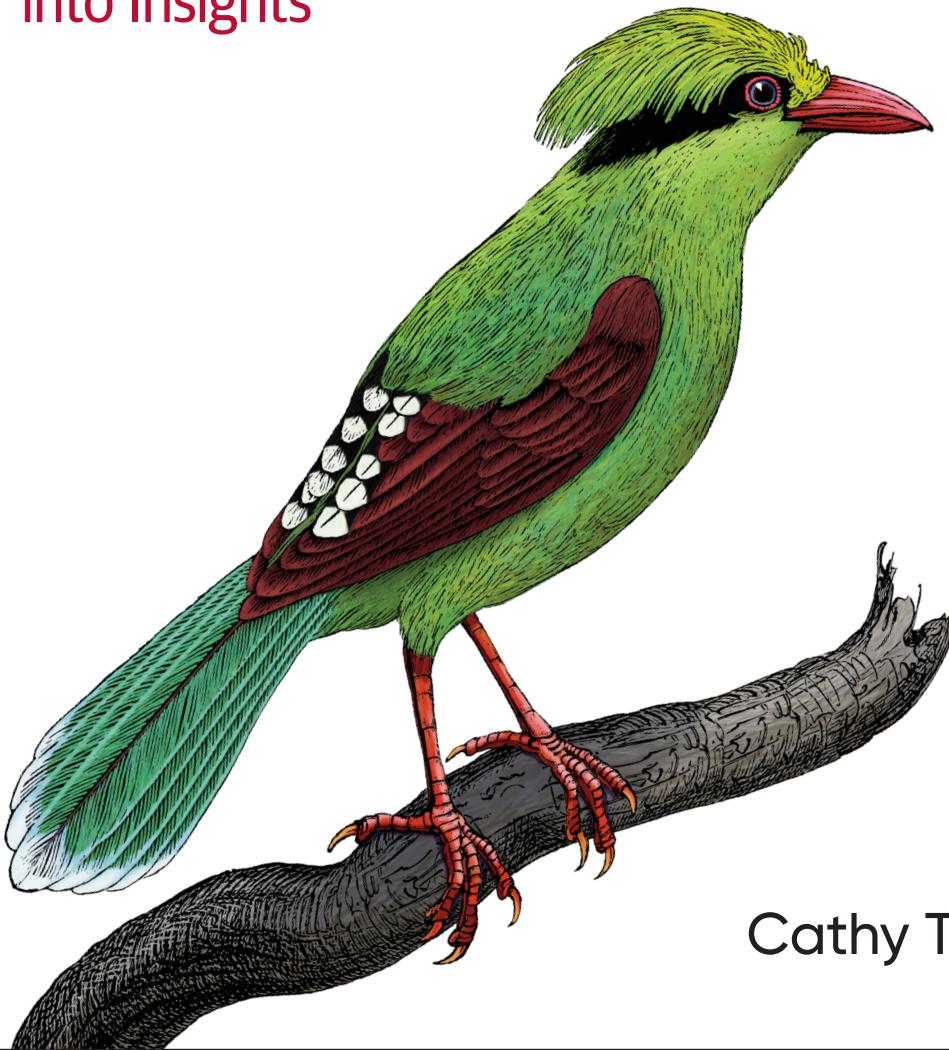


O'REILLY®

# SQL for Data Analysis

Advanced Techniques for Transforming Data  
into Insights



Cathy Tanimura

# SQL for Data Analysis

With the explosion of data, computing power, and cloud data warehouses, SQL has become an even more indispensable tool for the savvy analyst or data scientist. This practical book reveals new and hidden ways to improve your SQL skills, solve problems, and make the most of SQL as part of your workflow.

You'll learn how to use both common and exotic SQL functions such as joins, window functions, subqueries, and regular expressions in new, innovative ways—as well as how to combine SQL techniques to accomplish your goals faster, with understandable code. If you work with SQL databases, this is a must-have reference.

- Learn the key steps for preparing your data for analysis
- Perform time series analysis using SQL's date and time manipulations
- Use cohort analysis to investigate how groups change over time
- Use SQL's powerful functions and operators for text analysis
- Detect outliers in your data and replace them with alternate values
- Establish causality using experiment analysis, also known as A/B testing

Cathy Tanimura has been analyzing data for over 20 years across a wide range of industries, from finance to B2B software to consumer services. With a passion for connecting people and organizations to the data they need, Cathy has built and managed data teams and data infrastructure at several leading tech companies. She has experience analyzing data with SQL across most of the major proprietary and open source databases.

"I can't count the number of 'AHA!' moments I had, despite my 20 years of doing analysis in various SQL environments. I will be buying a copy for every current and future member of my team."

—Stuart Kim-Brown  
PhD, B2C and SaaS Product  
Analytics Leader

"So great to finally see a SQL book written specifically for those looking to do data analysis. Any aspiring analyst or data scientist would benefit from reading about the various methods of data analysis that can be tackled with SQL. Detailed examples and code make this book both a great introduction and reference."

—Dan Voorhies  
Director of Analytics, Zillow

---

DATA

US \$59.99      CAN \$79.99

ISBN: 978-1-492-08878-3



9 781492 088783

Twitter: @oreillymedia  
[facebook.com/oreilly](http://facebook.com/oreilly)

---

# **SQL for Data Analysis**

*Advanced Techniques for Transforming  
Data into Insights*

*Cathy Tanimura*

Beijing • Boston • Farnham • Sebastopol • Tokyo

**O'REILLY®**

## **SQL for Data Analysis**

by Cathy Tanimura

Copyright © 2021 Cathy Tanimura. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://oreilly.com>). For more information, contact our corporate/institutional sales department: 800-998-9938 or [corporate@oreilly.com](mailto:corporate@oreilly.com).

**Acquisitions Editor:** Andy Kwan

**Indexer:** Ellen Troutman-Zaig

**Development Editors** Amelia Blevins and Shira Evans

**Interior Designer:** David Futato

**Production Editor:** Kristen Brown

**Cover Designer:** Karen Montgomery

**Copyeditor:** Arthur Johnson

**Illustrator:** Kate Dullea

**Proofreader:** Paula L. Fleming

September 2021: First Edition

### **Revision History for the First Edition**

2021-09-09: First Release

See <http://oreilly.com/catalog/errata.csp?isbn=9781492088783> for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *SQL for Data Analysis*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the author, and do not represent the publisher's views. While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-492-08878-3

[LSI]

## Prefácio

Nos últimos 20 anos, passei muitas horas de trabalho manipulando dados com SQL. Durante a maior parte desses anos, trabalhei em empresas de tecnologia abrangendo uma ampla gama de setores de consumo e business-to-business. Nesse período, os volumes de dados aumentaram drasticamente e a tecnologia que uso melhorou aos trancos e barrancos. Os bancos de dados estão mais rápidos do que nunca, e as ferramentas de relatório e visualização usadas para comunicar o significado dos dados estão mais poderosas do que nunca. Uma coisa que permaneceu notavelmente constante, no entanto, é o SQL sendo uma parte fundamental da minha caixa de ferramentas.

Lembro-me de quando aprendi SQL pela primeira vez. Comecei minha carreira em finanças, onde as planilhas dominam, e fiquei muito bom em escrever fórmulas e memorizar todos aqueles atalhos de teclado. Um dia eu fiquei totalmente nerd e apertei Ctrl e Alt em cada tecla do meu teclado só para ver o que aconteceria (e então criei uma folha de dicas para meus colegas). Isso foi em parte diversão e em parte sobrevivência: quanto mais rápido eu fosse com minhas planilhas, maior a probabilidade de terminar meu trabalho antes da meia-noite para poder ir para casa e dormir um pouco. O domínio de planilhas me fez entrar na minha próxima função, uma startup onde fui apresentado pela primeira vez a bancos de dados e SQL.

Part of my role involved crunching inventory data in spreadsheets, and thanks to early internet scale, the data sets were sometimes tens of thousands of rows. This was “big data” at the time, at least for me. I got in the habit of going for a cup of coffee or for lunch while my computer’s CPU was occupied with running its vlookup magic. One day my manager went on vacation and asked me to tend to the data warehouse he’d built on his laptop using Access. Refreshing the data involved a series of steps: running SQL queries in a portal, loading the resulting csv files into the database, and then refreshing the spreadsheet reports. After the first successful load, I started tinkering, trying to understand how it worked, and pestering the engineers to show me how to modify the SQL queries.

Fiquei viciado e, mesmo quando pensei que poderia mudar de direção com minha carreira, continuei voltando aos dados. Manipulando dados, respondendo perguntas, ajudando meus

colegas trabalham melhor e de forma mais inteligente, e aprender sobre 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. Peguei um livro sobre sintaxe básica, li em uma noite e, a partir daí, aprendi principalmente por tentativa e erro. Nos dias em que eu estava aprendendo, eu consultava bancos de dados de produção diretamente e derrubava o site mais de uma vez com meu SQL excessivamente ambicioso (ou mais provavelmente apenas mal escrito). Felizmente, minhas habilidades melhoraram e, ao longo dos anos, aprendi a trabalhar para frente a partir dos dados em tabelas e para trás a partir da saída necessária, resolvendo desafios e quebra-cabeças técnicos e lógicos para escrever consultas que retornavam os dados corretos. Acabei projetando e construindo data warehouses para coletar dados de diferentes fontes e evitar derrubar bancos de dados críticos de produção. Aprendi muito sobre quando e como agregar dados antes de escrever a consulta SQL e quando deixar os dados em uma forma mais bruta.

Comparei notas com outras pessoas que entraram em dados na mesma época e está claro que aprendemos principalmente da mesma maneira ad hoc. Os sortudos entre nós tinham colegas com quem compartilhar técnicas. A maioria dos textos SQL são introdutórios e básicos (com certeza há um lugar para eles!) ou então voltados para desenvolvedores de banco de dados. Existem poucos recursos para usuários avançados de SQL que estão focados no trabalho de análise. O conhecimento tende a ser trancado em indivíduos ou pequenas equipes. Um objetivo deste livro é mudar isso, dando aos praticantes uma referência sobre como resolver problemas comuns de análise com SQL, e espero inspirar novas pesquisas sobre dados usando técnicas que talvez você não tenha visto antes.

## Convenções utilizadas neste livro

As seguintes convenções tipográficas são usadas neste livro:

### *Italic*

Indica novos termos, URLs, endereços de e-mail, nomes de arquivo, extensões de arquivo e palavras-chave.

### **Constant width**

Usado para listagens de programas, bem como dentro de parágrafos para se referir a elementos de programa, como nomes de variáveis ou funções, bancos de dados, variáveis de ambiente e instruções.

### **Constant width bold**

Mostra comandos ou outro texto que deve ser digitado literalmente pelo usuário.

### **Constant width italic**

Mostra o texto que deve ser substituído por valores fornecidos pelo usuário ou por valores determinados pelo contexto.



Este elemento significa uma dica ou sugestão.



Este elemento significa uma nota geral.



Este elemento indica um aviso ou cuidado.

## Usando exemplos de código

Material suplementar (exemplos de código, exercícios, etc.) está disponível para download em

[https://github.com/cathytnimura/sql\\_book](https://github.com/cathytnimura/sql_book).

Se você tiver uma pergunta técnica ou um problema usando os exemplos de código, envie um e-mail para [bookquestions@oreilly.com](mailto:bookquestions@oreilly.com).

Este livro está aqui para ajudá-lo a fazer o seu trabalho. Em geral, se um código de exemplo for oferecido com este livro, você poderá usá-lo em seus programas e documentação. Você não precisa entrar em contato conosco para obter permissão, a menos que esteja reproduzindo uma parte significativa do código. Por exemplo, escrever um programa que usa vários pedaços de código deste livro não requer permissão. Vender ou distribuir exemplos de livros da O'Reilly requer permissão. Responder a uma pergunta citando este livro e citando um código de exemplo não requer permissão. A incorporação de uma quantidade significativa de código de exemplo deste livro na documentação do seu produto requer permissão.

Apreciamos, mas geralmente não exigimos, atribuição. Uma atribuição geralmente inclui o título, autor, editora e ISBN. Por exemplo: “SQL for Data Analysis por Cathy Tanimura (O'Reilly). Copyright 2021 Cathy Tanimura, 978-1-492-08878-3.”

Se você achar que o uso de exemplos de código está fora do uso justo ou da permissão dada acima, sinta-se à vontade para entrar em contato conosco em [permissions@oreilly.com](mailto:permissions@oreilly.com).

# O'Reilly Online Learning



Por mais de 40 anos, a O'Reilly Media forneceu treinamento, conhecimento e insights em tecnologia e negócios para ajudar as empresas a ter sucesso.

Nossa rede exclusiva de especialistas e inovadores compartilha seu conhecimento e experiência por meio de livros, artigos e nossa plataforma de aprendizado online. A plataforma de aprendizado on-line da O'Reilly oferece acesso sob demanda a cursos de treinamento ao vivo, caminhos de aprendizado aprofundados, ambientes de codificação interativos e uma vasta coleção de texto e vídeo da O'Reilly e mais de 200 outras editoras. Para mais informações visite <http://oreilly.com>.

## Como entrar em contato conosco

Envie comentários e perguntas sobre este livro à editora:

O'Reilly Media, Inc.  
1005 Gravenstein Highway North  
Sebastopol, CA 95472  
800-998-9938 (in the United States or Canada)  
707-829-0515 (international or local)  
707-829-0104 (fax)

Temos uma página web para este livro, onde listamos erratas, exemplos e qualquer informação adicional. Você pode acessar esta página em <https://oreil.ly/sql-data-analysis>.

Email [bookquestions@oreilly.com](mailto:bookquestions@oreilly.com) para comentar ou fazer perguntas técnicas sobre este livro.

Para notícias e informações sobre nossos livros e cursos, visite <http://oreilly.com>.

Encontre-nos no Facebook: <http://facebook.com/oreilly>

Siga-nos no Twitter: <http://twitter.com/oreillymedia>

Assista-nos no YouTube: <http://www.youtube.com/oreillymedia>

## 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 pelo processo e deram feedback útil ao longo do caminho. Kristen Brown conduziu o livro durante o processo de produção. Arthur Johnson melhorou a qualidade e clareza do texto e inadvertidamente me fez pensar mais profundamente sobre palavras-chave SQL.

Muitos colegas ao longo dos anos desempenharam um papel importante em minha jornada SQL, e sou grato por seus tutoriais, dicas e código compartilhado, e pelo tempo gasto pensando em maneiras de resolver problemas de análise ao longo dos anos. Sharon Lin abriu meus olhos para expressões regulares. Elyse Gordon me deu muitos conselhos sobre como escrever livros. Dave Hoch e nossas conversas sobre análise de experimentos inspiraram o Capítulo 7. Dan, Jim e Stu, da Star Chamber, há muito são meus caras favoritos para sair. Também sou grato a todos os colegas que fizeram perguntas difíceis ao longo dos anos e, uma vez respondidas, fizeram perguntas ainda mais difíceis.

Gostaria de agradecer ao meu marido Rick, ao filho Shea, às filhas Lily e Fiona e à mãe Janet por seu amor, incentivo e, acima de tudo, pelo presente de tempo para trabalhar neste livro. Amy, Halle, Jessi e o Den of Slack me mantiveram sã e rindo durante meses de escrita e bloqueio pandêmico.

