encontramos o percentual de conclusão de cada variante dividindo o número de usuários que concluíram a ação pelo número total de integrantes da coorte:

```
SELECT a.variant
,count(a.user_id) as total_cohorted
,count(b.user_id) as completions
,count(b.user_id) / count(a.user_id) as pct_completed
FROM exp_assignment a
LEFT JOIN game_actions b on a.user_id = b.user_id
and b.action = 'onboarding complete'
WHERE a.exp_name = 'Onboarding'
GROUP BY 1
;
variant    total_cohorted  completions  pct_completed
---------  --------------  -----------  -------------
control    49897           36268        0.7269
variant 1  50275           38280        0.7614
```

Podemos ver que variant 1 tem um número maior de conclusões do que o

vivenciado por control, com 76,14% de conclusões em comparação com 72,69%. No entanto, essa diferença é estatisticamente significativa e nos permite rejeitar a hipótese de que não há diferença? Para saber, levamos nossos resultados para uma calculadora online e confirmamos que a taxa de conclusão de variant 1 foi significativamente mais alta a um nível de confiança de 95% do que a taxa de conclusão de control. Variant 1 pode ser declarado o vencedor.

Normalmente um nível de confiança de 95% é usado, embora ele não seja a única opção. Há muitos artigos e discussões online sobre o significado dos níveis de confiança, que nível usar e os ajustes feitos em cenários em que diversas variantes estejam sendo comparadas com um grupo de controle.

Os experimentos com resultado binário seguem esse padrão básico. Calculam os sucessos ou as conclusões assim como o total de membros de cada variante. O SQL usado para derivar os eventos de sucesso pode ser mais complicado dependendo das tabelas e de como as ações são armazenadas no banco de dados, mas a saída é consistente. A seguir, examinaremos experimentos com resultados contínuos.

Experimentos com resultados contínuos: o teste t

Muitos experimentos tentam melhorar *métricas contínuas*, em vez dos resultados binários discutidos na última seção. As métricas contínuas podem usar diferentes valores. Alguns exemplos seriam a quantia gasta pelos clientes, o tempo gasto na página e os dias durante os quais uma aplicação foi usada. Os sites de e-commerce estão sempre tentando aumentar as vendas e, portanto, podem fazer experimentos com páginas de produtos ou fluxos de finalização de compras. Os sites de conteúdo podem testar o layout, a navegação e os títulos para tentar aumentar o número de stories lidas. Uma empresa que estivesse executando uma aplicação poderia fazer uma campanha de remarketing para lembrar os usuários de voltarem para a aplicação.

Nestes e em outros experimentos com métricas de sucesso contínuas, o objetivo é descobrir se os valores das médias de cada variante diferem uns dos outros de maneira estatisticamente significativa. O teste estatístico relevante é o *teste t para duas amostras*, que determina se podemos rejeitar a

hipótese nula de que as médias são iguais com um intervalo de confiança definido, geralmente de 95%. O teste estatístico tem três entradas, todas elas fáceis de calcular com SQL: a média, o desvio-padrão e a contagem de observações.

Examinaremos um exemplo usando os dados de nosso jogo. Na última seção, verificamos se um novo fluxo de integração aumentava a taxa de conclusão. Agora consideraremos se esse novo fluxo aumentou o gasto do usuário na moeda do jogo. A métrica de sucesso é a quantia gasta, logo precisamos calcular a média e o desvio-padrão desse valor para cada variante. Primeiro temos de calcular a quantia por usuário, já que os usuários podem fazer várias compras. Recupere a atribuição de coorte na tabela `exp_assignment` e conte os usuários. Em seguida, use uma *LEFT JOIN* para a tabela `game_purchases` para coletar os dados das quantias. A *LEFT JOIN* é necessária porque nem todos os usuários estão fazendo compras, mas mesmo assim precisamos incluí-los nos cálculos da média e do desvio-padrão. Para usuários sem compras, a quantia é configurada com o padrão 0 com `coalesce`. Já que as funções `avg` e `stddev` ignoram nulos, o padrão 0 é requerido para assegurar que esses registros sejam incluídos. A consulta externa resume os valores da saída por variante:

```
SELECT variant
,count(user_id) as total_cohorted
,avg(amount) as mean_amount
,stddev(amount) as stddev_amount
FROM
(
    SELECT a.variant
    ,a.user_id
    ,sum(coalesce(b.amount,0)) as amount
    FROM exp_assignment a
    LEFT JOIN game_purchases b on a.user_id = b.user_id
    WHERE a.exp_name = 'Onboarding'
    GROUP BY 1,2
) a
GROUP BY 1
;
```

```
variant    total_cohorted  mean_amount  stddev_amount
---------  --------------  -----------  -------------
control    49897           3.781        18.940
variant 1  50275           3.688        19.220
```

Se passarmos esses valores para uma calculadora online, veremos que não há diferença significativa entre os grupos de controle e de variantes em um intervalo de confiança de 95%. O grupo "variant 1" parece ter aumentado as taxas de conclusão de integração, mas não a quantia gasta.

Outra questão que poderíamos considerar é se variant 1 provocou algum impacto entre os usuários que concluíram a integração. Quem não conclui a integração não consegue acessar o jogo e, portanto, não tem oportunidade de fazer uma compra. Para responder a essa pergunta, podemos usar uma consulta semelhante à anterior, mas adicionando uma *INNER JOIN* para a tabela `game_actions` para restringir os usuários contados apenas àqueles que têm uma ação "onboarding complete":

```
SELECT variant
,count(user_id) as total_cohorted
,avg(amount) as mean_amount
,stddev(amount) as stddev_amount
FROM
(
    SELECT a.variant
    ,a.user_id
    ,sum(coalesce(b.amount,0)) as amount
    FROM exp_assignment a
    LEFT JOIN game_purchases b on a.user_id = b.user_id
    JOIN game_actions c on a.user_id = c.user_id
     and c.action = 'onboarding complete'
    WHERE a.exp_name = 'Onboarding'
    GROUP BY 1,2
) a
GROUP BY 1
;
variant    total_cohorted  mean_amount  stddev_amount
---------  --------------  -----------  -------------
control    36268           5.202        22.049
```

```
variant 1  38280          4.843        21.899
```

A passagem desses valores para a calculadora revela que a média do grupo de controle tem relevância estatística mais alta do que a de variant 1 em um intervalo de confiança de 95%. Esse resultado pode parecer intrigante, mas ilustra por que é tão importante definir a métrica de sucesso de um experimento antecipadamente. O grupo variant 1 do experimento teve um efeito positivo sobre a conclusão da integração e, portanto, pode ser considerado um sucesso. No entanto, não gerou impacto sobre o nível de gasto geral. Isso pode ter ocorrido devido a uma mix shift: os usuários adicionais que passaram pela integração em variant 1 estavam menos propensos a gastar. Se a hipótese subjacente era a de que o aumento nas taxas de conclusão da integração aumentaria a receita, o experimento não deve ser considerado um sucesso e os gerentes de produto devem trazer novas ideias para serem testadas.

Desafios dos experimentos e opções para o resgate de experimentos que falharam

Embora a experimentação seja o padrão ouro para o conhecimento de causalidades, há várias maneiras pelas quais os experimentos podem dar errado. Se a premissa inteira estiver incorreta, não há muito que o SQL possa fazer para ajudar. Se a natureza da falha for, em grande parte, técnica, talvez possamos consultar os dados de uma forma que ajuste ou exclua pontos de dados problemáticos e interpretar alguns resultados. A execução de experimentos tem um custo relacionado ao tempo gasto pelos engenheiros, designers ou vendedores que criaram as variantes. Também há o *custo de oportunidade*, ou o benefício perdido que poderia ter sido ganho pelo direcionamento dos clientes por um caminho de conversão ou para uma experiência de uso de produto ideal. No nível prático, o uso de SQL para ajudar a organização pelo menos a aprender algo com um experimento geralmente é tempo bem gasto.

Atribuição de variantes

A atribuição aleatória de unidades de experimento (que podem ser usuários, sessões ou outras entidades) a grupos de controle e de variantes

é um dos principais elementos da experimentação. No entanto, às vezes ocorrem erros no processo de atribuição, devido a uma falha na especificação do experimento, a uma falha técnica ou a uma limitação no software de coorte. Como resultado, os grupos de controle e de variantes podem ter tamanhos diferentes, um número de entidades menor do que o esperado pode entrar na coorte, ou a atribuição pode não ocorrer de maneira realmente aleatória.

Em algumas situações, o SQL pode ajudar a salvar um experimento em que unidades demais tiverem entrado na coorte. Vi isso ocorrer em um experimento que deveria ter sido direcionado apenas a novos usuários, mas todos os usuários entraram na coorte. Outra maneira de esse problema ocorrer é quando um experimento testa algo que só um subconjunto dos usuários verá, porque se encontra a alguns cliques à frente na experiência ou porque certas condições precisem ser atendidas, como uma compra anterior. Devido a limitações técnicas, todos os usuários entram na coorte, ainda que uma parte deles não possa ver o tratamento experimental mesmo que esteja nesse grupo. A solução é adicionar uma *JOIN* ao SQL para restringir os usuários ou as entidades apenas àqueles que sejam elegíveis. Por exemplo, poderíamos adicionar uma *INNER JOIN* a uma tabela ou subconsulta que tivesse a data de registro do usuário e definir uma condição *WHERE* para excluir usuários cujo registro tenha ocorrido muito antes do evento de criação da coorte e, portanto, não sejam considerados novos. A mesma estratégia pode ser usada quando uma condição específica precisar ser atendida mesmo que apenas para a visualização do experimento. Restrinja as entidades incluídas excluindo as que não sejam elegíveis por meio de *JOIN*s e condições *WHERE*. Após fazê-lo, você deve verificar se a população resultante é uma amostra suficientemente grande para produzir resultados significativos.

Se poucos usuários ou entidades fizerem parte da coorte, será importante verificar se a amostra é suficientemente grande para produzir resultados relevantes. Se não for, execute o experimento novamente. Se a amostra for suficientemente grande, uma segunda consideração seria saber se há uma tendência para quem ou o que entrou na coorte. Como exemplo, vi casos em que usuários de certos navegadores ou de versões de aplicações mais

antigas não entraram na coorte devido a limitações técnicas. Se as populações que foram excluídas não forem aleatórias e representarem diferenças no local, na experiência técnica ou no status econômico, será importante considerar tanto o tamanho dessa população em relação ao restante quanto se algum ajuste deve ser feito para incluí-la na análise final.

Algo que também pode ocorrer é o sistema de atribuição de variantes apresentar falhas e as entidades não serem atribuídas aleatoriamente. Isso não é comum na maioria das ferramentas de experimentação modernas, mas, se ocorrer, invalidará o experimento inteiro. Resultados "bons demais para ser verdade" podem sinalizar um problema de atribuição de variantes. Por exemplo, vi casos em que usuários altamente engajados eram atribuídos acidentalmente tanto ao grupo de controle quanto ao grupo de tratamento devido a uma alteração na configuração do experimento. Uma criação de perfis cuidadosa pode verificar se entidades foram atribuídas a múltiplas variantes ou se usuários com alto ou baixo engajamento antes do experimento foram agrupados em uma variante específica.

A execução de um teste A/A pode ajudar a revelar falhas no software de atribuição de variantes. Nesse tipo de teste, as entidades são divididas em coortes e métricas de sucesso são comparadas, como em qualquer outro experimento. No entanto, nenhuma alteração é feita na experiência e as duas coortes recebem a experiência do grupo de controle. Já que os grupos recebem a mesma experiência, não devemos esperar diferenças significativas na métrica de sucesso. Contudo, se ocorrer uma diferença, uma investigação adicional deve ser feita para a descoberta e a correção do problema.

Valores discrepantes (Outliers)

Testes estatísticos para a análise de métricas de sucesso contínuas dependem de médias. Como resultado, são sensíveis a valores anômalos que sejam excessivamente altos ou baixos. Vi experimentos nos quais a presença de um ou dois clientes particularmente gastadores em uma variante dava a ela uma vantagem estatisticamente significativa em relação às outras. Sem esses poucos gastadores excessivos, o resultado poderia ser neutro ou até mesmo se inverter. Quase sempre, queremos saber se um

tratamento produz efeito em um grupo de indivíduos e, portanto, ajustar esses valores discrepantes pode fazer um experimento ter um resultado mais significativo.

Discutimos a detecção de anomalias no Capítulo 6, e a análise experimental é outra situação em que essas técnicas podem ser aplicadas. Valores anômalos podem ser determinados pela análise dos resultados do experimento ou pela busca da taxa básica antes do experimento. Os valores discrepantes podem ser removidos por meio de uma técnica como a winsorização (também discutida no Capítulo 6), que elimina valores que ultrapassam um limite, como o 95º ou o 99º percentil. Isso pode ser feito em SQL antes da entrada no restante da análise experimental.

Outra opção para o tratamento de valores discrepantes em métricas de sucesso contínuas seria transformar a métrica de sucesso em um resultado binário. Por exemplo, em vez de comparar o gasto médio nos grupos de controle e de variantes, que pode ser distorcido devido a alguns gastadores excessivos, compare a taxa de compras entre os dois grupos e siga o procedimento discutido na seção sobre experimentos com resultados binários. Poderíamos considerar a taxa de conversão para comprador entre os usuários que concluíram a integração nos grupos control e variant 1 do experimento "Onboarding":

```
SELECT a.variant
,count(distinct a.user_id) as total_cohorted
,count(distinct b.user_id) as purchasers
,count(distinct b.user_id) / count(distinct a.user_id)
as pct_purchased
FROM exp_assignment a
LEFT JOIN game_purchases b on a.user_id = b.user_id
JOIN game_actions c on a.user_id = c.user_id
and c.action = 'onboarding complete'
WHERE a.exp_name = 'Onboarding'
GROUP BY 1
;
variant    total_cohorted  purchasers  pct_purchased
---------  --------------  ----------  -------------
control    36268           4988        0.1000
variant 1  38280           4981        0.0991
```

Ao examinar os números, vemos que, mesmo existindo mais usuários em variant 1, há menos compras. O percentual de usuários que fez compras no grupo de controle é de 10%, em comparação com os 9,91% de variant 1. Agora podemos inserir os pontos de dados em uma calculadora online. A taxa de conversão tem relevância estatisticamente mais alta para o grupo de controle. Nesse caso, ainda que a taxa de compra seja mais alta para o grupo de controle, na prática poderíamos aceitar esse pequeno declínio se acharmos que o fato de mais usuários estarem concluindo o processo de integração traz outros benefícios. A existência de um número maior de participantes pode trazer para o jogo uma melhora no ranking, por exemplo, e os participantes que gostarem falarão dele com seus amigos, duas situações que podem ajudar na expansão e atrair novos jogadores que se tornarão compradores.

A métrica de sucesso também pode ser configurada com um limite, e poderíamos comparar a parcela de entidades que o atingissem. Por exemplo, a métrica poderia ser a leitura de pelo menos três stories ou o uso de uma aplicação pelo menos duas vezes na semana. Podemos construir um número infinito de métricas dessa forma, logo é essencial saber o que é importante e significativo para a organização.

Time Boxing

Geralmente os experimentos são executados no decorrer de várias semanas. Ou seja, indivíduos que entrarem no experimento mais cedo terão uma janela de tempo mais longa para concluir ações associadas à métrica de sucesso. Para controlar isso, podemos aplicar uma *time boxing* – impor um período de tempo fixo em relação à data de entrada no experimento e só considerar ações ocorridas durante essa janela de tempo. Esse conceito também foi abordado no Capítulo 4.

Para experimentos, o tamanho apropriado da time box depende do que estiver sendo medido. A janela de tempo poderia ser de apenas uma hora para a medição de uma ação que costuma ter resposta imediata, como clicar em um anúncio. Para a conversão em compras, geralmente os responsáveis pelo experimento permitem uma janela de tempo de 1 a 7 dias. Janelas mais curtas permitem que os experimentos sejam analisados mais cedo, já que é preciso dar o tempo completo a todas as entidades da

coorte para que elas concluam as ações. As melhores janelas chegam a um equilíbrio entre a necessidade de obter resultados e a dinâmica real da organização. Se normalmente os clientes se decidem por uma compra após alguns dias, considere uma janela de 7 dias; se eles levarem 20 dias ou mais, considere uma janela de 30 dias.

Como demonstração, podemos revisar nosso primeiro exemplo de experimentos com resultados contínuos incluindo apenas compras feitas dentro de 7 dias após o evento de criação da coorte. Lembre-se de que é importante usar a hora em que a entidade foi atribuída a uma variante como ponto de partida da time box. Uma cláusula *ON* adicional foi incluída, restringindo os resultados às compras ocorridas dentro do intervalo "7 days" (7 dias):

```
SELECT variant
,count(user_id) as total_cohorted
,avg(amount) as mean_amount
,stddev(amount) as stddev_amount
FROM
(
    SELECT a.variant
    ,a.user_id
    ,sum(coalesce(b.amount,0)) as amount
    FROM exp_assignment a
    LEFT JOIN game_purchases b on a.user_id = b.user_id
     and b.purch_date <= a.exp_date + interval '7 days'
    WHERE a.exp_name = 'Onboarding'
    GROUP BY 1,2
) a
GROUP BY 1
;
variant    total_cohorted  mean_amount  stddev_amount
---------  --------------  -----------  -------------
control    49897           1.369        5.766
variant 1  50275           1.352        5.613
```

As médias são semelhantes e, na verdade, estatisticamente não apresentam diferença significativa. Nesse exemplo, a conclusão com a time box está em concordância com a conclusão quando não há time box.