# CAPÍTULO 1

---

# Análise com SQL

Se você está lendo este livro, provavelmente está interessado em análise de dados e em usar SQL para realizá-la. Você pode ter experiência com análise de dados, mas é novo em SQL, ou talvez tenha experiência em SQL, mas é novo em análise de dados. Ou você pode ser totalmente novo em ambos os tópicos. Seja qual for o seu ponto de partida, este capítulo estabelece as bases para os tópicos abordados no restante do livro e garante que tenhamos um vocabulário comum. Começarei com uma discussão sobre o que é análise de dados e, em seguida, passarei para uma discussão sobre SQL: o que é, por que é tão popular, como se compara a outras ferramentas e como se encaixa na análise de dados. Então, como a análise de dados moderna está tão entrelaçada com as tecnologias que a permitiram, concluirrei com uma discussão sobre os diferentes tipos de bancos de dados que você pode encontrar em seu trabalho, por que eles são usados e o que tudo isso significa para o SQL que você escreve.

## O que é análise de dados?

Coletar e armazenar dados para análise é uma atividade muito humana. Os sistemas para rastrear os estoques de grãos, impostos e a população remontam a milhares de anos, e as [raízes da Estatística](#) datam de centenas de anos. Disciplinas relacionadas, incluindo controle estatístico de processos, pesquisa operacional e cibernética, explodiram no século XX. Muitos nomes diferentes são usados para descrever a disciplina de análise de dados, como business intelligence (BI), análise, ciência de dados e ciência da decisão, e os profissionais têm uma variedade de cargos. A análise de dados também é feita por profissionais de marketing, gerentes de produto, analistas de negócios e várias outras pessoas. Neste livro, usarei os termos *analista* de dados e *cientista* de forma intercambiável para significar a pessoa que trabalha com SQL para entender os dados. Vou me referir ao software usado para construir relatórios e dashboards como *ferramentas de BI*.

A análise de dados no sentido contemporâneo foi possibilitada e está entrelaçada com a história da computação. As tendências tanto na pesquisa quanto na comercialização o moldaram,

e a história inclui um quem é quem de pesquisadores e grandes empresas, sobre o qual falaremos na seção sobre SQL. A análise de dados combina o poder da computação com técnicas de estatísticas tradicionais. A análise de dados é a descoberta de dados de parte, interpretação de dados de parte e comunicação de dados de parte. Muitas vezes o objetivo da análise de dados é melhorar a tomada de decisão, por humanos e cada vez mais por máquinas através da automação.

Uma metodologia sólida é fundamental, mas a análise é mais do que apenas produzir o número certo. É sobre curiosidade, fazer perguntas e o “porquê” por trás dos números. Trata-se de padrões e anomalias, descobrindo e interpretando pistas sobre como as empresas e os humanos se comportam. Às vezes, a análise é feita em um conjunto de dados coletados para responder a uma pergunta específica, como em um ambiente científico ou em um experimento online. A análise também é feita em dados gerados como resultado de negócios, como nas vendas de produtos de uma empresa, ou gerados para fins de análise, como rastreamento de interação do usuário em sites e aplicativos móveis. Esses dados têm uma ampla gama de aplicações possíveis, desde a solução de problemas até o planejamento de melhorias na interface do usuário (IU), mas geralmente chegam em um formato e volume de tal forma que os dados precisam ser processados antes de gerar respostas. O Capítulo 2 abordará a preparação de dados para análise e o Capítulo 8 discutirá algumas das preocupações éticas e de privacidade com as quais todos os profissionais de dados devem estar familiarizados.

É difícil pensar em um setor que não tenha sido tocado pela análise de dados: manufatura, varejo, finanças, saúde, educação e até mesmo o governo foram alterados por ela. As equipes esportivas empregaram a análise de dados desde os primeiros anos do mandato de Billy Beane como gerente geral do Oakland Athletics, que ficou famoso pelo livro de Michael Lewis, *Moneyball* (Norton). A análise de dados é usada em marketing, vendas, logística, desenvolvimento de produtos, design de experiência do usuário, centros de suporte, recursos humanos e muito mais. A combinação de técnicas, aplicativos e poder de computação levou à explosão de campos relacionados, como engenharia de dados e ciência de dados.

A análise de dados é, por definição, feita em dados históricos, e é importante lembrar que o passado não necessariamente prevê o futuro. O mundo é dinâmico, e as organizações também são dinâmicas – novos produtos e processos são introduzidos, os concorrentes sobem e descem, os climas sociopolíticos mudam. As críticas são feitas contra a análise de dados por ser uma visão retrógrada. Embora essa caracterização seja verdadeira, tenho visto organizações ganharem um enorme valor ao analisar 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 informadas e intervalos previstos de resultados, que às vezes estarão errados, mas muitas vezes estarão certos. Dados passados podem apontar lacunas, fraquezas e oportunidades. Ele permite que as organizações otimizem, economizem dinheiro e reduzam riscos e fraudes. Também pode ajudar as organizações a encontrar oportunidades e pode se tornar a base de novos produtos que encantam os clientes.



As organizações que não fazem alguma forma de análise de dados são poucas e distantes hoje em dia, mas ainda existem alguns obstáculos. Por que algumas organizações não usam a análise de dados? Um argumento é a relação custo-valor. Coletar, processar e analisar dados exige trabalho e algum nível de investimento financeiro. Algumas organizações são muito novas ou muito aleatórias. Se não houver um processo consistente, é difícil gerar dados consistentes o suficiente para serem analisados. Finalmente, há considerações éticas. A coleta ou armazenamento de dados sobre determinadas pessoas em determinadas situações pode ser regulamentada ou até mesmo proibida. Os dados sobre crianças e intervenções de saúde são sensíveis, por exemplo, e há regulamentações extensas em torno de sua coleta. Mesmo as organizações que são orientadas por dados precisam cuidar da privacidade do cliente e pensar muito sobre quais dados devem ser coletados, por que são necessários e por quanto tempo devem ser armazenados. Regulamentos como o Regulamento Geral de Proteção de Dados da União Europeia, ou GDPR, e a Lei de Privacidade do Consumidor da Califórnia, ou CCPA, mudaram a maneira como as empresas pensam sobre os dados do consumidor. Discutiremos esses regulamentos com mais profundidade no [Capítulo 8](#). Como profissionais de dados, devemos sempre pensar nas implicações éticas de nosso trabalho.

Ao trabalhar com organizações, gosto de dizer às pessoas que a análise de dados não é um projeto que termina em uma data fixa – é um modo de vida. Desenvolver uma mentalidade informada por dados é um processo, e colher os frutos é uma jornada. Desconhecidas se tornam conhecidas, perguntas difíceis são eliminadas até que haja respostas e as informações mais críticas são incorporadas em painéis que impulsionam 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 e difícil de dominar. A tecnologia pode ser aprendida, particularmente SQL. Muitos problemas, como otimizar gastos com marketing ou detectar fraudes, são familiares e se traduzem em todas as empresas. Cada organização é diferente e cada conjunto de dados tem peculiaridades, portanto, mesmo problemas familiares podem representar novos desafios. Comunicar resultados é uma habilidade. Aprender a fazer boas recomendações e se tornar um parceiro confiável de uma organização leva tempo. Na minha experiência, a análise simples apresentada de forma persuasiva tem mais impacto do que a análise sofisticada apresentada de forma deficiente. A análise de dados bem-sucedida também requer parceria. Você pode ter ótimos insights, mas se não houver ninguém para executá-los, você realmente não causou impacto. Mesmo com toda a tecnologia, ainda é sobre pessoas, e os relacionamentos são importantes.

## Por que SQL?

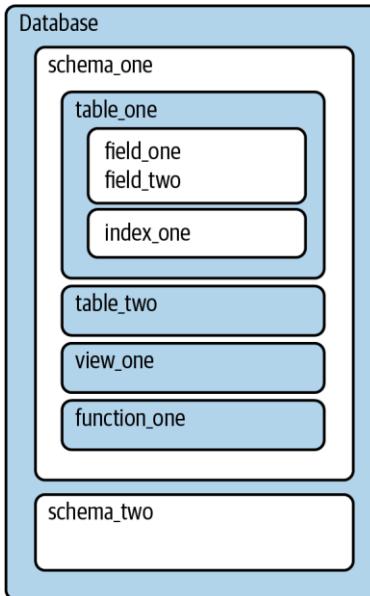
Esta seção descreve o que é SQL, os benefícios de usá-lo, como ele se compara a outras linguagens comumente usadas para análise e, finalmente, como o SQL se encaixa no fluxo de trabalho de análise.

### O que é SQL?

SQL é a linguagem usada para se comunicar com bancos de dados. A sigla significa Structured Query Language e é pronunciada como “sequel” ou dizendo cada letra, como em “ess cue el”. Esta é apenas a primeira de muitas controvérsias e inconsistências em torno do SQL que veremos, mas a maioria das pessoas saberá o que você quer dizer, independentemente de como você o diga. Há algum debate sobre se o SQL é ou não uma linguagem de programação. Não é uma linguagem de propósito geral como C ou Python são. SQL sem banco de dados e dados em tabelas é apenas um arquivo de texto. SQL não pode construir um site, mas é poderoso para trabalhar com dados em bancos de dados. Em um nível prático, o que mais importa é que o SQL pode ajudá-lo a realizar o trabalho de análise de dados.

A IBM foi a primeira a desenvolver bancos de dados SQL, a partir do modelo relacional inventado por Edgar Codd na década de 1960. O modelo relacional foi uma descrição teórica para o gerenciamento de dados usando relacionamentos. Ao criar os primeiros bancos de dados, a IBM ajudou a avançar a teoria, mas também teve considerações comerciais, assim como Oracle, Microsoft e todas as outras empresas que comercializaram um banco de dados desde então. Desde o início, houve tensão entre a teoria do computador e a realidade comercial. O SQL tornou-se um padrão da International Organization for Standards (ISO) em 1987 e um padrão do American National Standards Institute (ANSI) em 1986. Embora todos os principais bancos de dados começem com esses padrões em sua implementação do SQL, muitos têm variações e funções que facilitam a vida dos usuários desses bancos de dados. Isso tem o custo de tornar o SQL mais difícil de mover entre bancos de dados sem algumas modificações.

SQL é usado para acessar, manipular e recuperar dados de objetos em um banco de dados. Os bancos de dados podem ter um ou mais *esquemas*, que fornecem a organização e estrutura e contém outros objetos. Dentro de um esquema, os objetos mais comumente usados na análise de dados são tabelas, visualizações e funções. As tabelas contêm campos, que contêm os dados. As tabelas 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. Os índices geralmente são definidos por um administrador de banco de dados. As visualizações são essencialmente consultas armazenadas que podem ser referenciadas da mesma forma que uma tabela. As funções permitem que conjuntos de cálculos ou procedimentos comumente usados sejam armazenados e facilmente referenciados em consultas. Eles geralmente são criados por um administrador de banco de dados ou DBA. **Figura 1-1** fornece uma visão geral da organização dos bancos de dados.



*Figura 1-1. Visão geral da organização do banco de dados e objetos em um banco de dados*

Para se comunicar com bancos de dados, o SQL tem quatro sublinguagens para lidar com diferentes tarefas, e elas são geralmente padrão em todos os tipos de banco de dados. A maioria das pessoas que trabalha com análise de dados não precisa lembrar os nomes dessas sublinguagens diariamente, mas elas podem surgir em conversas com administradores de banco de dados ou engenheiros de dados, então vou apresentá-las brevemente. Todos os comandos funcionam juntos de forma fluida e alguns podem coexistir na mesma instrução SQL.

*DQL*, ou *linguagem de consulta de dados*, é o assunto principal deste livro. Ele é usado para *consultar* dados, que você pode pensar como usar código para fazer perguntas a um banco de dados. Os comandos DQL incluem *SELECT*, que será familiar para usuários anteriores de SQL, mas o acrônimo DQL não é usado com frequência em minha experiência. As consultas SQL podem ser tão curtas quanto uma única linha ou abranger muitas dezenas de linhas. As consultas SQL podem acessar uma única tabela (ou exibição), podem combinar dados de várias tabelas por meio do uso de junções e também podem consultar vários esquemas no mesmo banco de dados. As consultas SQL geralmente não podem consultar bancos de dados, mas, em alguns casos, configurações de rede inteligentes ou software adicional podem ser usados para recuperar dados de várias fontes, até mesmo bancos de dados de diferentes tipos. As consultas SQL são autocontidas e, além das tabelas, não fazem referência a variáveis ou saídas de etapas anteriores não contidas na consulta, ao contrário das linguagens de script.

*DDL*, ou *linguagem de definição de dados*, é usada para criar e modificar tabelas, visualizações, usuários e outros objetos no banco de dados. Afeta a estrutura, mas não o conteúdo. Existem três comandos comuns: *CREATE*, *ALTER* e *DROP*.

*CREATE* é usado para criar novos objetos. *ALTER* altera a estrutura de um objeto, por exemplo, adicionando uma coluna a uma tabela. *DROP* exclui todo o objeto e sua estrutura. Você pode ouvir DBAs e engenheiros de dados falarem sobre como trabalhar com DDLs – isso é apenas um atalho para os arquivos ou pedaços de código que criam, alteram ou descartam. Um exemplo de como o DDL é usado no contexto de análise é o código para criar tabelas temporárias.

*DCL*, ou *linguagem de controle de dados*, é usado para controle de acesso. Os comandos incluem *GRANT* e *REVOKE*, que dão permissão e removem permissão, respectivamente. Em um contexto de análise, *GRANT* pode ser necessário para permitir que um colega consulte uma tabela que você criou. Você também pode encontrar esse comando quando alguém lhe disser que uma tabela existe no banco de dados, mas você não podevê-la - as permissões podem precisar ser *GRANTED* para seu usuário.

*DML*, ou *linguagem de manipulação de dados*, é usada para agir sobre os próprios dados. Os comandos são *INSERT*, *UPDATE* e *DELETE*. *INSERT* adiciona novos registros e é essencialmente a etapa de “carregar” em extrair, transformar, carregar (ETL). *UPDATE* altera valores em um campo e *DELETE* remove linhas. Você encontrará esses comandos se tiver algum tipo de tabela autogerenciada — tabelas temporárias, tabelas sandbox — ou se estiver na função de proprietário e analisador do banco de dados.

Essas quatro sublinguagens estão presentes em todos os principais bancos de dados. Neste livro, focarei principalmente em DQL. Abordaremos alguns comandos DDL e DML no Capítulo 8 e você também verá alguns exemplos no [site do GitHub para o livro](#), onde eles são usados para criar e preencher os dados usados nos exemplos. Graças a esse conjunto comum de comandos, o código SQL escrito para qualquer banco de dados parecerá familiar para qualquer pessoa acostumada a trabalhar com SQL. No entanto, ler SQL de outro banco de dados pode parecer um pouco como ouvir alguém que fala a mesma língua que você, mas vem de outra parte do país ou do mundo. A estrutura básica da linguagem é a mesma, mas a gíria é diferente e algumas palavras têm significados completamente diferentes. Variações no SQL de banco de dados para banco de dados são frequentemente chamadas *dialetos*, e os usuários de banco de dados farão referência a Oracle SQL, MSSQL ou outros dialetos.

Ainda assim, uma vez que você conhece SQL, você pode trabalhar com diferentes tipos de banco de dados, desde que preste atenção a detalhes como o tratamento de nulos, datas e timestamps; a divisão de inteiros; e sensibilidade a maiúsculas e minúsculas.

Este livro usa o PostgreSQL, ou Postgres, para os exemplos, embora eu tente apontar onde o código seria significativamente diferente em outros tipos de banco 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 boas razões para usar o SQL para análise de dados, desde poder de computação até sua onipresença em ferramentas de análise de dados e sua flexibilidade.

Talvez a melhor razão para usar o SQL seja que muitos dos dados do mundo já estão em bancos de dados. É provável que sua própria organização tenha um ou mais bancos de dados. Mesmo que os dados ainda não estejam em um banco de dados, carregá-los em um pode valer a pena para aproveitar as vantagens de armazenamento e computação, especialmente quando comparado a alternativas como planilhas. O poder da computação explodiu nos últimos anos, e os data warehouses e a infraestrutura de dados evoluíram para aproveitá-lo. Alguns bancos de dados em nuvem mais recentes permitem que grandes quantidades de dados sejam consultadas na memória, acelerando ainda mais as coisas. Os dias de espera de minutos ou horas para que os resultados da consulta retornem podem ter acabado, embora os analistas possam apenas escrever consultas mais complexas em resposta.

SQL é o padrão de fato para interagir com bancos de dados e recuperar dados deles. Uma ampla variedade de softwares populares se conecta a bancos de dados com SQL, de planilhas a BI e ferramentas de visualização e linguagens de codificação como Python e R (discutidos na próxima seção). Devido aos recursos computacionais disponíveis, realizar o máximo de manipulação e agregação de dados possível no banco de dados geralmente traz vantagens no downstream. Discutiremos em profundidade as estratégias para construir conjuntos de dados complexos para ferramentas downstream no [Capítulo 8](#).

Os blocos de construção básicos do SQL podem ser combinados de inúmeras maneiras. Começando com um número relativamente pequeno de blocos de construção — a sintaxe — o SQL pode realizar uma ampla gama de tarefas. O SQL pode ser desenvolvido de forma iterativa e é fácil revisar os resultados à medida que avança. Pode não ser uma linguagem de programação completa, mas pode fazer muito, desde transformar dados até realizar cálculos complexos e responder a perguntas.

Por último, o SQL é relativamente fácil de aprender, com uma quantidade finita de sintaxe. Você pode aprender as palavras-chave básicas e a estrutura rapidamente e, em seguida, aprimorar seu ofício ao longo do tempo, trabalhando com conjuntos de dados variados. As aplicações do SQL são virtualmente infinitas, quando você leva em conta a variedade de conjuntos de dados no mundo e as possíveis perguntas que podem ser feitas aos dados. O SQL é ensinado em muitas universidades, e muitas pessoas adquirem algumas habilidades no trabalho. Mesmo os funcionários que ainda não possuem habilidades em SQL podem ser treinados, e a curva de aprendizado pode ser mais fácil do que para outras linguagens de programação. Isso torna o armazenamento de dados para análise em bancos de dados relacionais uma escolha lógica para as organizações.

## SQL Versus R ou Python

Embora o SQL seja uma linguagem popular para análise de dados, não é a única opção. R e Python estão entre as mais populares das outras linguagens usadas para análise de dados. R é uma linguagem estatística e gráfica, enquanto Python é uma linguagem de programação de uso geral que tem pontos fortes no trabalho com dados. Ambos são de código aberto, podem ser instalados 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 está além do escopo deste livro, mas há muitas discussões online sobre as vantagens relativas de cada um. Aqui vou considerá-los juntos como alternativas de linguagem de codificação ao SQL.

Uma grande diferença entre o SQL e outras linguagens de codificação é onde o código é executado e, portanto, quanto poder de computação está disponível. O SQL sempre roda em um servidor de banco de dados, aproveitando todos os seus recursos computacionais. Para fazer análises, R e Python geralmente são executados localmente em sua máquina, portanto, os recursos de computação são limitados pelo que estiver disponível localmente. Existem, é claro, muitas exceções: bancos de dados podem ser executados em laptops e R e Python podem ser executados em servidores com mais recursos. Quando você está executando qualquer coisa que não seja a análise mais simples em grandes conjuntos de dados, enviar o trabalho para um servidor de banco de dados com mais recursos é uma boa opção. Como os bancos de dados geralmente 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 está em como os dados são armazenados e organizados. Os bancos de dados relacionais sempre organizam os dados em linhas e colunas dentro de tabelas, portanto, o SQL assume essa estrutura para cada consulta. R e Python têm uma variedade maior de maneiras de armazenar dados, incluindo variáveis, listas e dicionários, entre outras opções. Estes oferecem mais flexibilidade, mas ao custo de uma curva de aprendizado mais acentuada. Para facilitar a análise de dados, o R possui data frames, que são semelhantes às tabelas de banco de dados e organizam os dados em linhas e colunas. O pacote pandas disponibiliza DataFrames em Python. Mesmo quando outras opções estão disponíveis, a estrutura da tabela permanece valiosa para análise.

O loop é outra grande diferença entre o SQL e a maioria das outras linguagens de programação de computador. Um *loop* é uma instrução ou um conjunto de instruções que se repete até que uma condição específica seja atendida. As agregações SQL fazem um loop implícito no conjunto de dados, sem nenhum código adicional. Veremos mais tarde como a falta de capacidade de fazer loop sobre campos pode resultar em instruções SQL longas ao dinamizar ou não dinamizar dados. Embora uma discussão mais profunda esteja além do escopo deste livro, alguns fornecedores criaram extensões para SQL, como PL/SQL no Oracle e T-SQL no Microsoft SQL Server, que permitem funcionalidades como looping.

Uma desvantagem do SQL é que seus dados devem estar em um banco de dados,<sup>1</sup> enquanto R e 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. Na outra direção, pacotes como dbplyr para R e SQLAlchemy para Python permitem que programas escritos nessas linguagens se conectem a bancos de dados, executem consultas SQL e usem os resultados em etapas de processamento adicionais. Nesse sentido, R ou Python podem ser complementares ao SQL.

R e Python têm funções estatísticas sofisticadas que são incorporadas ou estão disponíveis em pacotes. Embora o SQL tenha, por exemplo, funções para calcular média e desvio padrão, os cálculos de p-values e significância estatística que são necessários na análise de experimentos (discutidos no [Capítulo 7](#)) não podem ser realizados apenas com SQL. Além das estatísticas sofisticadas, o aprendizado de máquina é outra área que é melhor abordada com uma dessas outras linguagens de codificação.

Ao decidir usar SQL, R ou Python para uma análise, considere:

- Onde os dados estão localizados — em um banco de dados, um arquivo, um site?
- Qual é o volume de dados?
- Para onde estão indo os dados - em um relatório, uma visualização, uma análise estatística?
- Ele precisará ser atualizado ou atualizado com novos dados? Com que frequência?
- O que sua equipe ou organização usa e qual a importância de estar em conformidade com os padrões existentes?

Não há escassez de debate sobre quais linguagens e ferramentas são melhores para fazer análise de dados ou ciência de dados. Tal como acontece com muitas coisas, muitas vezes há mais de uma maneira de realizar uma análise. As linguagens de programação evoluem e mudam em popularidade, e temos sorte de viver e trabalhar em uma época com tantas opções boas. O SQL existe há muito tempo e provavelmente permanecerá popular nos próximos anos. O objetivo final é usar a melhor ferramenta disponível para o trabalho. Este livro irá ajudá-lo a tirar o máximo proveito do SQL para análise de dados, independentemente do que mais está em seu kit de ferramentas.

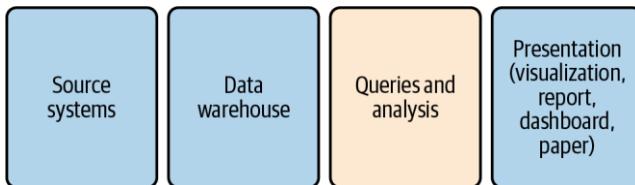
## SQL como parte do fluxo de trabalho de análise de dados

Agora que expliquei o que é o SQL, discuti alguns de seus benefícios e o comparei com outras linguagens, vamos discutir onde o SQL se encaixa no processo de análise de dados. O trabalho de análise sempre começa com uma pergunta, que pode ser sobre quantos novos clientes foram adquiridos, como as vendas estão tendendo ou por que alguns usuários permanecem por muito tempo enquanto outros tentam um serviço e nunca mais retornam. Uma vez formulada a pergunta, consideramos de onde os dados se originaram, onde os dados são armazenados, o

---

<sup>1</sup> Existem algumas tecnologias mais recentes que permitem consultas SQL em dados armazenados em fontes não relacionais.

plano de análise e como os resultados serão apresentados ao público. [Figura 1-2](#) mostra as etapas do processo. Consultas e análises são o foco deste livro, embora eu discuta as outras etapas brevemente para colocar as consultas e o estágio de análise em um contexto mais amplo.



*Figura 1-2. Etapas no 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 manualmente por pessoas, como quando alguém preenche um formulário ou faz anotações durante uma consulta médica. Os dados também podem ser gerados por máquina, como quando um banco de dados de aplicativos registra uma compra, um sistema de transmissão de eventos registra um clique em um site ou uma ferramenta de gerenciamento de marketing registra um e-mail aberto. Os sistemas de origem podem gerar muitos tipos e formatos de dados diferentes, e [Capítulo 2](#) os discutirá e 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 foco mais restrito; e *data lake*, um termo que pode significar que os dados residem em um sistema de armazenamento de arquivos ou que são armazenados em um banco de dados, mas sem o grau de transformação de dados comum em data warehouses. Os data warehouses variam de pequenos e simples a enormes e caros. Um banco de dados rodando em um laptop será suficiente para você acompanhar os exemplos deste livro. O que importa é ter os dados necessários para realizar uma análise juntos em um só lugar.



Normalmente, uma pessoa ou equipe é responsável por obter dados no data warehouse. Esse processo é chamado *ETL*, ou extrair, transformar, carregar. Extract extrai os dados do sistema de origem. Transform opcionalmente altera a estrutura dos dados, executa a limpeza da qualidade dos dados ou agrupa os dados. Load coloca os dados no banco de dados. Esse processo também pode ser chamado *ELT*, para extrair, carregar, transformar — a diferença é que, em vez de as transformações serem feitas antes do carregamento dos dados, todos os dados são carregados e, em seguida, as transformações são executadas, geralmente usando SQL. Você também pode ouvir os termos *origem* e *destino* no contexto de ETL. A origem é a origem dos dados e o destino é o destino, ou seja, o banco de dados e as tabelas nele contidas. Mesmo quando o SQL é usado para fazer a transformação, outra linguagem, como Python ou Java, é usada para unir as etapas, coordenar o agendamento e emitir alertas quando algo der errado. Existem vários produtos comerciais, bem como ferramentas de código aberto disponíveis, para que as equipes não precisem criar um sistema ETL inteiramente do zero.

Uma vez que os dados estão em um banco de dados, a próxima etapa é realizar consultas e análises. Nesta etapa, o SQL é aplicado para explorar, perfilar, limpar, moldar e analisar os dados. **Figura 1-3** mostra o fluxo geral do processo. Explorar os dados envolve familiarizar-se com o tópico, onde os dados foram gerados e as tabelas do banco de dados em que estão armazenados. A criação de perfil 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 sinalizadores e manipular valores nulos. A modelagem é o processo de organizar os dados nas linhas e colunas necessárias no conjunto de resultados. Por fim, analisar os dados envolve revisar a saída para tendências, conclusões e insights. Embora esse processo seja mostrado como linear, na prática, muitas vezes é cíclico – por exemplo, quando a modelagem ou análise revela dados que devem ser limpos.

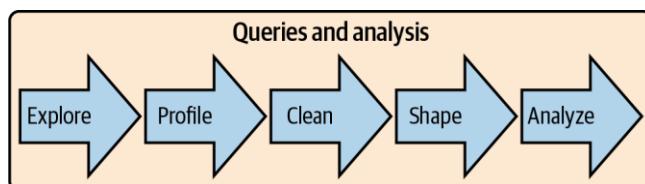


Figura 1-3. Etapas das consultas e da etapa de análise do fluxo

A apresentação dos dados em um formulário de saída final é a última etapa do fluxo de trabalho geral. Os empresários não vão gostar de receber um arquivo de código SQL; eles esperam que você apresente gráficos, tabelas e insights. A comunicação é fundamental para causar impacto com a análise, e para isso precisamos de uma forma de compartilhar os resultados com outras pessoas. Outras vezes, você pode precisar aplicar uma análise estatística mais sofisticada do que é possível no SQL ou pode querer alimentar os dados em um algoritmo de aprendizado de máquina (ML). Felizmente, a maioria das ferramentas de relatório e visualização tem conectores SQL que permitem extrair dados de tabelas inteiras ou consultas SQL pré-escritas. O software estatístico e as linguagens comumente usadas para ML também costumam ter conectores SQL.

Os fluxos de trabalho de análise abrangem várias etapas e geralmente incluem várias ferramentas e tecnologias. As consultas e análises SQL estão no centro de muitas análises e são o que focaremos 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 banco de dados que você provavelmente encontrará em sua jornada de análise.

## Tipos de banco de dados e como trabalhar com eles

Se você estiver trabalhando com SQL, estará trabalhando com bancos de dados. Há uma variedade de tipos de banco de dados - de código aberto a proprietário, armazenamento de linha a armazenamento de coluna. Existem bancos de dados locais e bancos de dados em nuvem, bem como bancos de dados híbridos, nos quais uma organização executa o software de banco de dados na infraestrutura de um fornecedor de nuvem. Há também vários armazenamentos de dados que não são bancos de dados, mas podem ser consultados com SQL.

Os bancos de dados não são todos iguais; cada tipo de banco de dados tem seus pontos fortes e fracos quando se trata de trabalho de análise. Ao contrário das ferramentas usadas em outras partes do fluxo de trabalho de análise, você pode não ter muito a dizer sobre qual tecnologia de banco de dados é usada em sua organização. Conhecer os detalhes do banco de dados que você possui o ajudará a trabalhar com mais eficiência e aproveitar todas as funções especiais do SQL que ele oferece. A familiaridade com outros tipos de bancos de dados o ajudará se você estiver trabalhando em um projeto para criar ou migrar 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 armazém em nuvem por motivos semelhantes.

Bancos de dados e armazenamentos de dados têm sido uma área dinâmica de desenvolvimento de tecnologia desde que foram introduzidos. Algumas tendências desde a virada do século 21 impulsionaram a tecnologia de maneiras que são realmente empolgantes para os profissionais de dados de hoje. Primeiro, os volumes de dados aumentaram incrivelmente com a Internet, os dispositivos móveis e a Internet das Coisas (IoT). Em 2020 , a IDC previu que a quantidade de dados armazenados globalmente crescerá para 175 zettabytes até 2025. Essa escala de dados é difícil de se pensar, e nem todos serão armazenados em bancos de dados para análise. Hoje em dia, não é incomum que as empresas tenham dados na escala de terabytes e petabytes, uma escala que seria impossível de processar com a tecnologia da década de 1990 e anteriores.

Em segundo lugar, as reduções nos custos de armazenamento de dados e computação, juntamente com o advento da nuvem, tornaram mais barato e fácil para as organizações coletar e armazenar essas enormes quantidades de dados. A memória do computador ficou mais barata, o que significa que grandes quantidades de dados podem ser carregadas na memória, cálculos realizados e resultados retornados, tudo sem ler e gravar em disco, aumentando muito a velocidade. Terceiro, a computação distribuída permitiu a divisão de cargas de trabalho em muitas máquinas. Isso permite que uma quantidade grande e ajustável de computação seja direcionada para tarefas de dados complexas.

Bancos de dados e armazenamentos de dados combinaram essas tendências tecnológicas de várias maneiras diferentes para otimizar para tipos específicos de tarefas. Existem duas grandes categorias de bancos de dados que são relevantes para o trabalho de análise: armazenamento de linha e armazenamento de coluna. Na próxima seção, vou apresentá-los, discutir o que os torna semelhantes e diferentes uns dos outros e falar sobre o que tudo isso significa no que diz respeito a fazer análises com dados armazenados neles. Finalmente, apresentarei alguns tipos adicionais de infra-estrutura de dados além dos bancos de dados que você pode encontrar.