Nesse caso, os eventos de compra são relativamente raros. Para métricas que avaliem eventos comuns que se acumulem rapidamente, como visualizações de páginas, cliques, curtidas (likes), e leitura de artigos, usar uma time box pode evitar que usuários que entraram na coorte mais cedo pareçam significativamente "melhores" do que os que entraram depois.

Experimentos com exposição repetida

Em discussões sobre experimentação online, a maioria dos exemplos apresenta o que gosto de chamar de experiências "de execução única": o usuário encontra um tratamento uma vez, reage a ele e não o encontra novamente. O registro de usuários é um exemplo clássico: um consumidor se cadastra em um serviço específico apenas uma vez e, portanto, qualquer alteração feita no processo afeta somente novos usuários. É relativamente fácil analisar testes feitos com essas experiências.

Há outro tipo de experiência que chamo de "exposição repetida", na qual um indivíduo entra em contato com a alteração muitas vezes no decorrer do uso de um produto ou serviço. Em qualquer experimento que envolver essas alterações, o esperado é que os indivíduos as encontrem mais de uma vez. Alterações na interface de usuário de uma aplicação, como na cor, no texto e na inserção de informações e links importantes, são vivenciadas pelos usuários durante todo o uso da aplicação. Programas de marketing por email que enviam para os clientes lembretes ou promoções regularmente também têm essa característica de exposição repetida. Os emails são vistos várias vezes como linhas de assunto na caixa de correio e como conteúdo quando abertos.

Medir experimentos de exposição repetida é mais complicado do que medir experimentos de execução única devido ao efeito novidade e à regressão à média. O *efeito novidade* é a tendência de o comportamento mudar só porque algo é novo, e não porque é necessariamente melhor. *Regressão à média* é a tendência de o fenômeno retornar a um nível médio com o passar do tempo. Como exemplo, alterar qualquer parte de uma interface de usuário tende a aumentar o número de pessoas que interagem com ela, seja a alteração uma nova cor para um botão, um logotipo ou a inserção de funcionalidade. Inicialmente a métrica parece boa, porque a taxa de cliques ou o engajamento aumenta. Esse é o efeito

novidade. No entanto, com o passar do tempo, os usuários se acostumam com a alteração e tendem a clicar na funcionalidade ou a usá-la a taxas que voltam para mais perto da linha de base. Essa é a regressão à média. A pergunta importante que deve ser respondida na execução desse tipo de experimento é se a nova linha de base é mais alta (ou mais baixa) do que a anterior. Uma solução é permitir a passagem de um período de tempo suficientemente longo, no qual você espere que a regressão ocorra, antes de avaliar os resultados. Em alguns casos, o período é de poucos dias; em outros, pode ser de algumas semanas ou meses.

Se houver muitas alterações, ou o experimento ocorrer em série como em campanhas por email ou por correio físico, descobrir se o programa inteiro faz diferença pode ser um desafio. É fácil alegar sucesso quando clientes que recebem uma variante de email específica compram um produto, mas como saber se eles fariam essa compra mesmo se não a recebessem? Uma opção seria definir um grupo de *holdout de longo prazo*. Trata-se de um grupo que é definido para não receber nenhuma mensagem de marketing ou alterações feitas na experiência de uso do produto. Repare que isso é diferente de simplesmente fazer a comparação com usuários que optaram por não receber mensagens de marketing, já que geralmente há alguma tendência em quem opta ou não pelo recebimento. Pode ser complicado definir grupos de holdout de longo prazo, mas há poucas maneiras melhores de medir os efeitos cumulativos de campanhas e alterações em produtos.

Outra opção seria executar a análise de coorte (discutida no Capítulo 4) com as variantes. Os grupos podem ser acompanhados por um período de tempo mais longo, de semanas a meses. Métricas de retenção ou cumulativas podem ser calculadas e testadas para sabermos se os efeitos diferem entre as variantes a longo prazo.

Mesmo com os diversos desafios que podem ser encontrados nos experimentos, eles ainda são a melhor maneira de testar e comprovar a causalidade em alterações feitas em experiências que vão da criação e recebimento de mensagens de marketing ao uso interno do produto. No entanto, geralmente encontramos situações na análise de dados que estão longe de serem as ideais; logo, a seguir, examinaremos algumas opções de análise para quando não for possível fazer o teste A/B.

Se não for possível executar experimentos controlados: análises alternativas

Experimentos randomizados são o padrão ouro quando a intenção é ir além da correlação para estabelecer causalidade. Contudo, há várias razões para não ser possível executar um experimento randomizado. Pode ser antiético dar um tratamento diferente para cada grupo, principalmente em contextos médicos ou educacionais. Requisitos regulatórios podem impedir experimentos em outros contextos, como o de serviços financeiros. Pode haver razões práticas, como a dificuldade de restringir o acesso ao tratamento de uma variante a apenas um grupo randomizado. É sempre útil considerar se existem partes cuja testagem seria válida ou que sejam testáveis respeitando-se os limites éticos, regulatórios e práticos. O texto usado, a inserção de elementos e outros aspectos do design são alguns exemplos.

Uma segunda situação em que é impossível executar a experimentação é se uma alteração tiver ocorrido no passado e os dados já tiverem sido coletados. Exceto pela reversão da alteração, voltar e executar um experimento não é uma opção. Às vezes um analista ou cientista de dados não está disponível para ajudar na experimentação. Em mais de uma situação, ingressei em uma organização e me solicitaram para reverter os resultados de alterações cuja compreensão teriam sido mais fácil se houvesse um grupo de holdout. Em outras, a alteração tinha sido involuntária. Alguns exemplos seriam indisponibilidades que afetaram alguns ou todos os clientes, erros em formulários e desastres naturais como tempestades, terremotos e incêndios.

Embora conclusões casuais não sejam tão sólidas em situações em que não houve experimento, existem alguns métodos de análise quase experimentais que podem ser usados para a obtenção de insights a partir dos dados. Eles se baseiam na construção de grupos dos dados disponíveis que cheguem o mais próximo possível das condições de "controle" e "tratamento".

Pré/pós-análise

Uma pré/pós-análise compara populações idênticas ou semelhantes antes

e depois de uma alteração. A medição da população antes da alteração é usada como controle, enquanto a medição após a alteração é usada como variante ou tratamento.

A pré/pós-análise funciona melhor quando há uma alteração claramente definida ocorrida em uma data conhecida e assim os grupos pré e pós-alteração podem ser divididos nitidamente. Nesse tipo de análise, você precisará definir por quanto tempo fará a medição antes e depois da alteração, mas os períodos devem ser iguais ou quase iguais. Por exemplo, se duas semanas tiverem se passado desde que uma alteração foi feita, compare esse período com as duas semanas anteriores à alteração. Considere comparar vários períodos, como uma semana, duas semanas, três semanas e quatro semanas antes e depois da alteração. Se os resultados estiverem em concordância ao longo de todas essas janelas, você poderá ter mais confiança neles do que teria se diferissem.

Acompanharemos um exemplo. Suponhamos que o fluxo de integração de nosso jogo para celular incluísse uma etapa na qual o usuário pudesse marcar uma caixa para indicar se deseja receber emails com novidades sobre o jogo. Essa caixa vinha marcada por padrão, mas uma nova regulação requer que agora ela venha desmarcada. Em 27 de janeiro de 2020, a alteração foi lançada no jogo e queremos saber se ela teve um efeito negativo sobre as taxas de opção pelo recebimento de emails. Para fazê-lo, compararemos as duas semanas anteriores à alteração com as duas semanas posteriores à alteração e veremos se a taxa de opção pelo recebimento teve uma mudança estatisticamente significativa. Poderíamos usar períodos de uma semana ou três semanas, mas o de duas semanas foi escolhido porque é suficientemente longo para permitir alguma variabilidade nos dias úteis e também suficientemente curto para restringir o número de outros fatores que de outra forma poderiam afetar a tendência de os usuários optarem positivamente.

As variantes são atribuídas na consulta SQL por meio de uma instrução CASE: os usuários que foram criados no intervalo anterior à alteração serão rotulados com "pre", enquanto os criados após a alteração receberão o rótulo "post". Em seguida, contaremos o número de usuários de cada grupo da tabela `game_users`. Depois contaremos o número de usuários que optaram positivamente, o que é feito com uma *LEFT JOIN* para a

tabela `game_actions`, restringindo os registros aos que tiverem a ação "email_optin". Dividiremos então os valores para encontrar o percentual de quem optou por receber emails. Gosto de incluir a contagem de dias em cada variante como uma verificação de qualidade, embora não seja necessário para a execução do restante da análise:

```
SELECT
case when a.created between '2020-01-13' and '2020-01-26' then 'pre'
     when a.created between '2020-01-27' and '2020-02-09' then
  'post'
     end as variant
,count(distinct a.user_id) as cohorted
,count(distinct b.user_id) as opted_in
,count(distinct b.user_id) / count(distinct a.user_id) as pct_optin
,count(distinct a.created) as days
FROM game_users a
LEFT JOIN game_actions b on a.user_id = b.user_id
and b.action = 'email_optin'
WHERE a.created between '2020-01-13' and '2020-02-09'
GROUP BY 1
;
variant  cohorted  opted_in  pct_optin  days
-------  --------  --------  ---------  ----
pre      24662     14489     0.5875     14
post     27617     11220     0.4063     14
```

Muitos bancos de dados reconhecem datas inseridas como strings, como em `'2020-01-13'`. Se seu banco de dados não reconhecer, converta a string para uma data usando uma destas opções:

```
cast('2020-01-13' as date)
date('2020-01-13')
'2020-01-13'::date
```

Nesse caso, podemos ver que os usuários que passaram pelo fluxo de integração antes da alteração tiveram uma taxa de opção pelo recebimento de emails muito mais alta – 58,75% em comparação com os 40,63% pós-alteração. A inserção dos valores em uma calculadora online resulta na confirmação de que a taxa do grupo "pre" tem relevância estatisticamente

mais alta do que a taxa do grupo "post". Nesse exemplo, não há muito que a empresa do jogo possa fazer, já que a alteração ocorreu devido a um regulamento. Testes posteriores poderiam determinar se o fornecimento de amostras de conteúdo ou outras informações sobre o programa de email encorajariam mais jogadores a optar positivamente, se esse for um objetivo da empresa.

Ao executar uma pré/pós-análise, lembre-se de que outros fatores além da alteração que você está tentando conhecer podem causar o aumento ou a diminuição da métrica. Eventos externos, sazonalidade, promoções de marketing etc. podem alterar drasticamente o ambiente e a mentalidade dos clientes até mesmo dentro de algumas semanas. Como resultado, esse tipo de análise não é tão bom como um experimento randomizado real para o fornecimento de causalidade. No entanto, pode ser uma das poucas opções de análise disponíveis e gerar hipóteses de trabalho para serem testadas e refinadas em futuros experimentos controlados.

Análise experimental natural

Um experimento natural ocorre quando as entidades têm experiências diferentes por meio de algum processo que se aproxime da aleatoriedade. Um grupo recebe a experiência normal ou de controle e o outro recebe alguma variação que pode ter um efeito positivo ou negativo. Geralmente isso não é intencional, como quando um bug é introduzido em um software, ou quando um evento ocorre em um local, mas não em outros. Para esse tipo de análise ser válido, temos de poder determinar claramente quais entidades foram expostas. Além disso, é necessário um grupo de controle que seja o máximo possível semelhante ao grupo exposto.

O SQL pode ser usado na construção das variantes e no cálculo do tamanho das coortes e de eventos de sucesso no caso de um evento com resultado binário (ou da média, do desvio-padrão e dos tamanhos de populações no caso de um evento com resultado contínuo). Os resultados podem ser inseridos em uma calculadora online como ocorre com qualquer experimento.

Como exemplo com nosso conjunto de dados de videogame, suponhamos que, durante o período de tempo dos dados, os usuários do Canadá tivessem recebido acidentalmente uma oferta diferente na página de

compras de moeda virtual na primeira vez que a examinaram: um zero a mais foi adicionado ao número de moedas virtuais de cada pacote. Logo, por exemplo, em vez de 10 moedas o usuário recebeu 100 moedas do jogo, ou em vez de 100 ele recebeu 1.000 moedas e assim por diante. A pergunta para a qual gostaríamos de uma resposta é se os canadenses foram convertidos em compradores a uma taxa maior do que a de outros usuários. Em vez de fazer a comparação com a base de usuários inteira, compararemos apenas com os usuários dos Estados Unidos. Os países ficam geograficamente próximos, e a maioria dos seus usuários fala o mesmo idioma – e para o exemplo, presumiremos que outra análise foi feita e mostrou que seu comportamento é semelhante, enquanto o comportamento de usuários de outros países é suficientemente diferente para que eles sejam excluídos.

Para a execução da análise, as "variantes" são criadas a partir de qualquer que seja a característica diferenciadora – neste caso, o campo `country` da tabela `game_users` – mas lembre-se de que um código SQL mais complexo pode ser necessário, dependendo do conjunto de dados. As contagens de usuários da coorte, e dos que fizeram compras, são calculadas da mesma forma que vimos anteriormente:

```
SELECT a.country
,count(distinct a.user_id) as total_cohorted
,count(distinct b.user_id) as purchasers
,count(distinct b.user_id) / count(distinct a.user_id)
as pct_purchased
FROM game_users a
LEFT JOIN game_purchases b on a.user_id = b.user_id
WHERE a.country in ('United States','Canada')
GROUP BY 1
;
country        total_cohorted  purchasers  pct_purchased
-------------  --------------  ----------  -------------
Canada         20179           5011        0.2483
United States  45012           4958        0.1101
```

A parcela de usuários do Canadá que fizeram compras é realmente mais alta – 24,83%, em comparação com os 11,01% nos Estados Unidos. A

inserção desses valores em uma calculadora online confirma que a taxa de conversão no Canadá tem relevância estatisticamente mais alta com um intervalo de confiança de 95%.

A parte mais difícil da análise de um experimento natural tende a ser encontrar uma população comparável e mostrar que as duas populações são suficientemente semelhantes para embasar as conclusões do teste estatístico. Embora seja praticamente impossível provar que não há fatores de confusão, uma comparação cuidadosa da demografia e do comportamento populacional dá credibilidade aos resultados. Já que um experimento natural não é um experimento aleatório real, a prova de causalidade é mais fraca, e isso deve ser mencionado na apresentação de uma análise desse tipo.

Análise de populações com base em um limite

Podemos ter casos em que um valor limite resulte em algumas pessoas ou outras unidades de interesse receberem um tratamento e outras não. Por exemplo, uma média de pontos específica poderia qualificar alunos para o recebimento de uma bolsa de estudos, um nível de renda específico poderia qualificar famílias para um plano de assistência médica subsidiado, ou uma pontuação alta para risco de rotatividade poderia acionar um representante de vendas para fazer o acompanhamento de um cliente. Nesses casos, podemos nos beneficiar da ideia de que as entidades de cada lado do valor limite devem ser semelhantes. Logo, em vez de comparar as populações inteiras que receberam ou não a recompensa ou intervenção, podemos comparar apenas as que estavam próximas do limite tanto no lado positivo quanto no negativo. O nome formal dessa operação é *RDD* (regression descontinuity design, design de descontinuidade por regressão).

Para executar esse tipo de análise, podemos construir "variantes" dividindo os dados ao redor de um valor limite, de forma semelhante ao que fizemos na pré-/pós análise. Infelizmente, não há uma regra rígida para a amplitude que as faixas de valores devem ter em cada lado do limite. As "variantes" devem ter tamanho semelhante e devem ser suficientemente grandes para permitir que haja significância na análise dos resultados. Uma opção seria executar a análise várias vezes com

alguns intervalos diferentes. Por exemplo, você poderia analisar as diferenças entre o grupo "tratado" e o grupo de controle quando cada grupo tiver entidades que fiquem dentro de 5%, 7,5% e 10% do limite. Se as conclusões dessas análises apresentarem concordância, haverá mais embasamento para as conclusões. No entanto, se não apresentarem, os dados podem ser considerados inconclusivos.

Como ocorre com outros tipos de análise não experimental, os resultados do RDD devem ser considerados como provando a causalidade de maneira menos conclusiva. Possíveis fatores de confusão também devem receber uma atenção cuidadosa. Por exemplo, se clientes que apresentarem alto risco de mudança de fornecedor receberem intervenções de várias equipes, ou um desconto especial para encorajá-los a ficar, além de uma ligação do representante de vendas, os dados podem ser afetados por essas outras alterações.