## Bancos de dados de armazenamento de linhas

*Bancos de dados de armazenamento de linha* — também chamados *transacionais* — são projetados para serem eficientes no processamento de transações: *INSERTs*, *UPDATEs* e *DELETEs*. Os bancos de dados de armazenamento de linha de código aberto populares incluem MySQL e Postgres. Do lado comercial, Microsoft SQL Server, Oracle e Teradata são amplamente utilizados. Embora não sejam realmente otimizados para análise, por vários anos os bancos de dados de armazenamento em linha foram a única opção para empresas que construíam data warehouses. Por meio de ajustes cuidadosos e design de esquema, esses bancos de dados podem ser usados para análises. Eles também são atraentes devido ao baixo custo das opções de código aberto e porque são familiares aos administradores de banco de dados que os mantêm. Muitas organizações replicam seu banco de dados de produção na mesma tecnologia como um primeiro passo para construir a infraestrutura de dados. Por todas essas razões, analistas de dados e cientistas de dados provavelmente trabalharão com dados em um banco de dados de armazenamento de linha em algum momento de sua carreira.

Pensamos em uma tabela como linhas e colunas, mas os dados precisam ser serializados para armazenamento. Uma consulta pesquisa um disco rígido pelos dados necessários. Os discos rígidos são organizados em uma série de blocos de tamanho fixo. A verificação do disco rígido consome tempo e recursos, portanto, é importante minimizar a quantidade de disco que precisa ser verificada para retornar os resultados da consulta. Bancos de dados de armazenamento de linha abordam esse problema serializando dados em uma linha. [Figura 1-4](#) mostra um exemplo de armazenamento de dados por linha. Ao consultar, a linha inteira é lida na memória. Essa abordagem é rápida ao fazer atualizações de linha, mas é mais lenta ao fazer cálculos em muitas linhas se apenas algumas colunas forem necessárias.

<b>id</b>	<b>sku</b>	<b>type</b>	<b>color</b>	<b>size</b>	<b>price</b>
1	123	tshirt	black	S	19.99
2	124	shorts	green	M	24.99

Figura 1-4. Armazenamento por linha, no qual cada linha é armazenada em disco

Para reduzir a largura das tabelas, os bancos de dados de armazenamento de linha geralmente são modelados na *terceira forma normal*, que é uma abordagem de projeto de banco de dados que busca armazenar cada informação apenas uma vez, para evitar duplicações e inconsistências. Isso é eficiente para o processamento de transações, mas geralmente leva a um grande número de tabelas no banco de dados, cada uma com apenas algumas colunas. Para analisar esses dados, muitas junções podem ser necessárias e pode ser difícil para não desenvolvedores entender como todas as tabelas se relacionam e onde um determinado dado está armazenado. Ao fazer análises, o objetivo geralmente é a desnормalização, ou reunir todos os dados em um só lugar.

As tabelas geralmente têm uma *chave primária* que impõe exclusividade — em outras palavras, impede que o banco de dados crie mais de um registro para a mesma coisa. As tabelas geralmente têm uma **id** que é um inteiro de incremento automático, onde cada novo registro obtém o próximo inteiro após o último inserido ou um valor alfanumérico que é criado por um gerador de chave primária. Também deve haver um conjunto de colunas que juntas tornem a linha exclusiva; essa combinação de campos é chamada de *chave composta* ou, às vezes, *de chave comercial*. Por exemplo, em uma tabela de pessoas, as colunas **first\_name**, **last\_name** e **birthdate** juntas podem tornar a linha exclusiva. **Social\_security\_id** também seria um identificador exclusivo, além da coluna **person\_id**.

As tabelas também têm, opcionalmente, índices que tornam a pesquisa de registros específicos mais rápida e tornam as junções envolvendo essas colunas mais rápidas. Os índices armazenam os valores no campo ou campos indexados como partes únicas de dados junto com um ponteiro de linha e, como os índices são menores do que a tabela inteira, são mais rápidos de verificar. Normalmente, a chave primária é indexada, mas outros campos ou grupos de campos também podem ser indexados. Ao trabalhar com bancos de dados de armazenamento de linha, é útil saber quais campos nas tabelas que você usa têm índices. As junções comuns podem ser aceleradas adicionando índices, portanto, vale a pena investigar se as consultas de análise demoram muito para serem executadas. Os índices não vêm de graça: eles ocupam espaço de armazenamento e retardam o carregamento, pois novos valores precisam ser adicionados a cada inserção. Os DBAs podem não indexar tudo o que pode ser útil para análise. Além de relatórios, o trabalho de análise também pode não ser rotineiro o suficiente para se preocupar com a otimização de índices. As consultas exploratórias e complexas geralmente usam padrões de junção complexos, e podemos descartar uma abordagem quando descobrimos uma nova maneira de resolver um problema.

**Modelagem de esquema em estrela** foi desenvolvido em parte para tornar os bancos de dados de armazenamento de linha mais amigáveis para cargas de trabalho analíticas.

Os fundamentos são apresentados no livro *The Data Warehouse Toolkit*,<sup>2</sup> que defende a modelagem dos dados como uma série de tabelas de fatos e dimensões. As tabelas de fatos representam eventos, como transações em lojas de varejo. As dimensões contêm descritores como nome do cliente e tipo de produto. Como os dados nem sempre se encaixam perfeitamente nas categorias de fato e dimensão, há uma extensão chamada **foco esquema** em que algumas dimensões têm dimensões próprias.

## Bancos de dados de armazenamento de colunas

Bancos de dados *Armazenamento de colunas* decolaram no início do século 21, embora sua história teórica volte até os bancos de dados de armazenamento de linha. Bancos de dados de armazenamento de coluna armazenam os valores de uma coluna juntos, em vez de armazenar os valores de uma linha juntos. Esse design é otimizado para consultas que lêem muitos registros, mas não necessariamente todas as colunas. Os bancos de dados de armazenamento de colunas populares incluem Amazon Redshift, Snowflake e Vertica.

Os bancos de dados de armazenamento em coluna são eficientes no armazenamento de grandes volumes de dados graças à compactação. Valores ausentes e valores repetidos podem ser representados por valores de marcador muito pequenos em vez do valor total. Por exemplo, em vez de armazenar “Reino Unido” milhares ou milhões de vezes, um banco de dados de armazenamento de colunas armazenará um valor substituto que ocupa muito pouco espaço de armazenamento, juntamente com uma pesquisa que armazena o valor “Reino Unido” completo. Os bancos de dados de armazenamento de coluna também compactam dados aproveitando as repetições de valores em dados classificados. Por exemplo, o banco de dados pode armazenar o fato de que o valor do marcador para “Reino Unido” é repetido 100 vezes, e isso ocupa ainda menos espaço do que armazenar esse marcador 100 vezes.

Os bancos de dados de armazenamento de coluna não impõem chaves primárias e não possuem índices. Valores repetidos não são problemáticos, graças à compressão. Como resultado, os esquemas podem ser adaptados para consultas de análise, com todos os dados juntos em um local, em vez de estarem em várias tabelas que precisam ser unidas. No entanto, dados duplicados podem se infiltrar facilmente sem chaves primárias, portanto, é importante entender a origem dos dados e a verificação de qualidade.

Atualizações e exclusões são caras na maioria dos bancos de dados de armazenamento de colunas, pois os dados de uma única linha são distribuídos em vez de armazenados juntos. Para tabelas muito grandes, pode existir uma política somente de gravação, portanto, também precisamos saber algo sobre como os dados são gerados para descobrir quais registros usar. Os dados também podem ser mais lentos para ler, pois precisam ser descompactados antes que os cálculos sejam aplicados.

Os bancos de dados de armazenamento de coluna geralmente são o padrão-ouro para o trabalho de análise rápido. Eles usam SQL padrão (com algumas variações específicas do fornecedor) e, de muitas maneiras, trabalhar com eles não é diferente de trabalhar com um banco de dados de armazenamento de linha em termos das consultas que você escreve.

---

<sup>2</sup> Ralph Kimball e Margy Ross, *The Data Warehouse Toolkit*, 3<sup>a</sup> ed. (Indianápolis: Wiley, 2013).

O tamanho dos dados é importante, assim como a computação e armazenamento recursos que foram alocados para o banco de dados. Já vi agregações passarem por milhões e bilhões de registros em segundos. Isso faz maravilhas para a produtividade.



Existem alguns truques para estar ciente. Como certos tipos de compactação dependem da classificação, conhecer os campos nos quais a tabela é classificada e usá-los para filtrar consultas melhora o desempenho. A junção de tabelas pode ser lenta se ambas as tabelas forem grandes.

No final das contas, alguns bancos de dados serão mais fáceis ou mais rápidos de trabalhar, mas não há nada inherente ao tipo de banco de dados que o impeça de realizar qualquer análise neste livro. Como em todas as coisas, usar uma ferramenta que seja adequadamente poderosa para o volume de dados e a complexidade da tarefa permitirá que você se concentre na criação de análises significativas.

## Outros tipos de infraestrutura de Dados

Os bancos de dados não são a única maneira de armazenar dados, e há uma variedade crescente de opções para armazenar dados necessários para análise e alimentação de aplicativos. Sistemas de armazenamento de arquivos, às vezes chamados *de data lakes*, são provavelmente a principal alternativa aos armazéns de banco de dados. Bancos de dados NoSQL e armazenamentos de dados baseados em pesquisa são sistemas alternativos de armazenamento de dados que oferecem baixa latência para desenvolvimento de aplicativos e pesquisa de arquivos de log. Embora normalmente não façam parte do processo de análise, eles são cada vez mais parte da infraestrutura de dados das organizações, portanto, também os apresentarei brevemente nesta seção. Uma tendência interessante a ser apontada é que, embora esses tipos mais novos de infra-estrutura inicialmente visassem romper com os limites dos bancos de dados SQL, muitos acabaram implementando algum tipo de interface SQL para consultar os dados.

O Hadoop, também conhecido como HDFS (para “sistema de arquivos distribuído do Hadoop”), é um sistema de armazenamento de arquivos de código aberto que aproveita o custo cada vez menor de armazenamento de dados e poder de computação, bem como sistemas distribuídos. Os arquivos são divididos em blocos e o Hadoop os distribui em um sistema de arquivos armazenado em nós, ou computadores, em um cluster. O código para executar as operações é enviado aos nós e eles processam os dados em paralelo. O grande avanço do Hadoop foi permitir que grandes quantidades de dados fossem armazenadas de forma barata. Muitas grandes empresas de internet, com grandes quantidades de dados muitas vezes não estruturados, acharam isso uma vantagem sobre as limitações de custo e armazenamento dos bancos de dados tradicionais. As primeiras versões do Hadoop tinham duas grandes desvantagens: habilidades de codificação especializadas eram necessárias para recuperar e processar dados, pois não eram compatíveis com SQL, e o tempo de execução dos programas geralmente era bastante longo. Desde então, o Hadoop amadureceu e várias ferramentas foram desenvolvidas que permitem acesso SQL ou semelhante a SQL aos dados e aceleram os tempos de consulta.

Outros produtos comerciais e de código aberto foram introduzidos nos últimos anos para aproveitar o armazenamento de dados barato e o processamento de dados rápido, geralmente na memória, enquanto oferecem capacidade de consulta SQL. Alguns deles até permitem que o analista escreva uma única consulta que retorne dados de várias fontes subjacentes. Isso é empolgante para quem trabalha com grandes quantidades de dados, e é uma validação de que o SQL veio para ficar.

NoSQL é uma tecnologia que permite a modelagem de dados que não é estritamente relacional. Ele permite armazenamento e recuperação de latência muito baixa, essenciais em muitos aplicativos online. A classe inclui armazenamento de pares de valores-chave e bancos de dados de gráficos, que armazenam em um formato de borda de nó, e armazenamentos de documentos. Exemplos desses armazenamentos de dados que você pode ouvir em sua organização são Cassandra, Couchbase, DynamoDB, Memcached, Giraph e Neo4j. No início, o NoSQL foi comercializado como tornando o SQL obsoleto, mas o acrônimo foi mais recentemente comercializado como “não apenas SQL”. Para fins de análise, o uso de dados armazenados em um armazenamento de chave-valor NoSQL para análise geralmente requer a movimentação para um data warehouse SQL mais tradicional, pois o NoSQL não é otimizado para consultar muitos registros de uma só vez. Os bancos de dados gráficos têm aplicativos como análise de rede, e o trabalho de análise pode ser feito diretamente neles com linguagens de consulta especiais. No entanto, o cenário de ferramentas está sempre evoluindo e talvez um dia possamos analisar esses dados também com SQL.

Os armazenamentos de dados baseados em pesquisa incluem Elasticsearch e Splunk. O Elasticsearch e o Splunk são frequentemente usados para analisar dados gerados por máquina, como logs. Essas e outras tecnologias semelhantes têm linguagens de consulta não SQL, mas se você conhece SQL, geralmente pode entendê-las. Reconhecendo como as habilidades SQL são comuns, alguns armazenamentos de dados, 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 adequadas aos tipos de tarefas de análise que este livro aborda. Como expliquei às pessoas ao longo dos anos, eles são ótimos para encontrar agulhas em palheiços. Eles não são tão bons em medir o palheiro em si.

Independentemente do tipo de banco de dados ou outra tecnologia de armazenamento de dados, a tendência é clara: mesmo que os volumes de dados cresçam e os casos de uso se tornem mais complexos, o SQL ainda é a ferramenta padrão para acessar dados. Sua grande base de usuários existente, curva de aprendizado acessível e poder para tarefas analíticas significam que mesmo as tecnologias que tentam se afastar do SQL voltam e o acomodam.

## Conclusão

A análise de dados é uma disciplina empolgante com uma variedade de aplicativos para empresas e outras organizações. SQL tem muitos benefícios para trabalhar com dados, particularmente quaisquer dados armazenados em um banco de dados.

Consultar e analisar dados faz parte do fluxo de trabalho de análise mais amplo e há vários tipos de armazenamentos de dados com os quais um cientista de dados pode esperar trabalhar. Agora que estabelecemos as bases para análise, SQL e armazenamentos de dados, o restante do livro abordará o uso do SQL para análise em profundidade. O Capítulo 2 se concentra na preparação de dados, começando com uma introdução aos tipos de dados e, em seguida, avançando para a criação de perfil, limpeza e modelagem de dados. Os capítulos 3 a 7 apresentam aplicações de análise de dados, com foco na análise de séries temporais, análise de coorte, análise de texto, detecção de anomalias e análise de experimentos. O Capítulo 8 aborda técnicas para desenvolver conjuntos de dados complexos para análise posterior em outras ferramentas. Por fim, Capítulo 9 conclui com reflexões sobre como os tipos de análise podem ser combinados para novos insights e lista alguns recursos adicionais para apoiar sua jornada analítica.

## CAPÍTULO 2

# Preparando dados para análise

As estimativas de quanto tempo os cientistas de dados gastam preparando seus dados variam, mas é seguro dizer que essa etapa ocupa uma parte significativa do tempo gasto trabalhando com dados. Em 2014, o *New York Times* informou que os cientistas de dados gastam de 50% a 80% de seu tempo limpando e organizando seus dados. Uma [pesquisa de 2016 da CrowdFlower](#) descobriram que os cientistas de dados gastam 60% de seu tempo limpando e organizando dados para prepará-los para análise ou trabalho de modelagem. A preparação de dados é uma tarefa tão comum que surgiram termos para descrevê-la, como processamento de dados, manipulação de dados e preparação de dados. (“Mung” é um acrônimo para Mash Until No Good, o que certamente já fiz de vez em quando.) Todo esse trabalho de preparação de dados é apenas um trabalho irracional ou é uma parte importante do processo?

A preparação de dados é mais fácil quando um conjunto de dados possui um *dicionário*, um documento ou repositório que possui descrições claras dos campos, valores possíveis, como os dados foram coletados e como eles se relacionam com outros dados. Infelizmente, este não é frequentemente o caso. A documentação geralmente não é priorizada, mesmo por pessoas que veem seu valor, ou fica desatualizada à medida que novos campos e tabelas são adicionados ou a forma como os dados são preenchidos muda. A criação de perfil de dados cria muitos dos elementos de um dicionário de dados, portanto, se sua organização já possui um dicionário de dados, este é um bom momento para usá-lo e contribuir com ele. Se nenhum dicionário de dados existir atualmente, considere iniciar um! Este é um dos presentes mais valiosos que você pode dar à sua equipe e ao seu futuro. Um dicionário de dados atualizado permite que você acelere o processo de criação de perfil de dados com base na criação de perfil que já foi feita, em vez de replicá-la. Isso também melhora a qualidade dos resultados de sua análise, pois você pode verificar se usou os campos corretamente e aplicou os filtros apropriados.

Mesmo quando existe um dicionário de dados, você provavelmente ainda precisará fazer o trabalho de preparação de dados como parte da análise. Neste capítulo, começarei com uma revisão dos tipos de dados que você provavelmente encontrará. Isso é seguido por uma revisão da estrutura de consulta SQL.

A seguir, falarei sobre o perfil dos dados como forma de conhecer seu conteúdo e verificar a qualidade dos dados. Em seguida, falarei sobre algumas técnicas de modelagem de dados que retornarão as colunas e linhas necessárias para análise posterior. Por fim, apresentarei algumas ferramentas úteis para limpar dados para lidar com quaisquer problemas de qualidade.

## Tipos de dados

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

## Tipos de dados de banco de dados

Todos os campos nas tabelas do banco de dados têm tipos de dados definidos. A maioria dos bancos de dados tem uma boa documentação sobre os tipos que eles suportam, e este é um bom recurso para qualquer detalhe necessário além do que é apresentado aqui. Você não precisa necessariamente ser um especialista nas nuances dos tipos de dados para ser bom em análise, mas mais adiante neste livro encontraremos situações em que considerar o tipo de dados é importante, portanto, esta seção abordará o básico. Os principais tipos de dados são strings, numéricos, lógicos e data/hora, conforme resumido na [Tabela 2-1](#). Eles são baseados no Postgres, mas são semelhantes na maioria dos principais tipos de banco de dados.

*Tabela 2-1. Um resumo dos tipos de dados comuns do banco de dados*

tipo	Nome do	Descrição
<b>String</b>	CHAR / VARCHAR	Contém strings. Um CHAR é sempre de tamanho fixo, enquanto um VARCHAR é de tamanho variável, até um tamanho máximo (256 caracteres, por exemplo).
	TEXT / BLOB	Armazena strings mais longas que não cabem em um VARCHAR. Descrições ou texto livre inseridos pelos respondentes da pesquisa podem ser mantidos nesses campos.
<b>Numeric</b>	INT / SMALLINT / BIGINT	Contém números inteiros (números inteiros). Alguns bancos de dados possuem SMALLINT e/ou BIGINT. SMALLINT pode ser usado quando o campo conterá apenas valores com um pequeno número de dígitos. SMALLINT ocupa menos memória do que um INT normal. BIGINT é capaz de armazenar números com mais dígitos que um INT, mas ocupa mais espaço que um INT.
	FLOAT / DOUBLE / DECIMAL	Contém números decimais, às vezes com o número de casas decimais especificado.
<b>Lógico</b>	BOOLEAN	Contém valores de TRUE ou FALSE.
	DATETIME / TIMESTAMP	Contém datas com horas. Normalmente no formato AAAA-MM-DD hh:mi:ss, onde AAAA é o ano de quatro dígitos, MM é o número do mês de dois dígitos, DD é o dia de dois dígitos, hh é a hora de dois dígitos (geralmente 24 horas, ou valores de 0 a 23), mi são os minutos de dois dígitos e ss são os segundos de dois dígitos. Alguns bancos de dados armazenam apenas timestamps sem fuso horário, enquanto outros possuem tipos específicos para timestamps com e sem fuso horário.
	TIME	Retém os tempos.

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 de string podem ser definidos para conter um número fixo ou variável de caracteres. Um campo CHAR pode ser definido para permitir que apenas dois caracteres contenham abreviações de estado dos EUA, por exemplo, enquanto um campo que armazena os nomes completos dos estados precisaria ser um VARCHAR para permitir um número variável de caracteres. Os campos podem ser definidos como TEXT, CLOB (Character Large Object) ou BLOB (Binary Large Object, que pode incluir tipos de dados adicionais, como imagens), dependendo do banco de dados para armazenar strings muito longas, embora muitas vezes ocupem muito de espaço, esses tipos de dados tendem a ser usados com moderação. Quando os dados são carregados, se chegarem strings muito grandes para o tipo de dados definido, elas podem ser truncadas ou rejeitadas completamente. O SQL tem várias funções de string que usaremos para vários propósitos de análise.

Tipos de dados numéricos são todos aqueles que armazenam números, tanto positivos quanto negativos. Funções matemáticas e operadores podem ser aplicados a campos numéricos. Os tipos de dados numéricos incluem os tipos INT, bem como os tipos FLOAT, DOUBLE e DECIMAL que permitem casas decimais. Os tipos de dados inteiros são frequentemente implementados porque usam menos memória do que suas contrapartes decimais. Em alguns bancos de dados, como o Postgres, a divisão de inteiros resulta em um inteiro, em vez de um valor com casas decimais, como seria de esperar. Discutiremos a conversão de tipos de dados numéricos para obter resultados corretos posteriormente neste capítulo.

O tipo de dados lógico é chamado BOOLEAN. Possui valores de TRUE e FALSE e é uma maneira eficiente de armazenar informações onde essas opções são apropriadas. As operações que compararam dois campos retornam um valor BOOLEAN como resultado. Esse tipo de dado é frequentemente usado para criar *flags*, campos que resumem a presença ou ausência de uma propriedade nos dados. Por exemplo, uma tabela que armazena dados de e-mail pode ter um campo BOOLEAN `has_opened`.

Os tipos datetime incluem DATE, TIMESTAMP e TIME. Os dados de data e hora devem ser armazenados em um campo de um desses tipos de banco de dados sempre que possível, pois o SQL possui várias funções úteis que operam neles. Timestamps e datas são muito comuns em bancos de dados e são críticos para muitos tipos de análise, particularmente análise de séries temporais (abordadas no [Capítulo 3](#)) e análise de coorte (abordadas no [Capítulo 4](#)). O [Capítulo 3](#) discutirá a formatação de data e hora, transformações e cálculos.

Outros tipos de dados, como JSON e tipos geográficos, são suportados por alguns, mas não por todos os bancos de dados. Não entrarei em detalhes sobre todos eles aqui, pois geralmente estão além do escopo deste livro. No entanto, eles são um sinal de que o SQL continua a evoluir para lidar com as tarefas de análise emergentes.

Além dos tipos de dados de banco de dados, há várias maneiras conceituais de categorizar os dados. Isso pode ter um impacto tanto em como os dados são armazenados quanto em como pensamos em analisá-los. Discutirei esses tipos de dados categóricos a seguir.

## Estruturado versus não estruturado

Os dados são frequentemente descritos como estruturados ou não estruturados, ou às vezes como semiestruturados. A maioria dos bancos de dados foi projetada para lidar com dados estruturados, onde cada atributo é armazenado em uma coluna e as instâncias de cada entidade são representadas como linhas. Um modelo de dados é criado primeiro e, em seguida, os dados são inseridos de acordo com esse modelo de dados. Por exemplo, uma tabela de endereços pode ter campos para endereço, cidade, estado e código postal. Cada linha conteria o endereço de um cliente específico. Cada campo tem um tipo de dados e permite que apenas dados desse tipo sejam inseridos. Quando dados estruturados são inseridos em uma tabela, cada campo é verificado para garantir que esteja em conformidade com o tipo de dados correto. Dados estruturados são fáceis de consultar com SQL.

*Os dados não estruturados* são o oposto dos dados estruturados. Não há estrutura, modelo de dados ou tipos de dados predeterminados. Dados não estruturados geralmente são “tudo o mais” que não são dados de banco de dados. Documentos, e-mails e páginas da web não são 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, são mais difíceis para bancos de dados relacionais armazenarem eficientemente e para consultas SQL. Como resultado, dados não estruturados são frequentemente armazenados fora de bancos de dados relacionais. Isso permite que os dados sejam carregados rapidamente, mas a falta de validação de dados 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 semi-estruturados* se enquadram entre essas duas categorias. Muitos dados “não estruturados” têm alguma estrutura que podemos usar. Por exemplo, e-mails têm endereços de e-mail de e para, linhas de assunto, corpo de texto e carimbos de data/hora 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 arquivos e armazenados para análise. Por exemplo, arquivos de áudio de música podem ser marcados com artista, nome da música, gênero e duração. Geralmente, as partes estruturadas de dados semi-estruturados podem ser consultadas com SQL, e o SQL pode ser frequentemente usado para analisar ou extrair dados estruturados para consultas adicionais. Veremos algumas aplicações disso na discussão da análise de texto no [Capítulo 5](#).

## Dados quantitativos versus dados qualitativos

*Os dados quantitativos* são numéricos. Ele mede pessoas, coisas e eventos. Os dados quantitativos podem incluir descriptores, como informações do cliente, tipo de produto ou configurações do dispositivo, mas também vêm com informações numéricas, como preço, quantidade ou duração da visita. Contagens, somas, médias ou outras funções numéricas são aplicadas aos dados. Os dados quantitativos geralmente são gerados por máquina hoje em dia, mas não precisam ser. Altura, peso e pressão arterial registrados em um formulário de entrada do paciente em papel são quantitativos, assim como as pontuações do questionário do aluno digitadas em uma planilha por um professor.

*Os dados qualitativos* geralmente são baseados em texto e incluem opiniões, sentimentos e descrições que não são estritamente quantitativas. Os níveis de temperatura e umidade são quantitativos, enquanto descriptores como “quente e úmido” são qualitativos. O preço que um cliente pagou por um produto é quantitativo; se eles gostam ou não, é qualitativo. O feedback da pesquisa, as consultas de suporte ao cliente e as postagens nas mídias sociais são qualitativos. Existem profissões inteiras que lidam com dados qualitativos. Em um contexto de análise de dados, geralmente tentamos quantificar o qualitativo. Uma técnica para isso é extrair palavras-chave ou frases e contar suas ocorrências. Veremos isso com mais detalhes quando nos aprofundarmos na análise de texto no [Capítulo 5](#). Outra técnica é a análise de sentimento, na qual a estrutura da linguagem é utilizada para interpretar o significado das palavras utilizadas, 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 obter insights que seriam difíceis de resumir de outra forma. Houve avanços interessantes no campo do processamento de linguagem natural, ou PNL, embora muito desse trabalho seja feito com ferramentas como o Python.

## Dados de primeiro, segundo e terceiro

Os dados primários são coletados pela própria organização. Isso pode ser feito por meio de logs de servidores, bancos de dados que rastreiam transações e informações de clientes, ou outros sistemas que são construídos e controlados pela organização e geram dados de interesse para análise. Como os sistemas foram criados internamente, geralmente é possível encontrar as pessoas que os construíram e aprender sobre como os dados são gerados. Os analistas de dados também podem influenciar ou ter controle sobre como determinados dados são criados e armazenados, principalmente quando os bugs são responsáveis pela baixa qualidade dos dados.

*Os dados de terceiros* vêm de fornecedores que prestam um serviço ou executam uma função comercial em nome da organização. Geralmente são produtos de software como serviço (SaaS); exemplos comuns são CRM, ferramentas de automação de e-mail e marketing, software que habilita o comércio eletrônico e rastreadores de interação na Web e em dispositivos móveis. Os dados são semelhantes aos dados primários, pois tratam da própria organização, criados por seus funcionários e clientes. No entanto, tanto o código que gera e armazena os dados quanto o modelo de dados são controlados externamente, e o analista de dados normalmente tem pouca influência sobre esses aspectos. Dados de terceiros são cada vez mais importados para o data warehouse de uma organização para análise. Isso pode ser feito com código personalizado ou conectores ETL, ou com fornecedores de SaaS que oferecem integração de dados.



Muitos fornecedores de SaaS fornecem alguns recursos de relatório, portanto, pode surgir a questão de se preocupar em copiar os dados para um data warehouse. O departamento que interage com uma ferramenta pode achar que os relatórios são suficientes, como um departamento de atendimento ao cliente que relata a tempo para resolver problemas e produtividade do agente de dentro de seu software de suporte técnico. Por outro lado, as interações de atendimento ao cliente podem ser uma entrada importante para um modelo de retenção de clientes, o que exigiria a integração desses dados em um armazenamento de dados com dados de vendas e cancelamentos. Aqui está uma boa regra geral ao decidir se importa dados de uma determinada fonte de dados: se os dados criarem valor quando combinados com dados de outros sistemas, importe-os; se não, espere até que haja um caso mais forte antes de fazer o trabalho.

*Os dados de terceiros* podem ser adquiridos ou obtidos de fontes gratuitas, como as publicadas por governos. A menos que os dados tenham sido coletados especificamente em nome da organização, as equipes de dados geralmente têm pouco controle sobre o formato, a frequência e a qualidade dos dados. Esses dados geralmente não têm a granularidade dos dados de primeira e segunda parte. Por exemplo, a maioria das fontes de terceiros não tem dados no nível do usuário e, em vez disso, os dados podem ser combinados com os dados primários no código postal ou no nível da cidade ou em um nível superior. No entanto, dados de terceiros podem ter informações exclusivas e úteis, como padrões de gastos agregados, dados demográficos e tendências de mercado que seriam muito caros ou impossíveis de coletar de outra forma.

## Dados Esparsos

*Dados esparsos* ocorrem quando há uma pequena quantidade de informações em um conjunto maior de informações vazias ou sem importância. Dados esparsos podem mostrar tantos nulos e apenas alguns valores em uma coluna específica. Nulo, diferente de um valor de 0, é a *ausência* de dados; que serão abordados posteriormente na seção sobre limpeza de dados. Dados esparsos podem ocorrer quando os eventos são raros, como erros de software ou compras de produtos na cauda longa de um catálogo de produtos. Também pode ocorrer nos primeiros dias de lançamento de um recurso ou produto, quando apenas testadores ou clientes beta têm acesso. JSON é uma abordagem que foi desenvolvida para lidar com dados esparsos de uma perspectiva de gravação e armazenamento, pois armazena apenas os dados presentes e omite o restante. Isso contrasta com um banco de dados de armazenamento de linha, que precisa armazenar memória para um campo, mesmo que não haja valor nele.

Dados esparsos podem ser problemáticos para análise. Quando os eventos são raros, as tendências não são necessariamente significativas e as correlações são difíceis de distinguir das flutuações do acaso. Vale a pena criar o perfil de seus dados, conforme discutido mais adiante neste capítulo, para entender se e onde seus dados são esparsos.