Conclusão

A análise experimental é uma área rica que geralmente incorpora diferentes tipos de análise que vimos em outras partes deste livro, da detecção de anomalias à análise de coorte. A criação de perfis de dados pode ser útil para a detecção dos problemas que ocorreram. Quando não for possível a realização de experimentos randomizados, outras técnicas estarão disponíveis, e o SQL pode ser usado na criação dos grupos de controle sintético e de variantes.

No próximo capítulo, examinaremos a construção de conjuntos de dados complexos para uso na análise, uma área que reúne muitos dos tópicos que discutimos no livro até agora.

---

1 Consulte *https://www.mathsisfun.com/data/chi-square-test.html* para ver uma boa explicação desse teste.

CAPÍTULO 8

Criando conjuntos de dados complexos para análise

Nos capítulos 3 a 7, examinamos várias maneiras de usar SQL para a análise de dados de bancos de dados. Além desses casos de uso específicos, o objetivo de uma consulta poderia ser a montagem de um conjunto de dados que fosse específico, porém de uso suficientemente geral para ser empregado na execução de análises posteriores. O destino poderia ser uma tabela de banco de dados, um arquivo de texto ou uma ferramenta de inteligência empresarial. O SQL necessário pode ser simples, demandando apenas alguns filtros ou agregações. No entanto, geralmente o código ou a lógica para a obtenção do conjunto de dados desejado pode ser muito complexo. É provável que esse código também seja atualizado com o passar do tempo, quando os stakeholders solicitarem pontos de dados ou cálculos adicionais. A organização, o desempenho e a capacidade de manutenção de seu código SQL serão críticos em um nível não requerido por análises de execução única.

Neste capítulo, discutirei os princípios de organização do código para torná-lo mais fácil de compartilhar e atualizar. Em seguida, abordarei quando devemos manter a lógica da consulta em SQL e quando é mais interessante movê-la para tabelas permanentes por meio de código ETL (extract-transform-load). Explicarei então as opções para o armazenamento de resultados intermediários – subconsultas, tabelas temporárias e CTEs (common table expressions, expressões de tabela comuns) – e apresentarei considerações para o seu uso no código. Por fim, concluirei com uma investigação das técnicas para a redução do tamanho do conjunto de dados e ideias para a manipulação da privacidade dos dados e a remoção de PII (personally identifiable information, informações de identificação pessoal).

Quando usar SQL para conjuntos de dados complexos

Quase todos os conjuntos de dados preparados para análise posterior contêm alguma lógica. A lógica pode variar de relativamente simples – como na maneira de as tabelas serem unidas com *JOIN* e de os filtros serem inseridos na cláusula *WHERE* – a cálculos complexos que agregam, categorizam, fazem o parsing ou executam funções de janela com partições dos dados. Na criação de conjuntos de dados para análise posterior, selecionar se a lógica deve ser mantida dentro da consulta SQL ou se será deixada para uma etapa inicial em um job ETL ou para uma etapa posterior em outra ferramenta geralmente exige arte e ciência. Conveniência, desempenho e disponibilidade de ajuda de engenheiros devem ser levados em consideração na decisão. Normalmente não há uma resposta definitiva, mas você desenvolverá sua intuição e confiança à medida que trabalhar mais com SQL.

As vantagens de usar SQL

O SQL é uma linguagem muito flexível. Espero tê-lo convencido nos capítulos anteriores de que uma grande variedade de tarefas de preparação e análise de dados pode ser executada com o uso de SQL. Essa flexibilidade é a principal vantagem do uso de SQL no desenvolvimento de conjuntos de dados complexos.

Nos estágios iniciais do trabalho com um conjunto de dados podem ser feitas muitas consultas. Geralmente o trabalho começa com consultas relacionadas a perfis para o conhecimento dos dados. Essa etapa é seguida da construção passo a passo da consulta, com a verificação de transformações e agregações ao longo do percurso para confirmar se os resultados retornados estão corretos. Esse processo pode ser intercalado por mais tarefas relacionadas a perfis, quando os valores reais diferirem das expectativas. Conjuntos de dados complexos podem ser construídos pela combinação de várias subconsultas que respondam a perguntas específicas com JOINs ou UNIONs. A execução da consulta e a verificação da saída são rápidas e permitem uma iteração veloz.

O SQL tem poucas dependências, precisa apenas que os dados das tabelas tenham qualidade e estejam atualizados. As consultas são feitas sob demanda e não dependem de um engenheiro de dados ou de um processo de liberação. Geralmente elas podem ser embutidas em ferramentas de BI

(business intelligence, inteligência empresarial) ou em código R ou Python pelo analista ou pelos cientistas de dados, sem solicitação de suporte técnico. Se um stakeholder precisar que outro atributo ou agregação seja adicionado à saída, as alterações podem ser feitas rapidamente.

Manter a lógica no código SQL é o ideal no trabalho em uma nova análise e se você estiver esperando que a lógica e o conjunto de resultados passem por alterações frequentes. Além disso, se a consulta for veloz e os dados retornarem para os stakeholders rapidamente, talvez não seja preciso mover a lógica para outro local.

Quando a lógica deve ser construída em ETL

Há situações nas quais mover a lógica para um processo ETL é uma opção melhor do que a manter totalmente em uma consulta SQL, principalmente se o trabalho for feito em uma organização que tenha um data warehouse ou data lake. As duas principais razões para o uso do ETL são o desempenho e a visibilidade.

O desempenho das consultas SQL depende da complexidade da lógica, do tamanho das tabelas consultadas e dos recursos computacionais do banco de dados subjacente. Embora muitas consultas sejam executadas com rapidez, principalmente em bancos de dados e hardware mais novos, inevitavelmente você acabará escrevendo algumas consultas que terão cálculos complexos, envolverão *JOIN*s de tabelas grandes ou *JOIN*s Cartesianas, ou farão o tempo de execução aumentar para minutos ou um período maior. Um analista ou um cientista de dados pode não se importar em esperar pelo retorno da consulta. No entanto, a maioria dos consumidores de dados está acostumada a tempos de resposta rápidos nos sites e ficará frustrada se tiver de esperar mais do que alguns segundos pelos dados.

O ETL é executado em segundo plano em horas agendadas e grava o resultado em uma tabela. Já que fica em segundo plano, pode ser executado por 30 segundos, cinco minutos ou uma hora e os usuários finais não são afetados. Geralmente os agendamentos são diários, mas podem ser configurados com intervalos mais curtos. Os usuários finais podem consultar a tabela resultante diretamente, sem a necessidade de *JOIN*s ou outra lógica e, portanto, vivenciam tempos de consulta rápidos.

Um bom exemplo de quando o ETL costuma ser uma opção melhor do que manter toda a lógica em uma consulta SQL é a tabela de snapshots diários. Em muitas organizações, a manutenção de um snapshot diário dos clientes, pedidos ou outras entidades é útil na resposta a perguntas analíticas. No caso dos clientes, poderíamos querer calcular o total de pedidos ou visitas até a data atual, o status atual em um pipeline de vendas, e outros atributos que possam mudar ou aumentar. Vimos como criar séries diárias, inclusive para dias em que uma entidade não estava presente, nas discussões de análise de séries temporais no Capítulo 3 e de análise de coorte no Capítulo 4. No nível de entidade individual, e durante longos períodos de tempo, essas consultas podem tornar-se lentas. Além disso, atributos como o status atual podem ser sobrepostos na tabela de origem, logo capturar um snapshot diário talvez seja a única maneira de preservar um retrato preciso da história. Geralmente o desenvolvimento do ETL e o armazenamento de resultados em snapshots diários são esforços válidos.

A visibilidade é a segunda razão para a transferência da lógica para o ETL. Normalmente as consultas SQL ficam no computador de uma pessoa ou permanecem ocultas dentro de código de relatório. Se já pode ser difícil para outras pessoas até mesmo encontrarem a lógica embutida na consulta, imagine entendê-la e procurar erros. Mover a lógica para o ETL e armazenar o código ETL em um repositório como o GitHub torna mais fácil para outras pessoas da organização o encontrarem, verificarem e usarem novamente. A maioria dos repositórios usados por equipes de desenvolvimento também armazena um histórico de alterações, um benefício adicional que nos permite ver quando uma linha específica de uma consulta foi adicionada ou alterada.

Existem boas razões para considerarmos inserir a lógica no ETL, mas essa abordagem também tem suas desvantagens. Uma delas é que resultados recentes não estarão disponíveis até o job ETL ser executado e atualizar os dados, mesmo se novos dados chegarem na tabela subjacente. Isso pode ser resolvido se continuarmos a executar o SQL nos dados brutos em busca de registros novos, porém limitando-o a uma janela de tempo pequena para que a consulta seja executada rapidamente. Opcionalmente essa abordagem pode ser combinada com uma consulta na tabela ETL,

com o uso de subconsultas ou *UNION*. Outra desvantagem da inserção de lógica no ETL é que fica mais difícil fazer alterações. Geralmente atualizações ou correções de bugs precisam ser passadas para um engenheiro de dados e o código tem de ser testado, inserido no repositório e lançado no data warehouse de produção. É por isso que prefiro esperar até que minhas consultas tenham passado pelo período de iteração rápida, e os conjuntos de dados resultantes tenham sido revisados e estejam sendo usados pela organização, para movê-las para o ETL. É claro que tornar o código mais difícil de alterar e impor revisões são ótimas maneiras de assegurar consistência e qualidade dos dados.

Views como alternativa ao ETL

Se você quiser ter a capacidade de reutilização e a visibilidade de código do ETL, porém sem armazenar os resultados e, portanto, ocupar espaço permanentemente, uma *view* de banco de dados pode ser uma boa opção. Uma view é basicamente uma consulta salva com um alias permanente que pode ser referenciada como qualquer tabela do banco de dados. A consulta pode ser simples ou complexa e envolver junções de tabela, filtros e outros elementos do SQL.

As views podem ser usadas para assegurar que todas as pessoas que consultarem os dados usem as mesmas definições, como estabelecido na consulta subjacente – por exemplo, excluindo sempre por filtragem as transações de teste. Elas podem ser empregadas para proteger os usuários da complexidade da lógica subjacente, o que é útil para usuários mais novos ou ocasionais de um banco de dados. As views também podem ser usadas para fornecer uma camada adicional de segurança ao restringir o acesso a certas linhas ou colunas do banco de dados. Por exemplo, uma view poderia ser criada para excluir PII, como os endereços de email, mas permitir que os criadores visualizem outros atributos aceitáveis dos clientes.

Contudo, as views têm algumas desvantagens. Elas são objetos do banco de dados e, portanto, requerem permissão para a criação e atualização de suas definições. As views não armazenam os dados; logo, sempre que uma consulta é executada com uma view, o banco de dados precisa voltar à tabela ou às tabelas subjacentes para buscá-los. Como resultado, elas não substituem o ETL, que cria uma tabela de dados pré-computada.

A maioria dos principais bancos de dados também tem *views materializadas (materialized views)*, que são semelhantes às views, mas armazenam os dados retornados em uma tabela. Geralmente o planejamento da criação e da atualização de views materializadas pode ser feito de maneira mais apropriada

com uma consulta a um administrador de banco de dados experiente, já que há várias considerações de desempenho que não fazem parte do escopo deste livro.

Quando inserir a lógica em outras ferramentas

O código SQL e a saída com os resultados no editor de consultas costumam ser apenas parte de uma análise. Normalmente os resultados são embutidos em relatórios, visualizados em tabelas e gráficos ou manipulados posteriormente em várias ferramentas, que podem variar de planilhas e softwares de BI a ambientes nos quais um código estatístico ou de machine learning é aplicado. Além da decisão de quando seria interessante mover a lógica em uma etapa inicial para o ETL, também temos opções para quando mover a lógica em uma etapa posterior para outras ferramentas. Tanto o desempenho quanto casos de uso específicos são fatores essenciais na decisão.

Cada tipo de ferramenta tem vantagens e limitações quando se trata de desempenho. As planilhas são muito flexíveis, mas não conseguem manipular grandes quantidades de linhas ou cálculos complexos de muitas linhas. Os bancos de dados definitivamente fornecem uma vantagem para o desempenho, logo geralmente é melhor executar o maior número possível de cálculos no banco de dados e passar o menor conjunto de dados possível para a planilha.

As ferramentas de BI têm muitos recursos, logo é importante saber tanto como o software manipula cálculos quanto como os dados serão usados. Algumas ferramentas de BI podem armazenar dados em *cache* (manter uma cópia local) em um formato otimizado, acelerando os cálculos. Outras emitem uma nova consulta sempre que um campo é adicionado ou removido em um relatório e, portanto, usam principalmente o poder computacional do banco de dados. Certos cálculos como `count distinct` e `median` requerem dados detalhados, no nível de entidade. Se não for possível antecipar todas as variações desses cálculos, pode ser necessário passar um conjunto de dados maior e mais detalhado do que seria o ideal. Além disso, se o objetivo for criar um conjunto de dados que permita a exploração e o fatiamento de muitas maneiras diferentes, quanto mais detalhes, melhor. Descobrir a combinação mais apropriada de computação com SQL, ETL e ferramenta de BI pode demandar alguma

iteração.

Quando o objetivo for executar análise estatística ou de machine learning no conjunto de dados com o uso de uma linguagem como R ou Python, dados detalhados podem ajudar. Essas duas linguagens podem executar tarefas semelhantes às do SQL, como a agregação e a manipulação de texto. Geralmente é melhor executar o maior número de cálculos possível em SQL, para aproveitar o poder computacional do banco de dados, mas não mais do que isso. Flexibilidade para iterar costuma ser uma parte importante do processo de modelagem. A decisão de se os cálculos serão executados em SQL ou outra linguagem também pode depender do seu nível de familiaridade e conforto com cada tipo de abordagem. Quem se sentir mais confortável com SQL talvez prefira fazer a maior parte dos cálculos no banco de dados, enquanto quem conhecer melhor R ou Python pode achar mais conveniente fazer cálculos nessas linguagens.

Embora existam poucas regras para a decisão de onde inserir a lógica, recomendo que você siga uma regra específica: evitar etapas manuais. É muito fácil abrir um conjunto de dados em uma planilha ou editor de texto, fazer uma pequena alteração, salvar e seguir adiante. No entanto, quando for preciso iterar, ou quando novos dados chegarem, você pode se esquecer dessa etapa manual ou pode executá-la inconsistentemente. Pelo que vivenciei até hoje, não há algo como uma solicitação "que não se repita". Se possível, insira a lógica em algum local do código.

O SQL é uma ótima ferramenta e é incrivelmente flexível. Ele também pode ser encontrado dentro do fluxo de trabalho de análise e em todo um ecossistema de ferramentas. Decidir onde inserir cálculos pode demandar uma abordagem de tentativa e erro enquanto você avalia o que é viável fazer com SQL, ETL ou ferramentas downstream. Quanto maior for a familiaridade e a experiência que você tiver com todas as opções disponíveis, melhor poderá estimar os prós e contras e mais apto estará para continuar a melhorar o desempenho e a flexibilidade de seu trabalho.

Organização do código

O SQL tem poucas regras de formatação, o que pode levar a consultas

confusas. As cláusulas da consulta devem estar na ordem correta: *SELECT* é seguida de *FROM*, e *GROUP BY* não pode preceder *WHERE,* por exemplo. Algumas palavras-chave como *SELECT* e *FROM* são *reservadas* (isto é, não podem ser usadas como nomes de campos, nomes de tabelas ou aliases). No entanto, ao contrário do que ocorre em algumas linguagens, novas linhas, espaços em branco (que não sejam os espaços que separam as palavras) e capitalização não são importantes, sendo ignorados pelo banco de dados. Qualquer um dos exemplos de consulta deste livro poderia ter sido escrito em uma única linha, e com ou sem letras maiúsculas, exceto em strings com aspas. Como resultado, grande parte da tarefa de organização do código é de responsabilidade da pessoa que está escrevendo a consulta. Felizmente, temos algumas ferramentas formais e informais para manter o código organizado, que vão dos comentários à formatação "cosmética", como a indentação, e às opções de armazenamento para arquivos de código SQL.

Comentários

A maioria das linguagens de codificação tem uma maneira de indicar que um bloco de texto deve ser tratado como comentário e ignorado durante a execução. O SQL tem duas opções. A primeira é usar dois símbolos de traço, o que transforma tudo o que estiver depois deles na linha em um comentário:

```
-- Este é um comentário
```

A segunda opção é usar os caracteres de barra (/) e asterisco (*) para iniciar um bloco de comentário, que pode se estender por várias linhas, seguidas de um asterisco e uma barra para terminar o bloco:

```
/*
Este é um bloco de comentário
com várias linhas
*/
```

Muitos editores SQL ajustam o estilo visual do código dentro dos comentários, esmaecendo-os com uma cor cinza ou alterando a cor para facilitar sua identificação.

Comentar o código é uma boa prática, porém é reconhecidamente uma

prática que muitas pessoas têm de se esforçar para usar regularmente. O código SQL costuma ser escrito com rapidez, e principalmente durante exercícios de exploração e criação de perfil, não é esperado que ele seja mantido por muito tempo. Códigos com excesso de comentários podem ser tão difíceis de ler quanto códigos sem comentários. E todos nós penamos com a ideia de que, já que escrevemos o código, teremos sempre de nos lembrar de *por que* o escrevemos. No entanto, qualquer pessoa que tiver herdado uma longa consulta escrita por um colega ou que não tenha lidado com uma consulta durante alguns meses e depois precisou voltar a ela para atualizá-la sabe que pode ser frustrante e demorado decifrar o código.