Algumas opções são agrupar eventos ou itens infreqüentes em categorias mais comuns, excluir inteiramente os dados esparsos ou o período de tempo da análise ou mostrar estatísticas descritivas junto com explicações de advertência de que as tendências não são necessariamente significativas.

Existem vários tipos diferentes de dados e uma variedade de maneiras de descrever os dados, muitos dos quais se sobrepõem ou não são mutuamente exclusivos. A familiaridade com esses tipos é útil não apenas para escrever um bom SQL, mas também para decidir como analisar os dados de maneira apropriada. Você nem sempre conhece os tipos de dados com antecedência, e é por isso que a criação de perfil de dados é tão importante. Antes de chegarmos a isso, e aos nossos primeiros exemplos de código, farei uma breve revisão da estrutura de consulta SQL.

## Estrutura de consulta SQL

As consultas SQL têm cláusulas e sintaxe comuns, embora possam ser combinadas em um número quase infinito de maneiras para atingir os objetivos de análise. Este livro pressupõe que você tenha algum conhecimento prévio de SQL, mas revisarei o básico aqui para que tenhamos uma base comum para os exemplos de código que virão.

A cláusula *SELECT* determina as colunas que serão retornadas pela consulta. Uma coluna será retornada para cada expressão dentro da cláusula *SELECT* e as expressões são separadas por vírgulas. Uma expressão pode ser um campo da tabela, uma agregação como uma **sum** ou qualquer número de cálculos, como instruções CASE, conversões de tipo e várias funções que serão discutidas posteriormente neste capítulo e ao longo do livro.

A cláusula *FROM* determina as tabelas das quais as expressões na cláusula *SELECT* são derivadas. Uma “tabela” pode ser uma tabela de banco de dados, uma visualização (um tipo de consulta salva que funciona como uma tabela) ou uma subconsulta. Uma subconsulta é ela mesma uma consulta, entre parênteses, e o resultado é tratado como qualquer outra tabela pela consulta que a referencia. Uma consulta pode fazer referência a várias tabelas na cláusula *FROM*, embora elas devam usar um dos tipos *JOIN* junto com uma condição que especifica como as tabelas se relacionam. A condição *JOIN* geralmente especifica uma igualdade entre os campos em cada tabela, como `orders.customer_id = clients.customer_id`. Condições *JOIN* pode incluir vários campos e também pode especificar desigualdades ou intervalos de valores, como intervalos de datas. Veremos uma variedade de condições *JOIN* que atingem objetivos específicos de análise ao longo do livro. Um *INNER JOIN* retorna todos os registros que correspondem em ambas as tabelas. Um *LEFT JOIN* retorna todos os registros da primeira tabela, mas apenas os registros da segunda tabela que correspondem. Um *RIGHT JOIN* retorna todos os registros da segunda tabela, mas apenas os registros da primeira tabela que correspondem. Um *FULL OUTER JOIN* retorna todos os registros de ambas as tabelas. Um *CARTESIAN JOIN* pode resultar quando cada registro na primeira tabela corresponde a mais de um registro na segunda tabela. Cartesianos *JOINS* geralmente devem ser evitados, embora existam alguns casos de uso específicos, como gerar dados para preencher uma série temporal, em que os usaremos intencionalmente.

Finalmente, as tabelas na cláusula *FROM* podem ser *alias* ou receber um nome mais curto de uma ou mais letras que podem ser referenciadas em outras cláusulas na consulta. Os aliases evitam que os escritores de consulta precisem digitar nomes de tabela longos repetidamente e facilitam a leitura das consultas.



Embora tanto *LEFT JOIN* quanto *RIGHT JOIN* possam ser usados na mesma consulta, é muito mais fácil acompanhar sua lógica quando você fica com apenas um ou outro. Na prática, *LEFT JOIN* é muito mais usado do que *RIGHT JOIN*.

A cláusula *WHERE* especifica restrições ou filtros necessários para excluir ou remover linhas do conjunto de resultados. *WHERE* é opcional.

A cláusula *GROUP BY* é obrigatória quando a *SELECT* contém agregações e pelo menos um campo não agregado. Uma maneira fácil de lembrar o que deve constar na cláusula *GROUP BY* é que ela deve conter todos os campos que não fazem parte de uma agregação. Na maioria dos bancos de dados, há duas maneiras de listar os campos *GROUP BY*: por nome de 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 de posição, principalmente quando os campos *GROUP BY* contêm expressões complexas ou quando estou fazendo muitas iterações. Este livro normalmente usará a notação de posição.

## Como não matar seu banco de dados: *LIMIT* e amostragem

As tabelas de banco de dados podem ser muito grandes, contendo milhões ou bilhões de registros. A consulta em todos esses registros pode causar, no mínimo, problemas e, no pior, travar os bancos de dados. Para evitar receber chamadas mal-humoradas de administradores de banco de dados ou ficar bloqueado, é uma boa ideia limitar os resultados retornados durante a criação de perfil ou durante o teste de consultas. Cláusulas *LIMIT* e amostragem são duas técnicas que devem fazer parte de sua caixa de ferramentas.

*LIMIT* é adicionado 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 nessa etapa, e somente 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 usando *top*:

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

A amostragem pode ser realizada usando uma função em um campo de ID que possui uma distribuição aleatória de dígitos no início ou no final. O módulo ou função *mod* retorna o resto quando um inteiro é dividido por outro. Se o campo ID for um inteiro, *mod* pode ser usado para encontrar o último, dois ou mais dígitos e filtrar o resultado:

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

Isso retornará todos os pedidos cujos dois últimos dígitos são 06, que deve ser cerca de 1% do total. Se o campo for alfanumérico, você pode usar uma função *right()* para encontrar um certo número de dígitos no final:

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

Isso retornará todos os pedidos com um último dígito de B, que ser cerca de 3% do total se todas as letras e números forem igualmente comuns, uma suposição que vale a pena validar.

Limitar o conjunto de resultados também torna seu trabalho mais rápido, mas esteja ciente de que os subconjuntos de dados podem não conter todas as variações de valores e casos extremos que existem no conjunto de dados completo. Lembre-se de remover o *LIMIT* ou a amostragem antes de executar sua análise ou relatório final com sua consulta, ou você terá resultados engraçados!

Isso cobre os fundamentos da estrutura de consulta SQL. O Capítulo 8 entrará em detalhes adicionais sobre cada uma dessas cláusulas, algumas outras que são menos comumente encontradas, mas aparecem neste livro, e a ordem em que cada cláusula é avaliada. Agora que temos essa base, podemos nos voltar para uma das partes mais importantes do processo de análise: a criação de perfil de dados.

## Profiling: Distribuições

Profiling é a primeira coisa que faço quando começo a trabalhar com qualquer novo conjunto de dados. Observo como os dados são organizados em esquemas e tabelas. Exmino os nomes das tabelas para me familiarizar com os tópicos abordados, como clientes, pedidos ou visitas. Verifico 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, as tabelas podem incluir uma tabela *order\_detail* com quebras de item de linha relacionadas à tabela *order* por meio de um *order\_id*, enquanto a tabela *order* se relaciona à tabela *customer* por meio de um *customer\_id*. Se houver um dicionário de dados, eu o reviso e o comparo com os dados que vejo em uma amostra de linhas.

As tabelas geralmente representam as operações de uma organização ou algum subconjunto das operações, então penso em qual domínio ou domínios são cobertos, como comércio eletrônico, marketing ou interações de produtos. Trabalhar com dados é mais fácil quando temos conhecimento de como os dados foram gerados. A criação de perfis pode fornecer pistas sobre isso ou sobre quais perguntas fazer à fonte ou a pessoas dentro ou fora da organização responsáveis pela coleta ou geração dos dados. Mesmo quando você mesmo coleta os dados, a criação de perfil é útil.

Outro detalhe que verifico é como a história é representada, se é que é. Conjuntos de dados que são réplicas de bancos de dados de produção podem não conter valores anteriores para endereços de clientes ou status de pedidos, por exemplo, enquanto um data warehouse bem construído pode ter instantâneos diários de campos de dados alterados.

Os dados de perfil estão relacionados ao conceito de *análise exploratória de dados*, ou EDA, nomeado por John Tukey. Em seu livro com esse nome,<sup>1</sup> Tukey descreve como analisar conjuntos de dados computando vários resumos e visualizando os resultados. Ele inclui técnicas para analisar distribuições de dados, incluindo gráficos de caule e folha, gráficos de caixa e histogramas.

Depois de verificar algumas amostras de dados, começo a olhar para as distribuições. As distribuições me permitem entender o intervalo de valores que existem nos dados e com que frequência eles ocorrem, se há nulos e se existem valores negativos ao lado de positivos. As distribuições podem ser criadas com dados contínuos ou categóricos e também são chamadas de frequências. Nesta seção, veremos como criar histogramas, como o binning pode nos ajudar a entender a distribuição de valores contínuos e como usar n-tiles para obter mais precisão sobre as distribuições.

## Histogramas e Frequências

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

A consulta é direta. O número de linhas pode ser encontrado com `count(*)`, e o campo com perfil está no `GROUP BY`. Por exemplo, podemos verificar a frequência de cada tipo de `fruit` em uma tabela fictícia `fruit_inventory`:

---

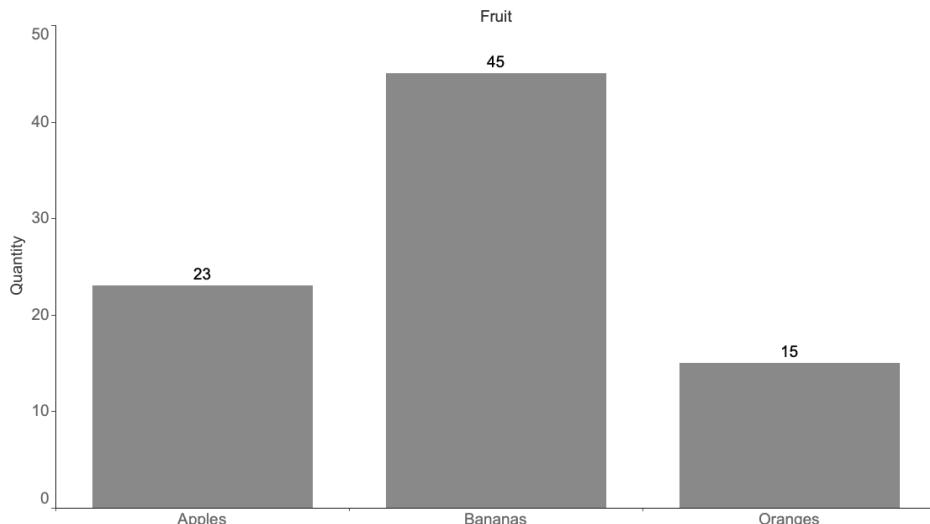
<sup>1</sup> John W. Tukey, *Exploratory Data Analysis* (Reading, MA: Addison-Wesley, 1977).

```
SELECT fruit, count(*) as quantity
FROM fruit_inventory
GROUP BY 1
;
```



Ao usar `count`, vale a pena levar um minuto para considerar se pode haver registros duplicados no conjunto de dados. Você pode usar `count(*)` quando quiser o número de registros, mas use `count distinct` para descobrir quantos itens exclusivos existem.

Um *gráfico de frequência* é uma maneira de visualizar o número de vezes que algo ocorre no conjunto de dados. O campo que está sendo perfilado geralmente é plotado no eixo x, com a contagem de observações no eixo y. [Figura 2-1](#) mostra um exemplo de plotagem da frequência de frutas de nossa consulta. Os gráficos de frequência também podem ser desenhados horizontalmente, o que acomoda bem nomes de valores longos. Observe que são dados categóricos sem qualquer ordem inherente.

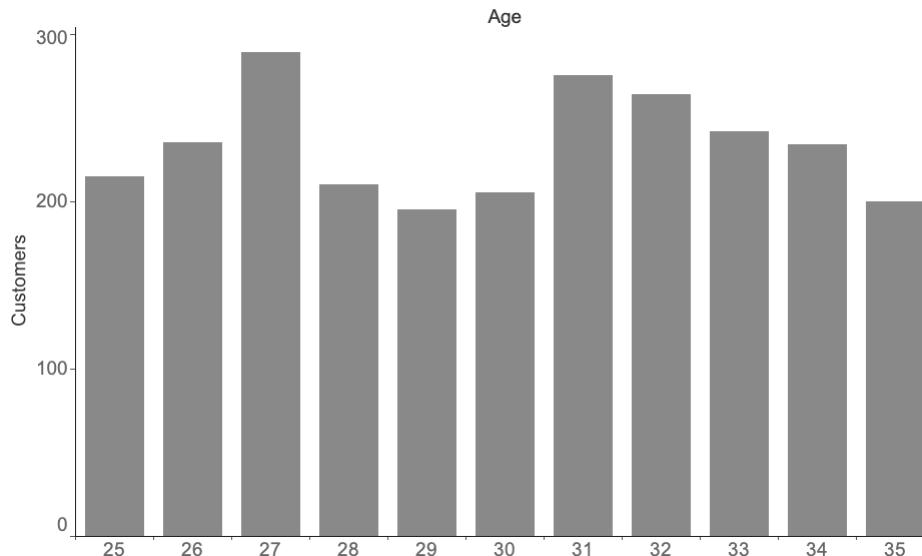


*Figura 2-1. Gráfico de frequência do inventário de frutas*

Um *histograma* é uma maneira de visualizar a distribuição de valores numéricos em um conjunto de dados e será familiar para aqueles com experiência em estatística. Um histograma básico pode mostrar a distribuição de idades em um grupo de clientes. Imagine que temos uma tabela `customers` que contém nomes, data de registro, idade e outros atributos. Para criar um histograma por idade, `GROUP BY` no campo numérico `age` e `count customer_id`:

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

Os resultados de nossa distribuição hipotética de idade estão representados graficamente na [Figura 2-2](#).



*Figura 2-2. Clientes por idade*

Outra técnica que usei repetidamente e que se tornou a base de uma das minhas perguntas de entrevista favoritas envolve uma agregação seguida de uma contagem de frequência. Forneço aos candidatos uma tabela hipotética chamada `orders`, que contém uma data, identificador de cliente, identificador de pedido e um valor, e depois peço 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 realizada com uma subconsulta. Primeiro, `count` o número de pedidos feitos por cada `customer_id` na subconsulta. A consulta externa usa o número de `orders` como uma categoria e `counts` 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 perfil pode ser aplicado sempre que você precisar ver com que frequência determinadas entidades ou atributos aparecem nos dados. Nesses exemplos, `count` foi usado, mas as outras agregações básicas (`sum`, `avg`, `min` e `max`) também podem ser usadas para criar histogramas. Por exemplo, podemos querer traçar o perfil dos clientes pela `sum` de todos os seus pedidos, o `avg` tamanho do pedido, `min` data do pedido, ou sua `max` (mais recente) data do pedido.

## Binning

Binning é útil ao trabalhar com valores contínuos. Em vez de contar o número de observações ou registros para cada valor, os intervalos de valores são agrupados e esses grupos são chamados de *compartimentos* ou *depósitos*. O número de registros que se enquadram em cada intervalo é então contado. Os compartimentos podem ser variáveis em tamanho ou ter um tamanho fixo, dependendo se seu objetivo é agrupar os dados em compartimentos que tenham um significado específico para a organização, tenham largura aproximadamente igual ou contenham números aproximadamente iguais de registros. Bins podem ser criados com instruções CASE, arredondamento e logaritmos.