Para encontrar um equilíbrio entre o incômodo e o benefício de comentar, tente seguir algumas regras gerais. Em primeiro lugar, adicione um comentário em qualquer local em que um valor tiver um significado que não for óbvio. Muitos sistemas de origem codificam valores como inteiros, e é fácil esquecer seu significado. Deixar um comentário esclarece o significado e facilita a alteração do código se necessário:

```
WHERE status in (1,2) -- 1 é Ativo, 2 é Pendente
```

Em segundo lugar, comente qualquer cálculo ou transformação que não for óbvio. Pode ser qualquer coisa que alguém que não tiver criado um perfil para o conjunto de dados talvez não saiba, desde erros na entrada de dados à existência de valores discrepantes:

```
case when status = 'Live' then 'Active'
     else status end
     /* costumávamos usar o termos Live para os clientes mas em
     2020 mudamos para Active */
```

A terceira prática que tento seguir no que diz respeito aos comentários é deixar notas quando a consulta contém várias subconsultas. Uma linha rápida sobre o que cada subconsulta calcula facilitará pular para a parte relevante quando você voltar posteriormente para fazer uma verificação de qualidade ou editar uma consulta mais longa:

```
SELECT...
FROM
( -- encontra a primeira data para cada cliente
    SELECT ...
```

```
    FROM ...
) a
JOIN
( -- encontra todos os produtos para cada cliente
    SELECT ...
    FROM ...
) b on a.field = b.field
...
;
```

Fazer comentários adequados exige prática e alguma disciplina, mas é recomendável usá-los para a maioria das consultas que ultrapasse algumas linhas. Os comentários também podem ser usados para a inclusão de informações úteis referentes à consulta em geral, como a finalidade, o autor, a data de criação e assim por diante. Seja gentil com seus colegas, e facilite a sua vida no futuro, inserindo comentários úteis em seu código.

Capitalização, indentação, parênteses e outros truques de formatação

A formatação, e principalmente uma formatação consistente, é uma boa maneira de manter o código SQL organizado e legível. Os bancos de dados ignoram a capitalização e os espaços em branco (espaços, tabulações e novas linhas) em SQL, logo podemos usá-los a nosso favor para formatar o código em blocos mais legíveis. Os parênteses podem controlar a ordem de execução, que discutiremos com mais detalhes posteriormente, e também agrupar visualmente elementos de cálculo.

Palavras capitalizadas se destacam do restante, como qualquer pessoa que já recebeu um email com a linha de assunto toda capitalizada pode confirmar. Gosto de usar a capitalização apenas para as cláusulas principais: *SELECT*, *FROM*, *JOIN*, *WHERE* e assim por diante. Principalmente em consultas longas ou complexas, poder detectá-las rapidamente e, portanto, saber onde a cláusula *SELECT* termina e a cláusula *FROM* começa economiza muito tempo.

O espaço em branco é outra maneira importante de organizar, tornar partes da consulta mais fáceis de encontrar e entender que partes estão ligadas logicamente. Qualquer consulta SQL pode ser escrita em uma

única linha no editor, mas quase sempre isso leva a muita rolagem do código para a esquerda e para a direita. Gosto de começar cada cláusula (*SELECT*, *FROM* etc.) em uma nova linha, o que, junto com a capitalização, ajuda a saber onde cada uma começa e termina. Além disso, acho que inserir as agregações em suas próprias linhas, assim como funções que ocupem algum espaço, ajuda na organização. No caso de instruções CASE com mais do que duas condições WHEN, separá-las em várias linhas também é uma boa maneira de ver e acompanhar facilmente o que está acontecendo no código. Como exemplo, podemos consultar `type` e `mag` (magnitude), fazer o parsing de `place` e contar os registros da tabela `earthquakes`, com alguma filtragem na cláusula *WHERE*:

```
SELECT type, mag
,case when place like '%CA%' then 'California'
      when place like '%AK%' then 'Alaska'
      else trim(split_part(place,',',2))
      end as place
,count(*)
FROM earthquakes
WHERE date_part('year',time) >= 2019
and mag between 0 and 1
GROUP BY 1,2,3
;
type                mag  place       count
------------------  ---  ----------  -----
chemical explosion  0    California  1
earthquake          0    Alaska      160
earthquake          0    Argentina   1
..                 ...  ...         ...
```

A indentação é outro truque para manter o código visualmente organizado. Adicionar espaços ou tabulações para o alinhamento dos itens WHEN dentro de uma instrução CASE é um exemplo. Você também viu subconsultas indentadas em exemplos no decorrer do livro. Esse recurso faz as subconsultas se destacarem visualmente, e quando uma consulta tem vários níveis de subconsultas aninhadas, facilita ver e entender a ordem na qual elas serão avaliadas e que subconsultas são pares em termos de nível:

```
SELECT...
FROM
(
    SELECT...
    FROM
    (
        SELECT...
        FROM...
    ) a
    JOIN
    (
        SELECT...
        FROM
    ) b on...
) a
...
;
```

Um número ilimitado de outras opções de formatação pode ser usado e a consulta retornará os mesmo resultados. Criadores de código SQL experientes tendem a ter suas próprias preferências de formatação. No entanto, uma formatação clara e consistente facilita muito a criação, a manutenção e o compartilhamento de código SQL.

Muitos editores de consulta SQL fornecem algum tipo de formatação e coloração de consultas. Geralmente as palavras-chave são coloridas, o que as torna mais fáceis de detectar dentro de uma consulta. Essas pistas visuais facilitam muito tanto o desenvolvimento quanto a revisão de consultas SQL. Se você tem escrito SQL em um editor de consultas, tente abrir um arquivo *.sql* em um editor de texto sem formatação para ver a diferença que a coloração faz. A Figura 8.1 mostra o exemplo de um editor de consultas SQL e o mesmo código é mostrado em um editor de texto sem formatação na Figura 8.2 (esses exemplos podem aparecer em escala de cinza em algumas versões deste livro).

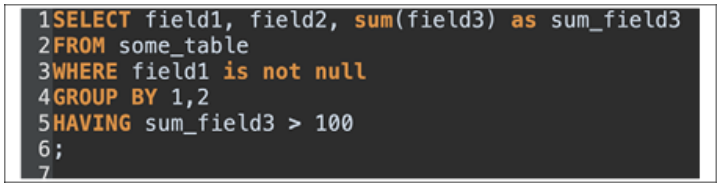


Figura 8.1: Screenshot da coloração de palavras-chave no editor de consultas SQL DBVisualizer.

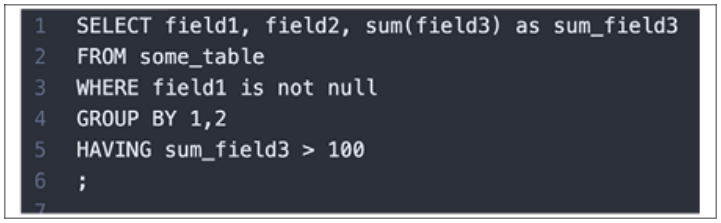


Figura 8.2: O mesmo código como texto sem formatação no editor de texto Atom.

A formatação é opcional do ponto de vista do banco de dados, mas é uma boa prática. O uso consistente de espaçamento, capitalização e outras opções de formatação ajudam muito a tornar o código legível, facilitando o compartilhamento e a manutenção.

Armazenando o código

Terminadas as difíceis tarefas de comentar e formatar o código, é uma boa ideia armazená-lo em algum local para o caso de você precisar usá-lo ou referenciá-lo posteriormente.

Muitos analistas e cientistas de dados trabalham com um editor SQL, geralmente um software para desktop. Os editores SQL são úteis porque costumam incluir ferramentas para a navegação no esquema do banco de dados junto a uma janela de código. Eles salvam arquivos com a extensão *.sql*, e esses arquivos de texto podem ser abertos e alterados em qualquer editor de texto. Os arquivos podem ser salvos em diretórios locais ou em serviços de armazenamento de arquivos baseados em nuvem.

Já que são textuais, os arquivos de código SQL são fáceis de armazenar em repositórios de controle de alterações como o GitHub. O uso de um repositório fornece uma boa opção de backup e facilita o compartilhamento com outras pessoas. Os repositórios também registram o histórico de alterações dos arquivos, o que será útil se você precisar saber quando uma alteração específica foi feita, ou quando o histórico de

alterações for necessário por razões regulatórias. A principal desvantagem do GitHub e outras ferramentas é que normalmente eles não são uma etapa obrigatória do fluxo de trabalho de análise. Você terá de se lembrar de atualizar seu código periodicamente, e, como ocorre com qualquer etapa manual, elas são fáceis de esquecer.

Organizando a computação

Dois problemas relacionados que enfrentamos quando criamos conjuntos de dados complexos são gerar uma lógica correta e obter um bom desempenho nas consultas. A lógica deve ser correta ou os resultados serão irrelevantes. O desempenho das consultas para fins de análise, ao contrário do que ocorre em sistemas transacionais, costuma chegar apenas a "razoavelmente satisfatório". Consultas que não retornam são problemáticas, mas a diferença entre esperar 30 segundos e aguardar um minuto por resultados pode não ser tão importante. Em SQL, com frequência há mais de uma maneira de escrever uma consulta que retorne os resultados corretos. Podemos usar isso a nosso favor tanto para assegurar que a lógica esteja correta quanto para ajustar o desempenho de consultas de execução demorada. Existem três maneiras principais de organizar o cálculo de resultados intermediários em SQL: a subconsulta, tabelas temporárias e CTEs (common table expressions, expressões de tabela comuns). Antes de as detalharmos, examinaremos a ordem de avaliação em SQL. Para concluir a seção, introduzirei os `grouping sets`, que podem substituir a necessidade de unir (com *UNION*) consultas em certos casos.

Entendendo a ordem de avaliação de cláusulas SQL

Os bancos de dados convertem o código SQL em um conjunto de operações que serão executadas em ordem para retornar os dados solicitados. Embora entender exatamente como isso ocorre não seja necessário para alguém se tornar bom em escrever SQL para análise, conhecer a ordem na qual o banco de dados executará suas operações é muito útil (e às vezes necessário para a depuração de resultados inesperados).

 Muitos bancos de dados modernos têm otimizadores de consultas sofisticados que consideram várias partes da consulta para fornecer o plano mais eficiente para a execução. Embora eles possam considerar partes da consulta em uma ordem diferente da discutida aqui e, portanto, talvez demandem menos otimização das consultas por parte dos humanos, não calcularão resultados intermediários em uma ordem diferente da discutida no livro.

A ordem geral de avaliação é mostrada na Tabela 8.1. Geralmente as consultas SQL incluem apenas um subconjunto das cláusulas possíveis, logo a avaliação real só considera as etapas relevantes para a consulta.

Tabela 8.1: Ordem de avaliação da consulta SQL

1	FROM incluindo *JOIN*s e suas cláusulas *ON*
2	WHERE
3	GROUP BY incluindo agregações
4	HAVING
5	Funções de janela
6	SELECT
7	DISTINCT
8	UNION
9	ORDER BY
10	LIMIT e OFFSET

Primeiro as tabelas da cláusula *FROM* são avaliadas, junto com qualquer cláusula *JOIN* que ela tiver. Se a cláusula *FROM* incluir alguma subconsulta, ela será avaliada antes da passagem para o restante das etapas. Em uma *JOIN*, a cláusula *ON* especifica como as tabelas devem ser associadas, o que também pode filtrar o conjunto de resultados.

 *FROM* é sempre a primeira a ser avaliada, com uma exceção: quando a consulta não tem uma cláusula *FROM*. Na maioria dos bancos de dados, é possível fazer consultas usando apenas uma cláusula *SELECT*, como visto em alguns dos exemplos deste livro. Uma consulta somente com *SELECT* pode retornar informações do sistema como a data e a versão do banco de dados. Ela também pode aplicar funções matemáticas, de data, de texto e de outros tipos a

constantes. Embora haja reconhecidamente pouco uso para essas consultas em análises finais, elas são úteis para o teste de funções ou para a rápida iteração por cálculos complicados.

Em seguida, a cláusula *WHERE* é avaliada para determinar que registros devem ser incluídos em cálculos posteriores. Observe que *WHERE* aparece em uma fase mais inicial da ordem de avaliação e, portanto, não pode incluir os resultados que ocorrerem em uma etapa posterior.

GROUP BY é calculada depois, incluindo as agregações relacionadas como `count`, `sum` e `avg`. Como era de se esperar, *GROUP BY* incluirá somente os valores que existirem nas tabelas de *FROM* após terem sido feitas a junção e a filtragem na cláusula *WHERE*.

HAVING é avaliada em seguida. Já que vem depois de *GROUP BY*, *HAVING* pode fazer a filtragem pelos valores agregados retornados por *GROUP BY*. A única outra maneira de filtrar pelos valores agregados é inserindo a consulta em uma subconsulta e aplicando os filtros na consulta principal. Por exemplo, poderíamos querer encontrar todos os estados que têm pelo menos mil mandatos na tabela `legislators_terms`; faremos a ordenação pelos termos em ordem descendente por precaução:

```
SELECT state
,count(*) as terms
FROM legislators_terms
GROUP BY 1
HAVING count(*) >= 1000
ORDER BY 2 desc
;
state  terms
-----  -----
NY     4159
PA     3252
OH     2239
...    ...
```

As funções de janela, se usadas, são avaliadas em seguida. O interessante é que, como a essa altura as agregações já foram calculadas, elas podem ser usadas na definição da função de janela. Por exemplo, no conjunto de dados `legislators` do Capítulo 4, poderíamos calcular tanto os mandatos

cumpridos por estado quanto a média de mandatos de todos os estados em uma única consulta:

```
SELECT state
,count(*) as terms
,avg(count(*)) over () as avg_terms
FROM legislators_terms
GROUP BY 1
;
state  terms  avg_terms
-----  -----  ---------
ND     170    746.830
NV     177    746.830
OH     2239   746.830
...    ...    ...
```

As agregações também podem ser usadas na cláusula *OVER*, como na consulta a seguir que classifica os estados em ordem descendente pelo número total de mandatos:

```
SELECT state
,count(*) as terms
,rank() over (order by count(*) desc)
FROM legislators_terms
GROUP BY 1
;
state  terms  rank
-----  -----  ----
NY     4159   1
PA     3252   2
OH     2239   3
...    ...    ...
```

Neste momento, a cláusula *SELECT* é finalmente avaliada. Isso é um pouco contraintuitivo, já que as agregações e as funções de janela são inseridas na seção de *SELECT* da consulta. No entanto, o banco de dados já se encarregou dos cálculos e os resultados estão disponíveis para manipulação posterior ou para exibição como se encontram. Por exemplo, uma agregação pode ser inserida dentro de uma instrução CASE e ter funções matemáticas, de data ou de texto aplicadas se o seu resultado tiver

um desses tipos de dados.

Os agregadores `sum`, `count` e `avg` retornam valores numéricos. Contudo, as funções `min` e `max` retornam o mesmo tipo de dado da entrada e usam a ordem inerente a esse tipo de dado. Por exemplo, no caso de datas, `min` e `max` retornam a primeira e a última datas do calendário, enquanto `min` e `max` em campos de texto usam a ordem alfabética para determinar o resultado.

Depois de *SELECT* temos *DISTINCT*, se presente na consulta. Isso significa que todas as linhas são calculadas e então ocorre a desduplicação.

UNION (ou *UNION ALL*) é executada em seguida. Até esse momento, cada consulta que compõe *UNION* está sendo avaliada independentemente. Esse estágio ocorre quando os conjuntos de resultados são reunidos em apenas um conjunto. Ou seja, as consultas podem fazer seus cálculos de maneiras muito diferentes ou a partir de conjuntos de dados distintos. No entanto, *UNION* precisa que haja o mesmo número de colunas e que essas colunas tenham tipos de dados compatíveis.

ORDER BY é quase a última etapa da avaliação. Isso significa que ela pode acessar qualquer um dos cálculos anteriores para classificar o conjunto de resultados. A única ressalva é que, se *DISTINCT* for usada, *ORDER BY* não poderá incluir nenhum campo que não seja retornado na cláusula *SELECT*. Caso contrário, seria possível ordenar um conjunto de resultados por um campo que não apareça na consulta.

LIMIT e *OFFSET* são as últimas a serem avaliadas na sequência de execução da consulta. Isso assegura que o subconjunto retornado tenha todos os resultados calculados como especificado pelas outras cláusulas existentes na consulta. Também significa que *LIMIT* tem uso limitado no controle do volume de trabalho executado pelo banco de dados antes de os resultados serem retornados. Talvez isso seja mais perceptível quando uma consulta contém um valor de deslocamento (*OFFSET*) maior. Para fazer o deslocamento por, digamos, três milhões de registros, o banco de dados também tem de calcular o conjunto de resultados inteiro, descobrir onde está o registro de número três milhões mais um, e retornar os registros especificados por *LIMIT*. Isso não significa que *LIMIT* não é útil. Verificar alguns resultados pode confirmar os cálculos sem sobrecarregar a rede ou a máquina local com dados. Usar *LIMIT* o mais cedo possível em

uma consulta, como em uma subconsulta, também pode reduzir drasticamente o trabalho requerido pelo banco de dados no desenvolvimento de uma consulta mais complexa.

Agora que sabemos a ordem na qual o banco de dados avalia as consultas e faz cálculos, examinaremos algumas opções para o controle dessas operações no contexto de uma consulta complexa maior: as subconsultas, as tabelas temporárias e as CTEs.

Subconsultas

Geralmente as *subconsultas* são a primeira maneira pela qual aprendemos a controlar a ordem de avaliação em SQL ou a fazer cálculos que não possam ser executados em uma única consulta principal. Elas são versáteis e podem ajudar a organizar consultas longas em blocos menores com finalidades distintas.

Uma subconsulta é inserida em parênteses, uma notação que deve ser familiar pelo seu uso em matemática, onde os parênteses também forçam a avaliação de alguma parte de uma equação antes do restante. Dentro dos parênteses fica uma consulta autônoma que é avaliada antes da consulta principal externa. Supondo que a subconsulta esteja na cláusula *FROM*, o conjunto de resultados poderá ser consultado pelo código principal, como ocorreria com qualquer outra tabela. Já vimos muitos exemplos com subconsultas neste livro.

Uma exceção à natureza autônoma da subconsulta seria um tipo especial chamado *subconsulta lateral*, que pode acessar resultados de itens anteriores na cláusula *FROM*. Uma vírgula e a palavra-chave *LATERAL* são usadas em vez de *JOIN*, e não há cláusula *ON*. Em vez disso, uma consulta anterior é usada dentro da subconsulta. Como exemplo, suponhamos que quiséssemos analisar a associação anterior a partidos para legisladores que estejam cumprindo mandato atualmente. Poderíamos encontrar o primeiro ano em que eles foram membros de um partido diferente e verificar o quanto isso é comum quando agrupado pelo partido atual. Na primeira subconsulta, encontraremos os legisladores com mandato atualmente. Na subconsulta lateral, que é a segunda, usaremos os resultados da primeira subconsulta para retornar o início de mandato mais distante (`term_start`) no qual o partido é diferente do atual:

```
SELECT date_part('year',c.first_term) as first_year
,a.party
,count(a.id_bioguide) as legislators
FROM
(
    SELECT distinct id_bioguide, party
    FROM legislators_terms
    WHERE term_end > '2020-06-01'
) a,
LATERAL
(
    SELECT b.id_bioguide
    ,min(term_start) as first_term
    FROM legislators_terms b
    WHERE b.id_bioguide = a.id_bioguide
    and b.party <> a.party
    GROUP BY 1
) c
GROUP BY 1,2
;
first_year  party        legislators
----------  ----------   -----------
1979.0      Republican   1
2011.0      Libertarian  1
2015.0      Democrat     1
```

Acabamos obtendo algo bem incomum. Só três legisladores atuais mudaram de partido, e nenhum partido teve mais pessoas se alternando. Há outras maneiras de retornar o mesmo resultado – por exemplo, alterando a consulta para uma *JOIN* e movendo os critérios da cláusula *WHERE* da segunda subconsulta para a cláusula *ON*:

```
SELECT date_part('year',c.first_term) as first_year
,a.party
,count(a.id_bioguide) as legislators
FROM
(
    SELECT distinct id_bioguide, party
    FROM legislators_terms
```

```
    WHERE term_end > '2020-06-01'
) a
JOIN
(
    SELECT id_bioguide, party
    ,min(term_start) as first_term
    FROM legislators_terms
    GROUP BY 1,2
) c on c.id_bioguide = a.id_bioguide and c.party <> a.party
GROUP BY 1,2
;
```

Se a segunda tabela for muito grande, filtrar por um valor retornado em uma subconsulta anterior pode acelerar a execução. Pelo que vi em meus anos de experiência, o uso de *LATERAL* é menos comum e, portanto, menos conhecido do que o uso de outras sintaxes; logo, é bom reservá-lo para casos de uso que não possam ser resolvidos eficientemente de outra maneira.

As subconsultas dão muita flexibilidade e controle sobre a ordem dos cálculos. No entanto, uma série de cálculos complexa no meio de uma consulta maior pode dificultar o entendimento e a manutenção. Também há casos em que o desempenho da subconsulta é muito lento ou a consulta não retorna resultados. Felizmente, o SQL tem algumas opções adicionais que podem ajudar nessas situações: as tabelas temporárias e as expressões de tabela comuns.

Tabelas temporárias

Uma *tabela temporária (temp)* é criada de maneira semelhante a qualquer outra tabela do banco de dados, mas com uma diferença importante: ela só persiste durante a sessão atual. As tabelas temporárias são úteis quando trabalhamos somente com uma pequena parte de uma tabela muito grande, já que é mais rápido consultar tabelas pequenas. Elas também são úteis quando queremos usar um resultado intermediário em várias consultas. Já que a tabela temporária é autônoma, pode ser consultada muitas vezes dentro da mesma sessão. Outra situação na qual elas são úteis é quando trabalhamos em certos bancos de dados, como o

Redshift ou o Vertica, que particionam os dados em nós. A inserção (com *INSERT*) de dados em uma tabela temporária pode alinhar o particionamento para outras tabelas que serão associadas (com *JOIN*) em uma consulta subsequente. Há duas desvantagens principais nas tabelas temporárias. Em primeiro lugar, elas requerem privilégios no banco de dados para a gravação dos dados, o que pode não ser permitido por razões de segurança. Em segundo lugar, algumas ferramentas de BI, como o Tableau e o Metabase, só permitem que uma única instrução SQL crie um conjunto de dados,[1] enquanto uma tabela temporária requer pelo menos duas: a instrução para criar (*CREATE*) e para inserir (*INSERT*) dados na tabela temporária e na consulta que a está usando.

Para criar uma tabela temporária, use o comando *CREATE*, seguido da palavra-chave `TEMPORARY` e do nome que você deseja dar à tabela. Ela poderá então ser definida, e uma segunda instrução poderá ser usada para preenchê-la, ou você pode usar *CREATE as SELECT* para criar e preencher a tabela na mesma etapa. Por exemplo, você poderia criar uma tabela temporária com os diferentes estados para os quais há legisladores:

```
CREATE temporary table temp_states
(
    state varchar primary key
)
;
INSERT into temp_states
SELECT distinct state
FROM legislators_terms
;
```

A primeira instrução cria a tabela, enquanto a segunda a preenche com valores de uma consulta. Observe que, ao definir a tabela antes, preciso especificar o tipo de dado para todas as colunas (nesse caso, `varchar` para a coluna `state`) e opcionalmente posso usar outros elementos de definição da tabela, como estabelecer uma chave primária. Gosto de prefixar os nomes com “temp_” ou “tmp_” para me lembrar do fato de que estou usando uma tabela temporária na consulta principal, mas isso não é obrigatório.

A maneira mais rápida e fácil de gerar uma tabela temporária é com o

método *CREATE as SELECT*:

```
CREATE temporary table temp_states
as
SELECT distinct state
FROM legislators_terms
;
```

Nesse caso, o banco de dados está decidindo o tipo de dado automaticamente de acordo com os dados retornados pela instrução *SELECT* e nenhuma chave primária foi definida. A menos que você precise de um controle mais detalhado por razões de desempenho, esse segundo método servirá.

Já que as tabelas temporárias são gravadas em disco, se você precisar preenchê-las novamente durante uma sessão, terá de remover (com *DROP*) e recriar a tabela ou truncar (com *TRUNCATE*) os dados. Desconectar-se e reconectar-se no banco de dados também funciona.

Expressões de tabela comuns

As CTEs são relativamente novas na linguagem SQL, tendo sido introduzidas em muitos dos principais bancos de dados somente no início dos anos 2000. Criei códigos SQL durante anos sem elas, usando subconsultas e tabelas temporárias. No entanto, preciso dizer que, desde que as conheci anos atrás, fui achando-as cada vez mais interessantes.

Podemos considerar uma *expressão de tabela comum* como uma subconsulta inserida no começo da execução da consulta. Ela cria um conjunto de resultados temporário que pode então ser usado em qualquer local da consulta subsequente. Uma consulta pode ter várias CTEs e estas podem usar resultados de CTEs anteriores para executar cálculos adicionais.

As CTEs são úteis principalmente quando o resultado é usado várias vezes no restante da consulta. A alternativa, definir a mesma subconsulta repetidamente, é lenta (já que o banco de dados precisa executar a mesma consulta diversas vezes) e propensa a erros. Se nos esquecermos de atualizar a lógica em cada subconsulta idêntica, introduziremos erros no resultado final. Como as CTEs fazem parte de uma única consulta, elas não requerem nenhuma permissão especial do banco de dados. Elas

também podem ser uma maneira útil de organizar o código em blocos distintos e evitar o espalhamento de subconsultas aninhadas.

A principal desvantagem das CTEs vem do fato de elas serem definidas no início, ficando longe de onde são usadas. Isso pode tornar uma consulta mais difícil de ser decifrada por outras pessoas se ela for muito longa, já que seria preciso rolar para o começo para verificar a definição e depois voltar para onde a CTE é usada para entender o que está ocorrendo. O bom uso de comentários pode ajudar a resolver isso. Um segundo desafio é que as CTEs dificultam a execução de seções de consultas longas. Para verificarmos os resultados intermediários de uma consulta mais longa, seria mais fácil selecionar e executar uma subconsulta em uma ferramenta de desenvolvimento de consultas. No entanto, se uma CTE estiver envolvida, primeiro todo o código ao redor deve ser desativado por comentários.

Para criar uma CTE, usamos a palavra-chave *WITH* no início da consulta, seguida de um nome para a CTE e da consulta que a compõe inserida em parênteses. Por exemplo, poderíamos criar uma CTE que calculasse o primeiro mandato de cada legislador e usasse esse resultado em um cálculo posterior, como no cálculo de coorte introduzido no Capítulo 4:

```
WITH first_term as
(
    SELECT id_bioguide
    ,min(term_start) as first_term
    FROM legislators_terms
    GROUP BY 1
)
SELECT date_part('year',age(b.term_start,a.first_term)) as periods
,count(distinct a.id_bioguide) as cohort_retained
FROM first_term a
JOIN legislators_terms b on a.id_bioguide = b.id_bioguide
GROUP BY 1
;
periods  cohort_retained
-------  ---------------
0.0      12518
1.0      3600
```

```
2.0      3619
...      ...
```

O resultado da consulta é exatamente o mesmo retornado pela consulta alternativa que usa subconsultas vista no Capítulo 4. Várias CTE podem ser usadas na mesma consulta, separadas por vírgulas:

```
WITH first_cte as
(
    SELECT...
),
second_cte as
(
    SELECT...
)
SELECT...
;
```

As CTEs são uma maneira útil de controlar a ordem de avaliação, melhorar o desempenho em alguns casos e organizar o código SQL. Elas serão fáceis de usar se você estiver familiarizado com a sintaxe e estão disponíveis na maioria dos principais bancos de dados. Geralmente há várias maneiras de se fazer algo em SQL, e, embora não sejam obrigatórias, as CTEs adicionarão uma flexibilidade útil ao seu conjunto de habilidades na linguagem.

grouping sets

Embora esse tópico não seja estritamente sobre controlar a ordem de avaliação, é uma maneira útil de evitar *UNION*s e fazer o banco de dados executar todo o trabalho em uma única instrução de consulta. Dentro da cláusula *GROUP BY*, há uma sintaxe especial disponível na maioria dos principais bancos de dados que inclui `grouping sets`, `cube` e `rollup` (o Redshift é uma exceção e o MySQL só tem `rollup`). Essas sintaxes ajudam quando o conjunto de dados precisa conter subtotais para várias combinações de atributos.

Para os exemplos desta seção, usaremos um conjunto de dados de vendas de videogames que está disponível no Kaggle (*https://oreil.ly/qIxRX*). Ele contém atributos para o nome de cada jogo assim como para a

plataforma, o ano, o gênero e o publicador do game. Os valores de vendas fornecidos são da América do Norte, União Europeia, Japão e Outros (o restante do mundo), e há o total global. O nome da tabela é `videogame_sales`. A Figura 8.3 mostra uma amostra da tabela.

*	rank	name	platform	year	genre	publisher	na_sales	eu_sales	jp_sales	other_sales	global_sales
1	1	Wii Sports	Wii	2006	Sports	Nintendo	41.49	29.02	3.77	8.46	82.74
2	2	Super Mario Bros.	NES	1985	Platform	Nintendo	29.08	3.58	6.81	0.77	40.24
3	3	Mario Kart Wii	Wii	2008	Racing	Nintendo	15.85	12.88	3.79	3.31	35.82
4	4	Wii Sports Resort	Wii	2009	Sports	Nintendo	15.75	11.01	3.28	2.96	33
5	5	Pokemon Red/Pokemon Blue	GB	1996	Role-Playing	Nintendo	11.27	8.89	10.22	1	31.37
6	6	Tetris	GB	1989	Puzzle	Nintendo	23.2	2.26	4.22	0.58	30.26
7	7	New Super Mario Bros.	DS	2006	Platform	Nintendo	11.38	9.23	6.5	2.9	30.01
8	8	Wii Play	Wii	2006	Misc	Nintendo	14.03	9.2	2.93	2.85	29.02
9	9	New Super Mario Bros. Wii	Wii	2009	Platform	Nintendo	14.59	7.06	4.7	2.26	28.62
10	10	Duck Hunt	NES	1984	Shooter	Nintendo	26.93	0.63	0.28	0.47	28.31
11	11	Nintendogs	DS	2005	Simulation	Nintendo	9.07	11	1.93	2.75	24.76
12	12	Mario Kart DS	DS	2005	Racing	Nintendo	9.81	7.57	4.13	1.92	23.42
13	13	Pokemon Gold/Pokemon Silver	GB	1999	Role-Playing	Nintendo	9	6.18	7.2	0.71	23.1
14	14	Wii Fit	Wii	2007	Sports	Nintendo	8.94	8.03	3.6	2.15	22.72
15	15	Wii Fit Plus	Wii	2009	Sports	Nintendo	9.09	8.59	2.53	1.79	22
16	16	Kinect Adventures!	X360	2010	Misc	Microsoft Game Studios	14.97	4.94	0.24	1.67	21.82
17	17	Grand Theft Auto V	PS3	2013	Action	Take-Two Interactive	7.01	9.27	0.97	4.14	21.4
18	18	Grand Theft Auto: San Andreas	PS2	2004	Action	Take-Two Interactive	9.43	0.4	0.41	10.57	20.81
19	19	Super Mario World	SNES	1990	Platform	Nintendo	12.78	3.75	3.54	0.55	20.61
20	20	Brain Age: Train Your Brain in Minutes a Day	DS	2005	Misc	Nintendo	4.75	9.26	4.16	2.05	20.22

Figura 8.3: Amostra da tabela videogame_sales.

No conjunto de dados de videogame, poderíamos querer, por exemplo, agregar `global_sales` por plataforma, gênero e publicador como agregações autônomas (em vez de usar apenas as combinações dos três campos que existem nos dados), mas exibir os resultados em um único conjunto de consultas. Isso pode ser feito pela união de três consultas. Observe que cada consulta deve conter placeholders para todos os três campos do agrupamento:

```
SELECT platform
,null as genre
,null as publisher
,sum(global_sales) as global_sales
FROM videogame_sales
GROUP BY 1,2,3
    UNION
SELECT null as platform
,genre
,null as publisher
,sum(global_sales) as global_sales
FROM videogame_sales
GROUP BY 1,2,3
    UNION
```

```
SELECT null as platform
,null as genre
,publisher
,sum(global_sales) as global_sales
FROM videogame_sales
GROUP BY 1,2,3
;
platform  genre      publisher        global_sales
--------  ------     ---------        ------------
2600      (null)     (null)           97.08
3DO       (null)     (null)           0.10
...       ...        ...              ...
(null)    Action     (null)           1751.18
(null)    Adventure  (null)           239.04
...       ...        ...              ...
(null)    (null)     10TACLE Studios  0.11
(null)    (null)     1C Company       0.10
...       ...        ...              ...
```

Isso poderia ser feito em uma consulta mais compacta com o uso de `grouping sets`. Dentro da cláusula *GROUP BY*, `grouping sets` é seguido da lista de agrupamentos a ser calculada. A consulta anterior pode ser substituída por:

```
SELECT platform, genre, publisher
,sum(global_sales) as global_sales
FROM videogame_sales
GROUP BY grouping sets (platform, genre, publisher)
;
platform  genre      publisher        global_sales
--------  ------     ---------        ------------
2600      (null)     (null)           97.08
3DO       (null)     (null)           0.10
...       ...        ...              ...
(null)    Action     (null)           1751.18
(null)    Adventure  (null)           239.04
...       ...        ...              ...
(null)    (null)     10TACLE Studios  0.11
(null)    (null)     1C Company       0.10
```

```
...       ...        ...              ...
```

Os itens dentro dos parênteses de `grouping sets` podem incluir tanto espaços em branco quanto listas de colunas separadas por vírgulas. Como exemplo, podemos calcular as vendas globais sem nenhum agrupamento, além dos agrupamentos por `platform`, `genre` e `publisher`, incluindo um item na lista que seja apenas um par de parênteses. Também podemos limpar a saída substituindo os itens nulos por "All", usando `coalesce`:

```
SELECT coalesce(platform,'All') as platform
,coalesce(genre,'All') as genre
,coalesce(publisher,'All') as publisher
,sum(global_sales) as na_sales
FROM videogame_sales
GROUP BY grouping sets ((), platform, genre, publisher)
ORDER BY 1,2,3
;
platform  genre      publisher        global_sales
--------  ------     ---------        ------------
All       All        All              8920.44
2600      All        All              97.08
3DO       All        All              0.10
...       ...        ...              ...
All       Action     All              1751.18
All       Adventure  All              239.04
...       ...        ...              ...
All       All        10TACLE Studios  0.11
All       All        1C Company       0.10
...       ...        ...              ...
```

Se quisermos calcular todas as combinações possíveis de plataforma, gênero e publicador, como os subtotais individuais que acabaram de ser calculados, mais todas as combinações de plataforma e gênero, plataforma e publicador, e gênero e publicador, podemos especificar todas essas combinações em `grouping sets`. Ou podemos usar a útil sintaxe `cube`, que manipulará tudo isso para nós:

```
SELECT coalesce(platform,'All') as platform
,coalesce(genre,'All') as genre
,coalesce(publisher,'All') as publisher
```

```
,sum(global_sales) as global_sales
FROM videogame_sales
GROUP BY cube (platform, genre, publisher)
ORDER BY 1,2,3
;
platform  genre      publisher        global_sales
--------  ------     ---------        ------------
All       All        All              8920.44
PS3       All        All              957.84
PS3       Action     All              307.88
PS3       Action     Atari            0.2
All       Action     All              1751.18
All       Action     Atari            26.65
All       All        Atari            157.22
...       ...        ...              ...
```

Uma terceira opção seria a função `rollup`, que retorna um conjunto de dados com combinações determinadas pela ordem dos campos nos parênteses em vez de todas as combinações possíveis. Logo, a consulta anterior com a cláusula a seguir:

```
GROUP BY rollup (platform, genre, publisher)
```

retorna agregações para as combinações de:

```
platform, genre, publisher
platform, genre
platform
```

No entanto, a consulta *não* retorna agregações para as combinações de:

```
platform, publisher
genre,publisher
genre
publisher
```

Embora seja possível criar a mesma saída usando *UNION*, as opções `grouping sets`, `cube` e `rollup` economizam muito espaço e tempo quando são necessárias agregações em vários níveis, porque elas resultam em menos linhas de código e menos verificações nas tabelas de banco de dados subjacentes. Uma vez criei uma consulta com centenas de linhas usando *UNION*s para gerar uma saída de modo que o elemento gráfico

dinâmico de um site tivesse todas as combinações de filtros pré-calculados possíveis. A verificação de qualidade foi uma tarefa enorme e a atualização foi ainda pior. Usar `grouping sets`, assim como CTEs, poderia ter ajudado a tornar o código mais compacto e a facilitar sua criação e manutenção.

Gerenciando o tamanho do conjunto de dados e considerações sobre privacidade

Após criar uma lógica apropriada no código SQL, organizar o código e torná-lo eficiente, geralmente temos de superar outro desafio: o tamanho do conjunto de resultados. Está cada vez mais barato armazenar dados, o que significa que as organizações estão armazenando conjuntos de dados cada vez maiores. O poder computacional também está sempre aumentando, nos permitindo organizar esses dados usando as maneiras sofisticadas que vimos em capítulos anteriores. No entanto, gargalos ainda ocorrem, em sistemas downstream como as ferramentas de BI ou na largura de banda disponível para a passagem de grandes conjuntos de dados entre sistemas. Além disso, a privacidade dos dados é uma grande preocupação que afeta como manipulamos dados sigilosos. Por essas razões, nesta seção discutirei algumas maneiras de limitar o tamanho dos conjuntos de dados, assim como apresentarei considerações sobre a privacidade dos dados.

Amostragem com %, mod

Uma maneira de reduzir o tamanho de um conjunto de resultados seria usando uma amostra dos dados de origem. Usar uma *amostragem* significa pegar apenas um subconjunto dos pontos de dados ou observações. Isso é apropriado quando o conjunto de dados é suficientemente grande e um subconjunto representa a população inteira. Com frequência podemos obter amostras do tráfego de um site e continuar retendo a maioria dos insights úteis, por exemplo. Há dois cuidados que devemos tomar ao usar amostragens. O primeiro está relacionado à obtenção de um tamanho de amostra que encontre o equilíbrio certo entre reduzir o tamanho do conjunto de dados e não perder muitos detalhes críticos. A amostra pode incluir 10%, 1% ou 0,1% dos pontos de dados, dependendo do volume

inicial. O segundo cuidado está relacionado à entidade da qual será tirada a amostragem. Poderíamos obter uma amostragem de 1% das *visitas* no site, mas, se o objetivo da análise for entender como os usuários navegam, uma amostragem de 1% dos *visitantes* do site seria uma opção melhor para preservar todos os pontos de dados dos usuários na amostra.

A maneira mais comum de obter amostras é filtrar os resultados da consulta na cláusula *WHERE* aplicando uma função a um identificador no nível de entidade. Muitos IDs são armazenados como inteiros. Se for esse o caso, calcular o módulo será uma maneira rápida de conseguir o resultado certo. O módulo é o resto na forma de valor inteiro quando um número é dividido por outro. Por exemplo, 10 dividido por 3 é igual a 3 com um resto (módulo) igual a 1. O SQL tem duas maneiras equivalentes para encontrar o módulo – com o sinal % e com a função `mod`:

```
SELECT 123456 % 100 as mod_100;
mod_100
-------
56
SELECT mod(123456,100) as mod_100;
mod_100
-------
56
```

Os dois retornam a mesma resposta, 56, que também são os dois últimos dígitos do valor da entrada `123456`. Para gerar uma amostra de 1% do conjunto de dados, insira uma das duas sintaxes na cláusula *WHERE* e configure-a como sendo igual a um inteiro – nesse caso, usamos 7:

```
SELECT user_id, ...
FROM table
WHERE user_id % 100 = 7
;
```

Um módulo de 100 cria uma amostra de 1%, enquanto um módulo de 1.000 criaria uma mostra de 0,1% e um módulo de 10 criaria uma amostra de 10%. Embora seja comum a obtenção de amostras em múltiplos de 10, isso não é obrigatório e qualquer inteiro funcionará.

Obter amostras originárias de identificadores alfanuméricos que incluam tanto letras quanto números não é tão fácil quanto obter amostras de

identificadores apenas numéricos. Funções de parsing de strings podem ser usadas para isolar apenas os primeiros ou os últimos caracteres, e filtros podem ser aplicados a eles. Por exemplo, podemos obter uma amostra somente de identificadores que terminem com a letra "b" fazendo o parsing do último caractere de uma string com a função `right`:

```
SELECT user_id, ...
FROM table
WHERE right(user_id,1) = 'b'
;
```

Supondo que qualquer letra maiúscula ou minúscula ou número seja um valor possível, isso resultará em uma amostra de aproximadamente 1,6% (1/62). Para retornar uma amostra maior, ajuste o filtro para permitir vários valores:

```
SELECT user_id, ...
FROM table
WHERE right(user_id,1) in ('b','f','m')
;
```

Para criar uma amostra menor, inclua vários caracteres:

```
SELECT user_id, ...
FROM table
WHERE right(user_id,2) = 'c3'
;
```

Na obtenção de amostras, é útil validar se a função usada cria realmente uma amostragem aleatória ou quase aleatória dos dados. Em um dos meus empregos anteriores, descobri que certos tipos de usuários tinham mais probabilidades de apresentar determinadas combinações dos dois últimos dígitos em seus IDs. Nesse caso, usar a função `mod` para gerar uma amostra de 1% resultou em um viés perceptível nos resultados. Com frequência, principalmente identificadores alfanuméricos apresentam padrões comuns no começo ou no fim da string que perfis de dados podem ajudar a identificar.

A amostragem é uma maneira fácil de reduzir o tamanho do conjunto de dados por ordens de magnitude. Pode tanto acelerar os cálculos dentro de instruções SQL quanto permitir que o resultado final seja mais compacto, tornando mais rápido e fácil transferi-lo para outra ferramenta ou sistema. No entanto, às vezes a perda de detalhes proveniente da

amostragem não é aceitável e outras técnicas são necessárias.

Reduzindo a dimensionalidade

O número de diferentes combinações de atributos, ou *dimensionalidade*, causa um grande impacto sobre a quantidade de registros de um conjunto de dados. Para entender isso, podemos fazer um experimento mental simples. Suponhamos que tivéssemos um campo com 10 valores distintos, contássemos o número de registros e fizéssemos o agrupamento por esse campo. A consulta retornará 10 resultados. Agora adicionaremos um segundo campo, também com 10 valores distintos, contaremos o número de registros e faremos o agrupamento pelos dois campos. A consulta retornará 100 resultados. Adicionaremos um terceiro campo com 10 valores distintos e a consulta crescerá para 1.000 resultados. Mesmo se não existirem todas as combinações dos três campos na tabela consultada, fica claro que a inclusão de campos adicionais em uma consulta pode aumentar drasticamente o tamanho dos resultados.

Na execução de uma análise, geralmente podemos controlar o número de campos e filtrar os valores incluídos para obter uma saída gerenciável. No entanto, na preparação de conjuntos de dados para análise posterior em outras ferramentas, o objetivo costuma ser dar flexibilidade e, portanto, incluir muitos atributos e cálculos diferentes. Para reter o maior número de detalhes possível gerenciando ao mesmo tempo o tamanho geral dos dados, podemos usar uma ou mais técnicas de agrupamento.

Normalmente a granularidade de datas e horas é um elemento óbvio que pode ajudar a reduzir o tamanho dos dados. Converse com seus stakeholders para determinar se dados diários são necessários, por exemplo, ou se agregações semanais ou mensais também serviriam. Agrupar os dados por mês e dia da semana pode ser uma solução para agregar dados e fornecer ao mesmo tempo visibilidade de padrões que difiram em dias úteis versus fins de semana. Restringir o período de tempo retornado é sempre uma opção, mas pode limitar a exploração de tendências de mais longo prazo. Vi equipes de dados fornecerem um único conjunto de dados com agregações no nível mensal e abrangendo vários anos, enquanto o conjunto de dados de uma empresa incluía os mesmos atributos, mas com dados diários ou até mesmo por hora para uma janela

de tempo muito mais curta.

Os campos de texto são outro local onde podemos procurar possíveis economias de espaço. Diferenças na grafia ou na capitalização podem resultar em um número maior de valores distintos do que o que seria útil. A aplicação das funções de texto discutidas no Capítulo 5, como `lower`, `trim` ou `initcap`, padronizaria os valores e também poderia tornar os dados mais úteis para os stakeholders. Instruções REPLACE ou CASE podem ser usadas para ajustes mais nuançados, como o ajuste da grafia ou a alteração de um nome que tenha sido atualizado com um novo valor.

Em certas situações, apenas alguns valores de uma lista mais longa podem ser relevantes para a análise, logo reter detalhes deles e agrupar o restante seria eficaz. Tenho visto isso com frequência no trabalho com localizações geográficas. Há aproximadamente duzentos países no mundo, mas geralmente só alguns têm clientes ou outros pontos de dados suficientes para tornar útil fazer relatórios individuais sobre eles. O conjunto de dados `legislators` usado no Capítulo 4 contém 59 valores para o estado, que incluem os 50 estados mais alguns territórios dos EUA que têm representantes. Poderíamos criar um conjunto de dados com detalhes para os cinco estados com as maiores populações (atualmente a Califórnia, o Texas, a Flórida, Nova York e a Pensilvânia) e agrupar o restante em uma categoria "other" com uma instrução CASE:

```
SELECT case when state in ('CA','TX','FL','NY','PA') then state
            else 'Other' end as state_group
,count(*) as terms
FROM legislators_terms
GROUP BY 1
ORDER BY 2 desc
;
state_group  count
-----------  -----
Other        31980
NY           4159
PA           3252
CA           2121
TX           1692
FL           859
```

A consulta retorna apenas 6 linhas, de um total de 59, o que representa uma redução significativa. Para tornar a lista mais dinâmica, antes podemos classificar os valores em uma subconsulta, nesse caso pelos valores distintos de `id_bioguide` (ID do legislador), e depois retornar o valor de `state` para os 5 estados principais e "Other" para o restante:

```
SELECT case when b.rank <= 5 then a.state
            else 'Other' end as state_group
,count(distinct id_bioguide) as legislators
FROM legislators_terms a
JOIN
(
    SELECT state
    ,count(distinct id_bioguide)
    ,rank() over (order by count(distinct id_bioguide) desc)
    FROM legislators_terms
    GROUP BY 1
) b on a.state = b.state
GROUP BY 1
ORDER BY 2 desc
;
state_group  legislators
-----------  -----------
Other        8317
NY           1494
PA           1075
OH           694
IL           509
VA           451
```

Vários estados mudaram nessa segunda lista. Se continuarmos atualizando o conjunto de dados com pontos de dados novos, a consulta dinâmica assegurará que a saída sempre reflita os valores principais atualizados.

A dimensionalidade também pode ser reduzida pela transformação dos dados em valores de flags. Geralmente as flags são binárias (isto é, só têm dois valores). Os valores BOOLEAN TRUE e FALSE podem ser usados para codificar flags, assim como o podem 1 e 0, "Yes" e "No" ou qualquer outro par de strings significativas. As flags são úteis quando um valor

limite é importante, mas maiores detalhes são menos interessantes. Por exemplo, poderíamos querer saber se um visitante de um site concluiu ou não uma compra, sem dar muita importância para detalhes sobre o número exato de compras.

No conjunto de dados `legislators`, há 28 mandatos distintos cumpridos pelos legisladores. No entanto, em vez do valor exato, poderíamos incluir em nossa saída apenas se um legislador cumpriu pelo menos dois mandatos, o que podemos fazer transformando os valores detalhados em uma flag:

```
SELECT case when terms >= 2 then true else false end as
  two_terms_flag
,count(*) as legislators
FROM
(
    SELECT id_bioguide
    ,count(term_id) as terms
    FROM legislators_terms
    GROUP BY 1
) a
GROUP BY 1
;
two_terms_flag  legislators
--------------  -----------
false           4139
true            8379
```

Aproximadamente o dobro dos legisladores cumpriu pelo menos dois mandatos em comparação com os que cumpriram apenas um. Quando combinado com outros campos de um conjunto de dados, esse tipo de informação pode gerar conjuntos de resultados bem menores.

Às vezes apenas um indicador verdadeiro/falso ou de presença/ausência não é suficiente para capturar a nuance necessária. Nesse caso, os dados numéricos podem ser convertidos em vários níveis para a manutenção de alguns detalhes adicionais. Isso é feito com uma instrução CASE, e o valor de retorno pode ser um número ou uma string.

Poderíamos incluir não só se um legislador cumpriu um segundo

mandato, mas também outro indicador para os que cumpriram 10 ou mais mandatos:

```
SELECT
case when terms >= 10 then '10+'
     when terms >= 2 then '2 - 9'
     else '1' end as terms_level
,count(*) as legislators
FROM
(
    SELECT id_bioguide
    ,count(term_id) as terms
    FROM legislators_terms
    GROUP BY 1
) a
GROUP BY 1
;
terms_level  legislators
-----------  -----------
1            4139
2 - 9        7496
10+          883
```

Aqui fizemos a redução de 28 valores distintos para 3, retendo a noção de legisladores de mandato único, dos que foram reeleitos e dos que se destacaram por permanecer muito tempo no cargo. Esses agrupamentos ou distinções ocorrem em muitas áreas. Como em todas as transformações que discutimos, pode ser necessário usar de tentativa e erro para a descoberta dos limites exatos que são mais significativos para os stakeholders. Encontrar o equilíbrio certo entre detalhe e agregação pode reduzir muito o tamanho do conjunto de dados e, portanto, geralmente melhora o tempo de distribuição e o desempenho da aplicação downstream.

PII e privacidade dos dados

A privacidade dos dados é um dos problemas mais importantes que os profissionais de dados enfrentam atualmente. Grandes conjuntos de dados com muitos atributos permitem análises mais robustas com

insights e recomendações detalhados. No entanto, quando o conjunto de dados é sobre pessoas, precisamos tomar cuidado com as dimensões ética e regulatória dos dados coletados e usados. Os regulamentos de privacidade de pacientes, estudantes e clientes de serviços financeiros já existem há muitos anos. Leis que regulam os direitos dos consumidores à privacidade dos dados também entraram em vigor nos últimos anos. O GDPR (General Data Protection Regulation, Regulamento Geral sobre a Proteção de Dados) promulgado pela União Europeia talvez seja a lei mais conhecida. Outros regulamentos incluem a CCPA (Consumer Privacy Act, Lei de Proteção da Privacidade do Consumidor da Califórnia), os Princípios de Privacidade Australianos, e a LGPD (General Data Protection Law, Lei Geral de Proteção de Dados) do Brasil.

Estes e outros regulamentos abrangem a manipulação, o armazenamento e (em alguns casos) a exclusão de *PII (personally identifiable information, informações de identificação pessoal)*. Algumas categorias de PII são óbvias: nome, endereço, email, data de nascimento e número do CPF. As PII também incluem indicadores de saúde como a frequência cardíaca, a pressão sanguínea, e diagnósticos médicos. Informações de localização, como coordenadas de GPS, são consideradas PII, já que um pequeno número de localizações de GPS pode identificar uma pessoa de maneira exclusiva. Por exemplo, leituras de GPS em minha casa e na escola de meus filhos poderiam identificar alguém de minha família de maneira exclusiva. Um terceiro ponto de GPS em meu escritório poderia me identificar de maneira exclusiva. Como praticante da análise de dados, recomendo conhecer o que esses regulamentos abordam e discutir como eles afetam seu trabalho com os advogados da organização especialistas em privacidade, que terão informações mais atualizadas.

Uma prática recomendada para uma análise de dados que inclua PII é evitar inserir as PII nas saídas. Isso pode ser feito pela agregação de dados, pela substituição de valores, ou com o uso de valores hash.

Na maioria das análises, o objetivo é encontrar tendências e padrões. Geralmente a finalidade é contar clientes e descobrir seu comportamento médio, em vez de incluir detalhes individuais na saída. As agregações costumam remover as PII; no entanto, lembre-se de que uma combinação de atributos que tenha uma contagem de usuários igual a 1 pode ser

associada a uma pessoa. Esses casos devem ser tratados como valores discrepantes e removidos do resultado para que seja mantido um grau mais alto de privacidade.

Se dados pessoais forem necessários por alguma razão – para tornar possível o cálculos dos usuários distintos de uma ferramenta downstream, por exemplo – podemos substituir valores problemáticos por valores alternativos aleatórios que mantenham a exclusividade. A função de janela `row_number` pode ser usada para atribuir um novo valor a cada indivíduo de uma tabela:

```
SELECT email
,row_number() over (order by ...)
FROM users
;
```

O desafio nesse caso é encontrar um campo para inserir em *ORDER BY* que torne a ordem suficientemente aleatória de modo a podermos considerar anonimizado o identificador de usuário resultante.

Os valores hash são outra opção. O hashing pega um valor de entrada e usa um algoritmo para criar um novo valor de saída. Um valor de entrada específico sempre resultará na mesma saída, o que torna essa uma boa opção para a manutenção da exclusividade obscurecendo ao mesmo tempo valores sigilosos. A função `md5` pode ser usada para gerar um valor hash:

```
SELECT md5('my info');
md5
--------------------------------
0fb1d3f29f5dd1b7cabbad56cf043d1a
```

A função md5 faz o hashing dos valores de entrada, mas não os criptografa e, portanto, pode ser revertida para a obtenção do valor original. No caso de dados altamente sigilosos, você deve trabalhar com um administrador de banco de dados para criptografar realmente os dados.

Evitar PII na saída das consultas SQL é sempre a melhor opção se possível, já que você impedirá que elas se proliferem para outros sistemas ou arquivos. Substituir ou mascarar os valores seria a segunda melhor opção. Você também pode explorar métodos seguros de

compartilhamento de dados, como desenvolver um pipeline de dados protegido diretamente entre um banco de dados e um sistema de email para evitar gravar endereços de email em arquivos, por exemplo. Com cuidado e parceria com colegas técnicos e da área jurídica, é possível obter uma análise de alta qualidade preservando ao mesmo tempo a privacidade das pessoas.

Conclusão

Em qualquer análise, várias decisões devem ser tomadas com relação à organização do código, ao gerenciamento da complexidade, à otimização do desempenho das consultas e à proteção da privacidade na saída. Neste capítulo, discutimos muitas opções e estratégias e a sintaxe SQL especial que pode ajudar nessas tarefas. Tente não se assustar com todas essas opções ou achar que, se não dominar esses tópicos, não conseguirá ser um analista ou um cientista de dados eficiente. Nem todas as técnicas são necessárias nas análises e geralmente existem outras maneiras de fazer o trabalho. Quanto maior for o tempo que você passar analisando dados com SQL, mais probabilidade terá de encontrar situações nas quais uma ou mais dessas técnicas serão úteis.

---

1 Embora no caso do Tableau, você possa resolver esse problema com a opção SQL inicial.

CAPÍTULO 9

Conclusão

No decorrer do livro, vimos que o SQL é uma linguagem flexível e poderosa para várias tarefas de análise de dados. Da criação de perfis de dados à análise de séries temporais e de texto e à detecção de anomalias, o SQL consegue atender a muitos dos requisitos comuns. Técnicas e funções também podem ser combinadas em qualquer instrução SQL para a execução de análises experimentais e a construção de conjuntos de dados complexos. Embora o SQL não consiga atender aos objetivos de todos os tipos de análises, ele se ajusta bem ao ecossistema de ferramentas de análise.

Neste último capítulo, discutirei alguns tipos de análise adicionais e destacarei como várias técnicas SQL abordadas no livro podem ser combinadas para sua execução. Em seguida, concluirei com alguns recursos que você pode usar para continuar sua jornada de dominar a análise de dados ou se aprofundar em tópicos específicos.

Análise de funil

Um funil é composto de uma série de etapas que devem ser concluídas para atingirmos um objetivo definido. O objetivo poderia ser registrar-se em um serviço, finalizar uma compra ou obter um certificado de conclusão de curso. As etapas do funil de compras de um site, por exemplo, poderiam incluir clicar no botão "Adicionar ao carrinho de compras", fornecer informações de envio, inserir um número de cartão de crédito e, por fim, clicar no botão "Fazer pedido".

A análise de funil combina elementos da análise de séries temporais, discutida no Capítulo 3, e da análise de coorte, discutida no Capítulo 4. Os dados da análise de funil vêm de uma série temporal de eventos, embora nesse caso esses eventos correspondam a ações distintas do mundo real em vez de serem repetições do mesmo evento. Medir a

retenção de etapa a etapa é um objetivo importante da análise de funil, mas nesse contexto geralmente usamos o termo *conversão*. Normalmente, entidades são removidas ao longo das etapas do processo, e um gráfico com seus valores em cada estágio acaba assumindo a forma de um funil doméstico – daí o nome.

Esse tipo de análise é usado para identificar áreas de fricção, dificuldade ou confusão. Etapas nas quais grandes números de usuários deixam de existir, ou que muitos não conseguem concluir, fornecem insights de oportunidades para otimização. Por exemplo, um processo de finalização de compra que solicite informações de cartão de crédito antes de exibir o valor total incluindo o envio pode afastar alguns possíveis compradores. Exibir o total antes dessa etapa encorajaria mais conclusões de compra. Essas alterações costumam ser temas de experimentos, discutidos no Capítulo 7. Os funis também podem ser monitorados para a detecção de eventos externos inesperados. Por exemplo, alterações nas taxas de conclusão podem estar relacionadas com um trabalho de relações públicas adequado (ou inadequado) ou com uma alteração no preço ou na tática de um concorrente.

A primeira etapa de uma análise de funil é descobrir a população base de todos os usuários, clientes ou outras entidades que eram elegíveis para entrar no processo. Em seguida, monte o conjunto de dados de conclusão de cada etapa de interesse, incluindo o objetivo final. Com frequência isso inclui uma ou mais *LEFT JOIN*s para a inserção de toda a população base, junto com quem concluiu cada etapa. Depois conte os usuários de cada etapa e divida essas contagens por etapa pela contagem total. Há duas maneiras de definir as consultas, dependendo de se todas as etapas serão obrigatórias.

Quando todas as etapas do funil forem obrigatórias – ou se você só quiser incluir usuários que tenham concluído todas as etapas – associe cada tabela com a tabela anterior com uma *LEFT JOIN*:

```
SELECT count(a.user_id) as all_users
,count(b.user_id) as step_one_users
,count(b.user_id) / count(a.user_id) as pct_step_one
,count(c.user_id) as step_two_users
,count(c.user_id) / count(b.user_id) as pct_one_to_two
```

```
FROM users a
LEFT JOIN step_one b on a.user_id = b.user_id
LEFT JOIN step_two c on b.user_id = c.user_id
;
```

Quando os usuários puderem pular uma etapa, ou se você quiser permitir essa possibilidade, associe com *LEFT JOIN* cada tabela à tabela que contém a população completa e calcule a parcela desse grupo inicial:

```
SELECT count(a.user_id) as all_users
,count(b.user_id) as step_one_users
,count(b.user_id) / count(a.user_id) as pct_step_one
,count(c.user_id) as step_two_users
,count(c.user_id) / count(b.user_id) as pct_step_two
FROM users a
LEFT JOIN step_one b on a.user_id = b.user_id
LEFT JOIN step_two c on a.user_id = c.user_id
;
```

É uma diferença sutil, mas na qual vale a pena prestar atenção e personalizar para o contexto específico. Considere inserir time boxes, para só incluir usuários que concluírem uma ação dentro de um intervalo de tempo definido, se os usuários puderem entrar novamente no funil após uma ausência longa. A análise de funil também pode incluir dimensões adicionais, como atributos de coorte e de outras entidades, para facilitar comparações e gerar hipóteses sobre por que um funil está ou não apresentando um bom desempenho.

Desistência, inatividade e outras definições de afastamento

O tópico da desistência (churn) surgiu no Capítulo 4, já que ela é basicamente o oposto da retenção. Geralmente as organizações preferem ou precisam criar uma definição específica de desistência para medi-la diretamente. Em algumas situações, há uma data final definida contratualmente, como no caso de software B2B. No entanto, a desistência costuma ser um conceito obscuro, e uma definição baseada em tempo é mais apropriada. Mesmo quando há uma data final contratual, medir quando um cliente parou de usar um produto pode ser um sinal precoce de um iminente cancelamento de contrato. As definições de desistência

também podem ser aplicadas a certos produtos ou recursos, mesmo quando o cliente não abandona totalmente a organização.

Uma métrica de desistência baseada em tempo conta os clientes como desistentes quando eles não compram ou interagem com um produto por um período de tempo, que geralmente vai de 30 dias a no máximo um ano. A duração exata depende muito do tipo de negócio e de padrões de uso típicos. A fim de chegar a uma boa definição de desistência, você pode usar a análise de lacunas para encontrar períodos típicos entre compras ou uso. Para executar a análise de lacunas, você precisará de uma série temporal de ações ou eventos, da função de janela `lag` e de alguma matemática de datas.

Como exemplo, podemos calcular as lacunas típicas existentes entre os mandatos dos representantes, usando o conjunto de dados de legisladores introduzido no Capítulo 4. Ignoraremos o fato de que com frequência os políticos não são eleitos em vez de decidirem deixar o cargo, já que dessa forma esse conjunto de dados terá a estrutura certa para esse tipo de análise. Primeiro encontraremos a lacuna média. Para fazê-lo, criaremos uma subconsulta que calculará a lacuna entre a data inicial (`start_date`) e a `start_date` anterior de cada legislador para cada mandato e, em seguida, encontraremos o valor médio na consulta externa. A data inicial anterior pode ser descoberta com o uso da função `lag`, e a lacuna como um intervalo de tempo é calculada com a função `age`:

```
SELECT avg(gap_interval) as avg_gap
FROM
(
    SELECT id_bioguide, term_start
    ,lag(term_start) over (partition by id_bioguide
                           order by term_start)
                           as prev
    ,age(term_start,
         lag(term_start) over (partition by id_bioguide
                               order by term_start)
         ) as gap_interval
    FROM legislators_terms
    WHERE term_type = 'rep'
```

```
) a
WHERE gap_interval is not null
;
avg_gap
-----------------------------------
2 years 2 mons 17 days 15:41:54.83805
```

Como era de se esperar, a média chega perto de dois anos, o que faz sentido, já que a duração do mandato desse cargo é de dois anos. Também podemos criar uma distribuição de lacunas para obter um limite de desistência realista. Nesse caso, converteremos a lacuna para meses:

```
SELECT gap_months, count(*) as instances
FROM
(
    SELECT id_bioguide, term_start
    ,lag(term_start) over (partition by id_bioguide
                           order by term_start)
                           as prev
    ,age(term_start,
         lag(term_start) over (partition by id_bioguide
                               order by term_start)
         ) as gap_interval
    ,date_part('year',
               age(term_start,
                   lag(term_start) over (partition by id_bioguide
                                         order by term_start)
                   )
              ) * 12
     +
     date_part('month',
               age(term_start,
                   lag(term_start) over (partition by id_bioguide
                                         order by term_start)
                   )
              ) as gap_months
    FROM legislators_terms
    WHERE term_type = 'rep'
) a
```

```
GROUP BY 1
;
gap_months  instances
----------  ---------
1.0         25
2.0         4
3.0         2
...         ...
```

Se `date_part` não for suportada em seu banco de dados, `extract` pode ser usada como alternativa (consulte o Capítulo 3 para ver uma explicação e exemplos). A saída pode ser representada em gráfico, como na Figura 9.1. Já que há uma cauda longa de meses, esse gráfico recebeu zoom para exibir o intervalo no qual se encontra a maioria das lacunas. A lacuna mais comum é de 24 meses, mas também existem várias centenas de casos por mês até alcançar 32 meses. Há outro pequeno aumento que ultrapassa os 100 casos em 47 e 48 meses. Com a média e a distribuição em mãos, eu provavelmente definiria um limite de 36 ou 48 meses e diria que qualquer representante que não tiver sido reeleito dentro dessa janela "desistiu".

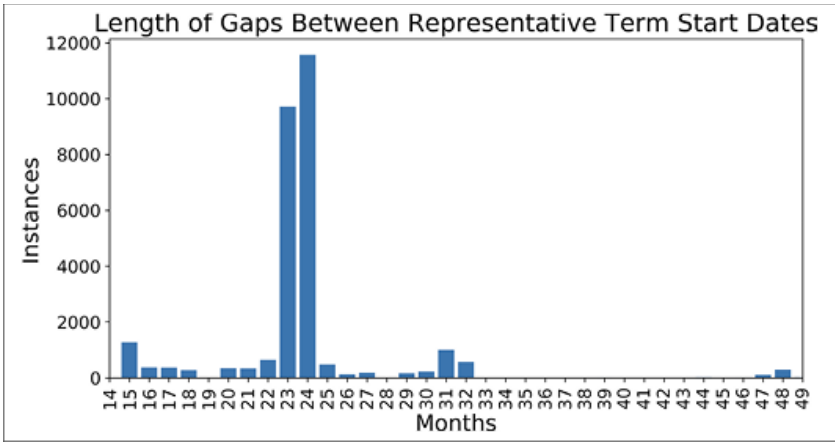


Figura 9.1: Distribuição da extensão das lacunas entre as datas iniciais dos mandatos dos representantes, exibindo intervalo de 10 a 59 meses.

Uma vez que você tiver definido um limite para a desistência, poderá monitorar a base de clientes com uma análise do "tempo passado desde o último evento". Esse evento pode ser a última compra, o último pagamento, a última vez que uma aplicação foi aberta ou qualquer métrica baseada em tempo que seja relevante para a organização. Para

fazer esse cálculo, você precisa de um conjunto de dados que tenha a data ou o timestamp mais recente para cada cliente. Se inicialmente usar uma série temporal, antes encontre o timestamp mais recente de cada cliente em uma subconsulta. Em seguida, aplique a matemática de datas para encontrar o tempo passado entre essa data e a data atual ou a última data do conjunto de dados se algum tempo tiver passado desde que a data foi obtida.

Por exemplo, poderíamos encontrar a distribuição de anos desde a última eleição na tabela `legislators_terms`. Na subconsulta, calcule a última data inicial usando a função `max` e depois encontre o tempo passado desde então usando a função `age`. Nesse caso, a data máxima do conjunto de dados, 5 de maio de 2019, é usada. Em um conjunto com dados atualizados, substitua por `current_date` ou uma expressão equivalente. A consulta externa encontra os anos do intervalo usando `date_part` e conta o número de legisladores:

```
SELECT date_part('year',interval_since_last) as years_since_last
,count(*) as reps
FROM
(
    SELECT id_bioguide
    ,max(term_start) as max_date
    ,age('2020-05-19',max(term_start)) as interval_since_last
    FROM legislators_terms
    WHERE term_type = 'rep'
    GROUP BY 1
) a
GROUP BY 1
;
years_since_last  reps
----------------  -----
0.0               6
1.0               440
2.0               1
...               ...
```

Um conceito relacionado é "inativo" (lapsed), que geralmente é usado como estágio intermediário entre clientes totalmente ativos e clientes que

desistiram (churned) e alternativamente pode ser chamado de "estagnado". Um cliente inativo pode apresentar alto risco de desistência porque já não o vemos durante algum tempo, mas ainda há uma boa probabilidade de ele retornar de acordo com nossa experiência passada. Em serviços prestados a consumidores, vi "inativo" abranger períodos de 7 a 30 dias, com a "desistência" sendo definida como um cliente que não usa o serviço há mais de 30 dias. As empresas costumam fazer experimentos de reativação de usuários inativos, usando táticas que vão do email ao contato feito pela equipe de suporte. Os clientes de cada estado podem ser definidos primeiro pelo cálculo de seu "tempo passado desde o último evento" como acima e, em seguida, por sua marcação em uma instrução CASE com o número de dias ou meses apropriado. Por exemplo, podemos agrupar os representantes de acordo com há quanto tempo eles foram eleitos:

```
SELECT
case when months_since_last <= 23 then 'Current'
     when months_since_last <= 48 then 'Lapsed'
     else 'Churned'
     end as status
,sum(reps) as total_reps
FROM
(
    SELECT
    date_part('year',interval_since_last) * 12
      + date_part('month',interval_since_last)
      as months_since_last
    ,count(*) as reps
    FROM
    (
        SELECT id_bioguide
        ,max(term_start) as max_date
        ,age('2020-05-19',max(term_start)) as interval_since_last
        FROM legislators_terms
        WHERE term_type = 'rep'
        GROUP BY 1
    ) a
```

```
    GROUP BY 1
) a
GROUP BY 1
;
status    total_reps
-------   ----------
Churned   10685
Current   446
Lapsed    105
```

Esse conjunto de dados contém mais de duzentos anos de mandatos de legisladores, logo é claro que muitas das pessoas incluídas já morreram, e algumas ainda estão vivas, mas se aposentaram. No contexto de uma empresa, o esperado seria que os clientes desistentes não superassem os clientes atuais por uma margem tão ampla, e poderíamos querer saber mais sobre os clientes inativos.

A maioria das organizações fica muito preocupada com a desistência, já que conquistar clientes costuma ser mais caro do que retê-los. Para você saber mais sobre os clientes de qualquer um dos status ou sobre o intervalo de tempo passado desde a última que vez que foram vistos, essas análises podem receber um nível ainda maior de detalhamento com o uso de um dos atributos de cliente disponíveis no conjunto de dados.

Análise de cesta de compras

Tenho três filhos, e, quando vou à mercearia, minha cesta de compras (ou com maior frequência meu carrinho de compras) fica rapidamente cheia de produtos para suas refeições semanais. Leite, ovos e pão costumam estar na lista, mas outros itens podem mudar dependendo do produto cuja estação tiver chegado, se for época de aulas ou de férias para as crianças ou se estivermos planejando preparar uma refeição especial. O nome análise de cesta de compras vem da prática de uma análise dos produtos que os consumidores compram juntos para a busca de padrões que possam ser usados em marketing, inserção de produtos em lojas ou outras decisões estratégicas. O objetivo da análise de cesta de compras pode ser encontrar grupos de itens que são comprados juntos. No entanto, ele também pode ser baseado em um produto específico: quando

alguém compra sorvete, o que mais essa pessoa compra?

Embora originalmente a análise de cesta de compras fosse baseada em itens que foram comprados juntos em uma única transação, o conceito pode ser estendido de várias maneiras. Um varejista ou uma loja de e-commerce poderia estar interessado na cesta de itens que um cliente comprou durante toda a sua vida. Serviços e o uso das características de um produto também podem ser analisados dessa forma. Serviços que normalmente são contratados juntos podem ser agrupados em uma nova oferta, como quando sites de viagens oferecem descontos se um voo, um hotel e o aluguel de um carro forem reservados juntos. Características de produtos que são usados juntos podem ser inseridas na mesma janela de navegação ou utilizadas como sugestões de para onde o desenvolvimento de uma aplicação deve seguir. A análise de cesta de compras também pode ser usada para identificar personas, ou segmentos, de stakeholders que são então utilizadas em outros tipos de análises.

Para encontrar as cestas de compras mais comuns, usando todos os itens de uma cesta, podemos empregar a função `string_agg` (ou uma função análoga, dependendo do tipo de banco de dados – consulte o Capítulo 5). Por exemplo, suponhamos que tivéssemos uma tabela `purchases` com uma linha para cada produto comprado por um `customer_id`. Primeiro, use a função `string_agg` para encontrar a lista de produtos comprados por cliente em uma subconsulta. Em seguida, faça o agrupamento por essa lista e conte o número de clientes:

```
SELECT products
,count(customer_id) as customers
FROM
(
    SELECT customer_id
    ,string_agg(product,', ') as products
    FROM purchases
    GROUP BY 1
) a
GROUP BY 1
ORDER BY 2 desc
;
```

Essa técnica funciona bem quando há um número relativamente pequeno de itens possíveis. Outra opção é encontrar pares de produtos comprados juntos. Para fazer isso, use uma self-*JOIN* para associar a tabela `purchases` com ela própria, fazendo a junção em `customer_id`. A segunda condição de *JOIN* resolve o problema de entradas duplicadas que difiram somente na ordem. Por exemplo, considere um cliente que comprasse maçãs e bananas – sem essa cláusula, o conjunto de resultados incluiria "maçãs, bananas" e "bananas, maçãs". A cláusula `b.product > a.product` assegura que somente uma dessas variações seja incluída e também exclui por filtragem resultados em que um produto seja comparado com ele próprio:

```
SELECT product1, product2
,count(customer_id) as customers
FROM
(
    SELECT a.customer_id
    ,a.product as product1
    ,b.product as product2
    FROM purchases a
    JOIN purchases b on a.customer_id = b.customer_id
    and b.product > a.product
) a
GROUP BY 1,2
ORDER BY 3 desc
;
```

Esse código pode ser estendido para incluir três ou mais produtos com a inserção de *JOIN*s adicionais. Para incluir cestas de compra que contenham apenas um item, altere *JOIN* para uma *LEFT JOIN*.

Existem alguns desafios comuns na execução de uma análise de cesta de compras. O primeiro é o desempenho, em especial quando há um grande catálogo de produtos, serviços ou recursos. Os cálculos resultantes podem tornar-se lentos no banco de dados, principalmente quando o objetivo for encontrar grupos de três ou mais itens e, portanto, o SQL tiver três ou mais self-*JOIN*s. Considere filtrar as tabelas com cláusulas *WHERE* para remover itens comprados com pouca frequência antes de executar as *JOIN*s. Outro desafio surge quando alguns itens são tão comuns que

sobrecarregam todas as outras combinações. Por exemplo, o leite é comprado com tanta frequência que grupos que o contêm junto com algum outro item encabeçam a lista de combinações. Embora precisos, os resultados da consulta podem ser irrelevantes do ponto de vista prático. Nesse caso, considere remover totalmente os itens mais comuns, novamente com uma cláusula *WHERE*, antes de executar as *JOIN*s. Isso deve produzir o benefício adicional de melhorar o desempenho da consulta ao diminuir o conjunto de dados.

O último desafio da análise de cesta de compras é a profecia autorrealizável. Itens que aparecerem juntos em uma análise de cesta de compras podem ser comercializados em conjunto, aumentando a frequência com que serão comprados juntos. Isso reforçará ainda mais sua comercialização em conjunto, levando a mais compras associadas e assim por diante. Produtos que tiverem melhor encaixe podem nunca ter uma chance, simplesmente porque não apareceram na análise original e não se tornaram candidatos à promoção. A famosa *correlação de cerveja e fraldas*[1] (*https://oreil.ly/4d5PF*) é apenas um dos exemplos. Várias técnicas de machine learning e grandes empresas online tentaram lidar com esse problema, e há muitas direções interessantes ainda a serem desenvolvidas para a análise nessa área.

Recursos

A análise de dados como profissão (ou até mesmo como hobby!) requer uma combinação de proficiência técnica, conhecimento da área, curiosidade e habilidades de comunicação. Resolvi que seria interessante compartilhar alguns de meus recursos favoritos para que você possa recorrer a eles ao continuar sua jornada, tanto para aprender mais quanto para praticar suas novas habilidades com conjuntos de dados reais.

Livros e blogs

Embora este livro demande conhecimento funcional de SQL, bons recursos sobre os aspectos básicos ou para uma recapitulação são:

- Forta, Ben. Sams *SQL em 10 Minutos por Dia*. 5. ed. São Paulo: Novatec, 2020.

- A empresa de software Mode oferece um *tutorial de SQL* com interface de consulta interativa, útil para você praticar suas habilidades (*https://mode.com/sql-tutorial/*).

Não há um estilo SQL definitivo e universalmente aceito, mas você pode achar úteis o *SQL Style Guide* (*https://www.sqlstyle.guide/*)e o *Modern SQL Style Guide* (*https://oreil.ly/rsxBh*). É bom ressaltar que seus estilos não coincidem exatamente com os usados neste livro ou um com o outro. Acho que usar um estilo que seja tanto autoconsistente quanto legível é a consideração mais importante.

A abordagem que você usar para a análise e para comunicar os resultados pode ser tão importante quanto o código criado. Dois bons livros que aperfeiçoam os dois aspectos são:

- Hubbard, Douglas W. *How to Measure Anything: Finding the Value of "Intangibles" in Business*. 2. ed. Hoboken, NJ: Wiley, 2010.
- Kahneman, Daniel. *Rápido e Devagar: Duas Formas de Pensar*. São Paulo: Objetiva, 2012.

O blog *Towards Data Science* (*https://towardsdatascience.com/*) é uma ótima fonte de artigos sobre muitos tópicos referentes à análise. Embora várias das publicações feitas nele enfoquem Python como linguagem de programação, geralmente as abordagens e técnicas podem ser adaptadas para SQL.

Para ver uma demonstração divertida de correlação versus causa, acesse *Spurious Correlations de Tyler Vigen (https://www.tylervigen.com/spurious-correlations)*.

As expressões regulares podem ser complexas. Se você quiser entender melhor ou resolver casos complicados não abordados neste livro, um bom recurso é:

- Forta, Ben. *Learning Regular Expressions*. Boston: Addison-Wesley, 2018.

O teste randomizado tem um longo histórico e afeta muitas áreas das ciências natural e social. No entanto, em comparação com a estatística, a análise de experimentos online ainda é relativamente nova. Muitos textos estatísticos clássicos fornecem uma boa introdução, mas discutem problemas em que o tamanho da amostra é menor; logo, não conseguem abordar as oportunidades e os desafios exclusivos do teste online. Dois

livros interessantes que discutem os experimentos online são:

- Georgiev, Georgi Z. *Statistical Methods in Online A/B Testing*. Sofia, Bulgária: autopublicação, 2019.
- Kohavi, Ron, Diane Tang e Ya Xu. *Trustworthy Online Controlled Experiments: A Practical Guide to A/B Testing*. Cambridge, UK: Cambridge University Press, 2020.

Awesome A/B Tools de Evan Miller (https://www.evanmiller.org/ab-testing/) tem as calculadoras para experimentos de resultado binário e contínuo, assim como vários outros testes que podem ser úteis para designs de experimentos além dos encontrados neste livro.

Conjuntos de dados

A melhor maneira de aprender e melhorar as habilidades em SQL é colocá-las em uso com dados reais. Se você está empregado e tem acesso a um banco de dados dentro de sua organização, esse seria um bom local para começar porque já deve haver um contexto referente a como os dados são produzidos e ao que eles significam. Contudo, existem muitos conjuntos de dados públicos interessantes que você pode analisar, e eles abrangem muitos tópicos. Abaixo estão listados alguns pontos de partida se você estiver procurando conjunto de dados interessantes:

- *Data Is Plural* (*https://www.data-is-plural.com/*) é uma newsletter de conjuntos de dados novos e interessantes, e o archive *Data Is Plural* (*https://dataset-finder.netlify.app/*) é uma coleção valiosa de conjuntos de dados que podem ser pesquisados.
- *FiveThirtyEight* (*https://fivethirtyeight.com/*) é um site jornalístico que aborda política, esportes e ciência à luz dos dados. Os conjuntos de dados existentes por trás das histórias estão no site *FiveThirtyEight* do GitHub (*https://github.com/fivethirtyeight/data*).
- *Gapminder* (*https://www.gapminder.org/data/*) é uma fundação sueca que publica dados anuais de muitos indicadores de desenvolvimento humano e econômico, incluindo vários provenientes do Banco Mundial.
- As Nações Unidas publicam várias estatísticas. O Departamento das Nações Unidas para Assuntos Econômicos e Sociais produz dados

sobre a *dinâmica das populações* (https://population.un.org/wpp/Download/Standard/Population/) em um formato relativamente fácil de usar.

- O Kaggle hospeda competições de análise de dados e tem uma *biblioteca de conjuntos de dados* (https://www.kaggle.com/datasets) que pode ser baixada e analisada até mesmo fora das competições formais.
- Muitos governos em todos os níveis, do nacional ao local, adotaram o movimento por dados abertos e publicam várias estatísticas. O site *Data.gov* (https://data.gov/open-gov/) mantém uma lista de sites tanto nos Estados Unidos quanto ao redor do globo que é um bom ponto de partida.

Considerações finais

Espero que você tenha achado úteis as técnicas e os códigos deste livro. Acho importante ter uma boa base sobre as ferramentas que usamos, e há muitas funções e expressões SQL úteis que podem tornar suas análises mais rápidas e precisas. No entanto, desenvolver boas habilidades em análise não é apenas aprender as últimas técnicas e linguagens. Uma boa análise acontece quando fazemos perguntas apropriadas, reservamos algum tempo para entender os dados e a área, aplicamos técnicas de análise adequadas para gerar respostas confiáveis e de alta qualidade e, para concluir, comunicamos os resultados para o público-alvo de uma maneira que seja relevante e que ajude na tomada de decisões. Mesmo depois de quase 20 anos de trabalho com SQL, ainda fico empolgado querendo descobrir novas maneiras de aplicá-lo, novos conjuntos de dados para usar, e todos os insights existentes no mundo esperando pacientemente para serem descobertos.

---

1 N.T.: A lenda estatística das 'fraldas & cervejas' representa emblematicamente a descoberta de padrões inusitados de consumo que fazem sentido quando compreendemos todo o contexto de vida e a rotina de compras de um jovem pai com seu novo bebê em casa durante todo o fim de semana, talvez um bom motivo para recompensar a situação com um estoque extra de cerveja.

Sobre o autor

Cathy Tanimura se dedica a conectar as pessoas e organizações aos dados dos quais elas precisam. Há mais de 20 anos ela analisa dados para vários segmentos, que vão do setor financeiro ao de software B2B e ao de serviços ao consumidor. Ela tem experiência na análise de dados com SQL na maioria dos grandes bancos de dados proprietários e open source. Construiu e gerenciou equipes e infraestruturas de dados em muitas das principais empresas de tecnologia. Cathy também é uma oradora frequente nas conferências mais importantes, falando sobre tópicos que incluem a construção de culturas de dados, o desenvolvimento de produtos baseado em dados, além da análise de dados.

Colofão

O animal da capa de *SQL para Análise de Dados* é um pega verde (*Cissa chinensis*). Geralmente chamado de pega verde comum, esse pássaro com tom de joia é membro da família dos corvos. Encontrada nas florestas perenes e de bambus das planícies do nordeste da Índia, da região central da Tailândia, da Malásia, de Sumatra e do noroeste de Bornéu, essa espécie de pássaro é ruidosa e altamente sociável. No ambiente selvagem, ela pode ser identificada por sua plumagem de cor jade, que contrasta elegantemente com seu bico vermelho e uma faixa preta ao longo dos olhos. Ela também tem uma cauda de ponta branca e asas avermelhadas.

Altamente sociável e barulhento, o pega verde pode ser identificado pelos seus guinchos estridentes seguidos de uma nota semelhante a um "sugar" de som vago e resoluto. Eles costumam ser difíceis de detectar porque deslizam de uma árvore para a outra nos níveis médios superiores da floresta. Constroem seus ninhos em árvores, grandes arbustos e emaranhados de videiras trepadeiras. Às vezes chamados de cissas caçadores, são principalmente carnívoros – consumindo vários invertebrados, assim como jovens pássaros e ovos, pequenos répteis, e mamíferos.

O pega verde é fascinante devido à sua habilidade de alterar as cores. Embora sejam verde-jade no ambiente selvagem, foram observados com uma cor distintamente turquesa em cativeiro. Eles obtêm sua coloração verde de uma combinação de duas fontes: uma estrutura de penas especial que produz coloração azul devido à refração da luz nas penas, e carotenoides – pigmentos amarelo, laranja e vermelho que vêm da dieta do pássaro. A exposição prolongada à luz do sol destrói os carotenoides e é por isso que a ave fica turquesa.

A espécie pega verde é encontrada em uma extensão bastante grande e, embora a tendência populacional pareça ser a diminuição, o declínio não é suficientemente rápido para levá-la para a categoria de vulnerabilidade. Logo, seu status de conservação atual é de "Menor Preocupação".

A ilustração da capa é de Karen Montgomery, baseada na gravura em preto e branco da *English Cyclopedia*. As fontes da capa são Gilroy Semibold e Guardian Sans. A fonte do texto é Adobe Minion Pro, a fonte do título é Adobe Myriad Condensed e a dos códigos é Ubuntu Mono de Dalton Maag.