Uma instrução CASE permite que a lógica condicional seja avaliada. Essas declarações são muito flexíveis e voltaremos a elas ao longo do livro, aplicando-as à criação de perfil de dados, limpeza, análise de texto e muito mais. A estrutura básica de uma 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, desigualdade ou outra condição lógica. O valor de retorno THEN pode ser uma constante, uma expressão ou um campo na tabela. Qualquer número de condições pode ser incluído, mas a instrução parará de ser executada e retornará o resultado na primeira vez 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 incluído, qualquer não correspondência retornará nulo. As instruções CASE também podem ser aninhadas para que o valor de retorno seja outra instrução CASE.



Os valores de retorno após THEN devem ser todos do mesmo tipo de dados (strings, numérico, BOOLEAN, etc.), caso contrário você receberá um erro. Considere converter para um tipo de dados comum, como string, se você encontrar isso.

Uma instrução CASE é uma maneira flexível de controlar o número de compartimentos, o intervalo de valores que se enquadram em cada compartimento e como os compartimentos são nomeados.

Acho-os particularmente úteis quando há uma cauda longa de valores muito pequenos ou muito grandes que quero agrupar juntos em vez de ter caixas vazias em parte da distribuição. Certos intervalos de valores têm um significado comercial que precisa ser recriado nos dados. Muitas empresas B2B separam seus clientes nas categorias “empresa” e “SMB” (pequenas e médias empresas) com base no número de funcionários ou receita, porque seus padrões de compra são diferentes. Como exemplo, imagine que estamos considerando ofertas de frete com desconto e queremos saber quantos clientes serão afetados. Podemos agrupar `order_amount` 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
;
```

Compartimentos de tamanho arbitrário podem ser úteis, mas em outras ocasiões, compartimentos de tamanho fixo são mais apropriados para a análise. Bins de tamanho fixo podem ser realizados de algumas maneiras, inclusive com arredondamento, logaritmos e n-tiles. Para criar compartimentos de largura igual, o arredondamento é útil. O arredondamento reduz a precisão dos valores, e geralmente pensamos em arredondar como reduzir o número de casas decimais ou removê-las completamente arredondando para o inteiro mais próximo. A função `round` toma a forma:

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

O número de casas decimais também pode ser um número negativo, permitindo que esta função arredonde para as dezenas, centenas, milhares e assim por diante. 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*

Decimal places	Formula	Result
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 comportamentos, principalmente em conjuntos de dados nos quais os maiores valores são ordens de magnitude maiores que os menores valores. A distribuição da riqueza das famílias, o número de visitantes do site em diferentes propriedades na internet e a força dos terremotos são exemplos de fenômenos que possuem essa propriedade. Embora não criem comportamentos de largura igual, os logaritmos criam comportamentos 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:

$$\log(number) = exponent$$

Nesse caso, 10 é chamado de base, e geralmente é a implementação padrão em bancos de dados, mas tecnicamente a base pode ser qualquer número. A Tabela 2-3 mostra os logaritmos para várias potências de 10.

*Table 2-3. Resultados da função log em potências de 10*

Formula	Result
$\log(1)$	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 usado em qualquer valor positivo, não apenas em múltiplos de 10. No entanto, a função logaritmo não funciona quando os valores podem ser menores ou iguais a 0; ele retornará null ou um erro, dependendo do banco de dados.

## n-Tiles

Você provavelmente está familiarizado com a mediana, ou valor médio, de um conjunto de dados. Este é o valor do percentil 50. Metade dos valores são maiores que a mediana e a outra metade são menores. Com quartis, preenchemos os valores dos percentis 25 e 75. Um quarto dos valores são menores e três quartos são maiores para o percentil 25; três quartos são menores e um quarto são maiores no percentil 75. Os decís dividem o conjunto de dados em 10 partes iguais. Tornando esse conceito genérico, n-tiles nos permitem calcular qualquer percentil do conjunto de dados: 27º percentil, 50,5º percentil e assim por diante.

## Funções do Window

As funções de n-tiles fazem parte de um grupo de funções SQL chamadas funções de janela ou analíticas. Ao contrário da maioria das funções SQL, que podem operar apenas na linha de dados atual, as funções de janela executam cálculos que abrangem 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 nas quais operar e a ordem dessas linhas. O formato geral de uma função de janela é:

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

A função pode ser qualquer uma das agregações normais (`count`, `sum`, `avg`, `min`, `max`) bem como várias funções especiais , incluindo `rank`, `first_value` e `ntile`. A cláusula *PARTITION BY* pode incluir zero ou mais campos. Quando nenhum campo for especificado, a função operará em toda a tabela, mas quando um ou mais campos forem especificados, a função operará apenas nessa seção de linhas. Por exemplo, podemos *PARTITION BY* um `customer_id` para realizar cálculos sobre todos os registros por cliente, reiniciando o cálculo para cada cliente. A cláusula *ORDER BY* determina a ordenação das linhas para funções que dependem disso; por exemplo, para `customer rank`, precisamos especificar um campo pelo qual ordená-los, como número de pedidos. Todos os principais tipos de banco de dados têm funções de janela, exceto as versões do MySQL anteriores à 8.0.2. Veremos essas funções úteis ao longo do livro, juntamente com explicações adicionais de como elas funcionam e como configurar os argumentos corretamente.

Muitos bancos de dados têm uma função `median` incorporada, mas contam com funções mais genéricas de n-tile para o resto. Essas funções são funções de janela, computando em um intervalo de linhas para retornar um valor para uma única linha. Eles recebem um argumento que especifica o número de compartimentos para dividir os dados e, opcionalmente, uma *PARTITION BY* e/ou uma cláusula *ORDER BY*:

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

Como exemplo , imagine que tivemos 12 transações com `order_amounts` 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. Realizar um cálculo `ntile` com 10 caixas classifica cada `order_amount` e atribui uma caixa de 1 a 10:

<code>order_amount</code>	<code>ntile</code>
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

Isso pode ser usado para armazenar registros na prática, primeiro calculando o `ntile` de cada linha em uma subconsulta e, em seguida, envolvendo-o em uma consulta externa que usa `min` e `max` para encontrar o valor superior e limites inferiores 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 compartimentos em que os dados se enquadram, `percent_rank` retorna o percentil. Não leva nenhum argumento, mas requer parênteses e, opcionalmente, recebe uma `PARTITION BY` e/ou uma cláusula `ORDER BY`:

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

Embora não seja tão útil quanto `ntile` para binning, `percent_rank` pode ser usado para criar uma distribuição contínua, ou pode ser usado como uma saída para relatórios ou análises adicionais. Tanto `ntile` quanto `percent_rank` podem ser caros para calcular em grandes conjuntos de dados, pois exigem a classificação de todas as linhas. Filtrar a tabela apenas para o conjunto de dados que você precisa ajuda. Alguns bancos de dados implementaram versões aproximadas das funções que são mais rápidas de calcular e geralmente retornam resultados de alta qualidade se a precisão absoluta não for necessária. Veremos usos adicionais para n-tiles na discussão da detecção de anomalias no [Capítulo 6](#).

Em muitos contextos, não existe uma única maneira correta ou objetivamente melhor de observar as distribuições de dados. Há uma margem de manobra significativa para os analistas usarem as técnicas anteriores para entender os dados e apresentá-los a outras pessoas. No entanto, os cientistas de dados precisam usar o julgamento e devem trazer seu radar ético sempre que compartilhar distribuições de dados confidenciais.

## Profiling: Qualidade dos dados

A qualidade dos dados é absolutamente crítica quando se trata de criar uma boa análise. Embora isso possa parecer óbvio, foi uma das lições mais difíceis que aprendi em meus anos de trabalho com dados. É fácil se concentrar demais na mecânica de processamento

dos dados, encontrar técnicas de consulta inteligentes e apenas a visualização correta, apenas para que as partes interessadas ignorem tudo isso e apontem a inconsistência dos dados. Garantir a qualidade dos dados pode ser uma das partes mais difíceis e frustrantes da análise. O ditado “garbage in, garbage out” captura apenas parte do problema. Bons ingredientes e suposições incorretas também podem levar à saída de lixo.

Comparar dados com a verdade, ou o que se sabe ser verdade, é ideal, embora nem sempre seja possível. Por exemplo, se você estiver trabalhando com uma réplica de uma produção de banco de dados, poderá comparar as contagens de linhas em cada sistema para verificar se todas as linhas chegaram ao banco de dados de réplica. Em outros casos, você pode saber o valor em dólares e a contagem de vendas em um determinado mês e, portanto, pode consultar essas informações no banco de dados para garantir que a `sum` das vendas e `count` de registros correspondam. Muitas vezes, a diferença entre os resultados da sua consulta e o valor esperado se resume ao fato de você ter aplicado os filtros corretos, como excluir pedidos cancelados ou contas de teste; como você lidou com nulos e anomalias ortográficas; e se você configurou as condições corretas `JOIN` entre as tabelas.

A criação de perfil é uma maneira de descobrir problemas de qualidade de dados desde o início, antes que eles afetem negativamente os resultados e as conclusões extraídas dos dados. A criação de perfil revela nulos, codificações categóricas que precisam ser decifradas, campos com vários valores que precisam ser analisados e formatos incomuns de data e hora. A criação de perfil também pode descobrir lacunas e alterações em etapas nos dados que resultaram do rastreamento de alterações ou interrupções. Os dados raramente são perfeitos e, muitas vezes, é apenas por meio de seu uso na análise que os problemas de qualidade de dados são descobertos.

## Detectando duplicatas

Uma *duplicata* é quando você tem duas (ou mais) linhas com as mesmas informações. Duplicatas podem existir por vários motivos. Um erro pode ter sido cometido durante a entrada de dados, se houver alguma etapa manual. Uma chamada de rastreamento pode ter sido disparada duas vezes. Uma etapa de processamento pode ter sido executada várias vezes.muitos-para-muitos oculto `JOIN`. Seja como for, as duplicatas podem realmente atrapalhar sua análise. Lembro-me de momentos no início da minha carreira quando pensei que tinha uma grande descoberta, apenas para ter um gerente de produto apontando que meu número de vendas era o dobro das vendas reais. É embaraçoso, corrói a confiança e requer retrabalho e, às vezes, revisões minuciosas do código para encontrar o problema. Aprendi a verificar se há duplicatas à medida que vou.

Felizmente, é relativamente fácil encontrar duplicatas em nossos dados. Uma maneira é inspecionar uma amostra, com todas as colunas ordenadas:

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

Isso revelará se os dados estão cheios de duplicatas, por exemplo, ao analisar um conjunto de dados totalmente novo, quando você suspeitar que um processo está gerando duplicatas ou após um possível *JOIN*. Se houver apenas algumas duplicatas, elas podem não aparecer na amostra. E percorrer os dados para tentar identificar duplicatas está sobrecarregando seus olhos e cérebro. Uma maneira mais sistemática de encontrar duplicatas é *SELECT* as colunas `e count` as linhas (isso pode parecer familiar da discussão sobre histogramas!):

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

Isso informará se há casos de duplicatas. Se a consulta retornar 0, você estará pronto. Para 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. Como é avaliado após a agregação e *GROUP BY*, *HAVING* pode ser usado para filtrar o valor de 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 estratégias para manter suas consultas SQL organizadas.

Para obter detalhes completos sobre quais registros têm duplicatas, você pode listar todos os campos e usar essas informações para identificar 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 duplicatas é uma coisa; descobrir o que fazer com eles é outra. É quase sempre útil entender por que as duplicatas estão ocorrendo e, se possível, corrigir o problema upstream. Um processo de dados pode ser melhorado para reduzir ou remover a duplicação? Existe um erro em um processo ETL? Você não conseguiu explicar um relacionamento um-para-muitos em um *JOIN*? A seguir, veremos algumas opções para manipular e remover duplicatas com SQL.

## Desduplicação com GROUP BY e DISTINCT

Duplicações acontecem e nem sempre são resultado de dados incorretos. Por exemplo, imagine que queremos encontrar uma lista de todos os clientes que concluíram uma transação com sucesso para que possamos enviar a eles um cupom para o próximo pedido. Podemos *JOIN* a tabela `customers` da tabela `transactions`, o que restringiria os registros retornados apenas aos clientes que aparecem nas `transações`:

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

No entanto, isso retornará uma linha para cada cliente para cada transação, e esperamos que pelo menos alguns clientes tenham feito transações mais de uma vez. Criamos duplicatas acidentalmente, não porque haja algum problema de qualidade de dados subjacente, mas porque não tomamos o cuidado de evitar duplicação nos resultados. Felizmente, existem várias maneiras de evitar isso com SQL. Uma maneira de remover duplicatas é usar 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 um *GROUP BY*, que, embora normalmente visto em conexão com uma agregação, também desduplicará da mesma maneira que *DISTINCT*. Lembro-me da primeira vez que vi um colega usar o *GROUP BY* sem uma desduplicação de agregação —

nem sabia que era possível. Acho um pouco menos intuitivo 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 realizar uma agregação que retorne uma linha por entidade. Embora tecnicamente não deduplicando, tem um efeito semelhante. Por exemplo, se tivermos várias transações do mesmo cliente e precisarmos retornar um registro por cliente, poderíamos encontrar o *min* (primeiro) e/ou o *max* (mais recente) *transaction\_date*:

```
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
;
```

Dados duplicados, ou dados que contêm vários registros por entidade, mesmo que tecnicamente não sejam duplicados, é um dos motivos mais comuns para resultados de consulta incorretos. Você pode suspeitar de duplicatas como a causa se, de repente, o número de clientes ou o total de vendas retornado por uma consulta for muitas vezes maior do que o esperado. Felizmente, existem várias técnicas que podem ser aplicadas para evitar que isso ocorra.

Outro problema comum é a falta de dados, aos quais nos voltaremos a seguir.

## Preparando: Limpeza de dados

A criação de perfil geralmente revela onde as alterações podem tornar os dados mais úteis para análise. Algumas das etapas são transformações CASE, ajuste para nulo e alteração de tipos de dados.

## Limpando Dados com Transformações CASE

As instruções CASE podem ser usadas para executar uma variedade de tarefas de limpeza, enriquecimento e resumo. Às vezes, os dados existem e são precisos, mas seria mais útil para análise se os valores fossem padronizados ou agrupados em categorias. A estrutura das declarações CASE foi apresentada anteriormente neste capítulo, na seção sobre binning.

Valores fora do padrão ocorrem por vários motivos. Os valores podem vir de sistemas diferentes com listas de opções ligeiramente diferentes, o código do sistema pode ter sido alterado,

as opções podem ter sido apresentadas ao cliente em diferentes idiomas ou o cliente pode ter preenchido o valor em vez de escolher em uma lista.

Imagine um campo contendo informações sobre o gênero de uma pessoa. Os valores que indicam uma pessoa do sexo feminino existem como “F”, “feminino” e “femme”. Podemos padronizar os valores assim:

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

Declarações CASE também podem ser usadas para adicionar categorização ou enriquecimento que não existe nos dados originais. Como exemplo, muitas organizações usam um Net Promoter Score, ou NPS, para monitorar o sentimento do cliente. As pesquisas de NPS pedem que os entrevistados classifiquem, em uma escala de 0 a 10, a probabilidade de recomendar uma empresa ou produto a um amigo ou colega. Escores de 0 a 6 são considerados detratores, 7 e 8 são passivos e 9 e 10 são promotores. A pontuação final é calculada subtraindo a porcentagem de detratores da porcentagem de promotores. Os conjuntos de dados de resultados da pesquisa geralmente incluem comentários de texto livre opcionais e às vezes são enriquecidos com informações que a organização conhece sobre a pessoa pesquisada. Dado um conjunto de dados de respostas da pesquisa do NPS, o primeiro passo é agrupar as respostas nas categorias de detrator, passivo e promotor:

```
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 dados pode diferir entre o campo que está sendo avaliado e o tipo de dados 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 especificar uma lista de itens em vez de ter que escrever uma igualdade para cada um separadamente. É útil quando a entrada não é contínua ou quando os valores em ordem não devem 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
```

Declarações CASE pode considerar várias colunas e pode conter lógica AND/OR. Eles também podem ser aninhados, embora muitas vezes isso possa ser evitado com 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 limpeza de dados

Limpar ou enriquecer dados com uma instrução CASE funciona bem desde que haja uma lista relativamente curta de variações, você pode encontrá-los todos nos dados, e não se espera que a lista de valores seja alterada. Para listas mais longas e que mudam com frequência, uma tabela de pesquisa pode ser uma opção melhor. Uma tabela de pesquisa existe no banco de dados e é estática ou preenchida com código que verifica novos valores periodicamente. A consulta se *unirá* à tabela de pesquisa para obter os dados limpos. Dessa forma, os valores limpos podem ser mantidos fora do seu código e usados por muitas consultas, sem que você precise se preocupar em manter a consistência entre eles. Um exemplo disso pode ser uma tabela de pesquisa que mapeia abreviações de estado para nomes de estado completos. Em meu próprio trabalho, geralmente começo com uma instrução CASE e crio uma tabela de pesquisa somente depois que a lista se torna indisciplinada ou quando fica claro que minha equipe ou eu precisaremos usar essa etapa de limpeza repetidamente.

Claro, vale a pena investigar se os dados podem ser limpos a montante. Certa vez, comecei com uma instrução CASE de 5 ou mais linhas que cresceram para 10 linhas e, eventualmente, para mais de 100 linhas, ponto em que a lista era indisciplinada e difícil de manter. Os insights foram valiosos o suficiente para que eu conseguisse convencer os engenheiros a alterar o código de rastreamento e enviar as categorizações significativas no fluxo de dados em primeiro lugar.

Outra coisa útil que você pode fazer com as instruções CASE é criar sinalizadores indicando se um determinado valor está presente, sem retornar o valor real. Isso pode ser útil durante a criação de perfil para entender o quão comum é a existência de um atributo específico. Outro uso para sinalização é durante a preparação de um conjunto de dados para análise estatística. Nesse caso, um sinalizador também é conhecido como variável dummy, assumindo valor 0 ou 1 e indicando a presença ou ausência de alguma variável qualitativa. Por exemplo, podemos criar sinalizadores `is_female` e `is_promoter` com instruções CASE nos campos `gender` e `likelihood` (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 com várias linhas por entidade, como itens de linha em um pedido, poderá nivelar os dados com uma instrução CASE envolvida em um agregado e transformá-lo em um sinalizador ao mesmo tempo usando 1 e 0 como o valor de retorno. Vimos anteriormente que um tipo de dado BOOLEAN é frequentemente utilizado para criar flags (campos que representam a presença ou ausência de algum atributo). Aqui, 1 é substituído por TRUE e 0 é substituído por FALSE para que um `max` possa ser aplicado. A maneira como isso funciona é que, para cada cliente, a instrução CASE retorna 1 para qualquer linha com um tipo de fruta "maçã". Em seguida, `max` é avaliado e retornará o maior valor de qualquer uma das linhas. Desde que um cliente tenha comprado uma maçã pelo menos uma vez, a bandeira será 1; se não, 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 sinalizadores, como exigir um limite ou quantidade de algo antes de rotular com um valor de 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 para limpar, enriquecer e sinalizar ou adicionar variáveis fictícias a conjuntos de dados. Na próxima seção, veremos algumas funções especiais relacionadas a instruções CASE que tratam especificamente de valores nulos.

## Conversões e tipos de conversão

Cada campo em um banco de dados é definido com um tipo de dados, que revisamos no início deste capítulo. Quando os dados são inseridos em uma tabela, os valores que não são do tipo do campo são rejeitados pelo banco de dados. Strings não podem ser inseridas em campos inteiros e booleanos não são permitidos em campos de data. Na maioria das vezes, podemos considerar os tipos de dados como

garantidos e aplicar funções de string a strings, funções de data a datas e assim por diante. Ocasionalmente, no entanto, precisamos substituir o tipo de dados do campo e forçá-lo a ser outra coisa. É aqui que entram as conversões de tipo e a conversão.

*Funções de conversão de tipo* permitem que pedaços de dados com o formato apropriado sejam alterados de um tipo de dado para outro. A sintaxe vem em algumas formas que são basicamente equivalentes. Uma maneira de alterar o tipo de dados é com a função `cast`, `cast (input as data_type)`, ou dois pontos, `input :: data_type`. Ambos 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 ao categorizar valores numéricos com algum valor superior ou inferior ilimitado. Por exemplo, no código a seguir, deixar os valores menores ou iguais a 3 como inteiros enquanto retorna a string “4+” para valores mais altos resultaria em um erro:

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

A conversão de 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 são analisados de uma string, e então queremos agregar os valores ou usar funções matemáticas neles. Imagine que temos um conjunto de dados de preços, mas os valores incluem o cifrão (\$) e, portanto, o tipo de dados do campo é VARCHAR. Podemos remover o caractere \$ com uma função chamada `replace`, que será mais discutida durante nossa análise de texto no [Capítulo 5](#):

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

O resultado ainda é um VARCHAR, portanto, tentar aplicar uma agregação retornará um erro. Para corrigir isso, podemos `cast` o resultado como um FLOAT:

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

Dates and datetimes podem vir em uma variedade desconcertante de formatos, e entender como *convertê-los* no formato desejado é útil. Mostrarei alguns exemplos de conversão de tipos aqui, e [Capítulo 3](#) entrará em mais detalhes sobre cálculos de data e hora. Como um exemplo simples, imagine que dados de transações ou eventos geralmente cheguem ao

banco de dados como um TIMESTAMP, mas queremos resumir alguns valores, como transações por dia. O simples agrupamento pelo carimbo de data/hora resultará em mais linhas do que o necessário. A conversão do TIMESTAMP para um DATE reduz o tamanho dos resultados e atinge nosso objetivo de sumarização:

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

Da mesma forma, um DATE pode ser convertido em um TIMESTAMP quando uma função SQL requer um argumento TIMESTAMP. Às vezes, o ano, mês e dia são armazenados em colunas separadas ou acabam como elementos separados porque foram analisados a partir de uma string mais longa. Estes, então, precisam ser montados de volta em uma data. Para fazer isso, usamos o operador de concatenação || (tubulação dupla) ou função `concat` e, em seguida, converte o resultado em um DATE. Qualquer uma dessas sintaxes funciona e retorna o mesmo valor:

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

Ou equivalente:

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

Outra maneira de converter entre valores de string e datas é usando a função `date`. Por exemplo, podemos construir um valor de string como acima e convertê-lo em uma data:

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

As funções `to_datatype` podem receber um valor e uma string de formato e assim, você tem mais controle sobre como os dados são convertidos. [Tabela 2-4](#) resume as funções e seus propósitos. Eles são particularmente úteis ao converter dentro e fora dos formatos DATE ou DATETIME, pois permitem especificar a ordem dos elementos de data e hora.

*Tabela 2-4. As funções to\_datatype*

Função	Objetivo
<code>to_char</code>	Converte outros tipos em string
<code>to_number</code>	Converte outros tipos em numérico
<code>to_date</code>	Converte outros tipos em data, com partes de data especificadas
<code>to_timestamp</code>	Converte outros tipos em data, com partes de data e hora especificadas

Às vezes, o banco de dados converte automaticamente um tipo de dados. Isso é chamado *de coerção de tipo*. Por exemplo, os numéricos INT e FLOAT geralmente podem ser usados juntos em funções matemáticas ou agregações sem alterar explicitamente o tipo. Os valores CHAR e VARCHAR geralmente podem ser misturados. Alguns bancos de dados forçarão os campos BOOLEAN a valores 0 e 1, onde 0 é FALSE e 1 é TRUE, mas alguns bancos de dados exigem que você converta os valores explicitamente.

Alguns bancos de dados são mais exigentes do que outros sobre a mistura de datas e datas em conjuntos de resultados e funções. Você pode ler a documentação ou fazer alguns experimentos de consulta simples para saber como o banco de dados com o qual você está trabalhando lida com tipos de dados de forma implícita e explícita. Geralmente, há uma maneira de realizar o que você deseja, embora às vezes você precise ser criativo ao usar funções em suas consultas.

## Lidando com Nulos: Funções coalesce, nullif, nvl

Null foi um dos conceitos mais estranhos com os quais tive que me acostumar quando comecei a trabalhar com dados. Nulo simplesmente não é algo em que pensamos na vida cotidiana, onde estamos acostumados a lidar com quantidades concretas de coisas. *Null* tem um significado especial em bancos de dados e foi introduzido por Edgar Codd, o inventor do banco de dados relacional, para garantir que os bancos de dados tenham uma maneira de representar informações ausentes. Se alguém me perguntar quantos pára-quedas eu tenho, posso responder “zero”. Mas se a pergunta nunca for feita, tenho pára-quedas nulos.

Nulos podem representar campos para os quais nenhum dado foi coletado ou que não são aplicáveis a essa linha. Quando novas colunas são adicionadas a uma tabela, os valores das linhas criadas anteriormente serão nulos, a menos que sejam explicitamente preenchidos com algum outro valor. Quando duas tabelas são unidas por meio de um *OUTER JOIN*, nulos aparecerão em todos os campos para os quais não há registro correspondente na segunda tabela.

Nulos são problemáticos para certas agregações e agrupamentos, e diferentes tipos de bancos de dados os tratam de maneiras diferentes. Por exemplo, imagine que eu tenha cinco registros, com 5, 10, 15, 20 e null. A soma destes é 50, mas a média é 10 ou 12,5 dependendo se o valor nulo é contado no denominador. A questão inteira também pode ser considerada inválida, pois um dos valores é nulo. Para a maioria das funções de banco de dados, uma entrada nula retornará uma saída nula. Igualdades e desigualdades envolvendo nulo também retornam nulo. Uma variedade de resultados inesperados e frustrantes podem ser gerados a partir de suas consultas se você não estiver procurando por nulos.

Quando as tabelas são definidas, elas podem permitir nulos, rejeitar nulos ou preencher um valor padrão se o campo for deixado nulo. Na prática, isso significa que você nem sempre pode confiar em um campo para aparecer como nulo se os dados estiverem ausentes, porque ele pode ter sido preenchido com um valor padrão como 0. Certa vez, tive um longo debate com um engenheiro de dados quando se descobriu que as datas nulas no sistema de origem estavam padronizadas para “1970-01-01” em nosso data warehouse. Insisti que as datas deveriam ser nulas, para refletir o fato de que eram desconhecidas ou não aplicáveis. O engenheiro apontou que eu conseguia me lembrar de filtrar essas datas ou alterá-las de volta para null com uma instrução CASE. Eu finalmente venci ao apontar que um dia outro usuário que não estava tão ciente das nuances das datas padrão apareceria, executaria uma consulta e obteria o intrigante grupo de clientes cerca de um ano antes da empresa ser fundada.

Nulos geralmente são inconvenientes ou inadequados para a análise que você deseja fazer. Eles também podem tornar a saída confusa para o público-alvo de sua análise. As pessoas de negócios não necessariamente entendem como interpretar um valor nulo ou podem supor que valores nulos representam um problema com a qualidade dos dados.

## Strings Vazias

Um conceito relacionado, mas ligeiramente diferente de nulls, é *a string vazia*, onde não há valor, mas o campo não é tecnicamente nulo. Uma razão pela qual uma string vazia pode ser usada é para indicar que um campo é conhecido por estar em branco, em oposição a um nulo, onde o valor pode estar ausente ou desconhecido. Por exemplo, o banco de dados pode ter um campo `name_suffix` que pode ser usado para armazenar um valor como “Jr.” Muitas pessoas não têm um `name_suffix`, então uma string vazia é apropriada. Uma string vazia também pode ser usada como valor padrão em vez de null, ou como forma de superar uma restrição NOT NULL inserindo um valor, mesmo que vazio. Uma string vazia pode ser especificada em uma consulta com duas aspas:

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

A criação de perfil das frequências de valores deve revelar se seus dados incluem nulos, strings vazias ou ambos.

Existem algumas maneiras de substituir nulos por valores alternativos: instruções CASE e as funções especializadas `coalesce` e `nullif`. Vimos anteriormente que as instruções CASE podem verificar uma condição e retornar um valor. Eles também podem ser usados para verificar se há um 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 conseguir isso. 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` existe em alguns databases e é semelhante a `coalesce`, mas permite apenas dois argumentos.

A função `nullif` compara dois números e, se não forem iguais, retorna o primeiro número; se forem iguais, a função retornará null. Executando este código:

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

retorna 6, enquanto null é retornado por:

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

`nullif` é equivalente à seguinte declaração case mais prolixa:

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

Esta função pode ser útil para transformar valores em nulos quando você sabe que um determinado valor padrão foi inserido no banco de dados. Por exemplo, com meu exemplo de tempo padrão, poderíamos alterá-lo novamente para null usando:

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



Nulos podem ser problemáticos ao filtrar dados na cláusula `WHERE`. Retornar valores nulos é bastante simples:

```
WHERE my_field is null
```

No entanto, imagine que `my_field` contém alguns nulos e também alguns nomes de frutas. Eu gostaria de retornar todas as linhas que não são maçãs. Parece que isso deve funcionar:

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

No entanto, alguns bancos de dados excluíram as linhas “apple” e todas as linhas com valores nulos em `my_field`. Para corrigir isso, o SQL deve filtrar “apple” e incluir explicitamente nulos conectando as condições com OR:

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

Nulos são um fato da vida ao trabalhar com dados. Independentemente do motivo pelo qual ocorrem, muitas vezes precisamos considerá-los na criação de perfil e como alvos para limpeza de dados. Felizmente, existem várias maneiras de detectá-los com SQL, bem como várias funções úteis que nos permitem substituir nulos por valores alternativos. A seguir, veremos os dados ausentes, um problema que pode causar valores nulos, mas tem implicações ainda mais amplas e, portanto, merece uma seção própria.

## Dados ausentes

Os dados podem estar ausentes por vários motivos, cada um com suas próprias implicações sobre como você decide lidar com a ausência de dados. Um campo pode não ter sido exigido pelo sistema ou processo que o coletou, como com um opcional “como você ficou sabendo sobre nós?” campo em um fluxo de checkout de comércio eletrônico. Exigir esse campo pode criar atritos para o cliente e diminuir os checkouts bem-sucedidos.

Alternativamente, os dados podem ser normalmente necessários, mas não foram coletados devido a um erro de código ou erro humano, como em um questionário médico em que o entrevistador perdeu a segunda página de perguntas. Uma alteração na forma como os dados foram coletados pode resultar em registros antes ou depois da alteração com valores ausentes. Uma ferramenta de rastreamento de interações de aplicativos móveis pode adicionar um registro de campo adicional se a interação foi um toque ou uma rolagem, por exemplo, ou remover outro campo devido à alteração de funcionalidade. Os dados podem ficar órfãos quando uma tabela faz referência a um valor em outra tabela e essa linha ou a tabela inteira foi excluída ou ainda não foi carregada no data warehouse. Finalmente, os dados podem estar disponíveis, mas não no nível de detalhe ou granularidade necessários para a análise. Um exemplo disso vem dos negócios de assinatura, onde os clientes pagam anualmente por um produto mensal e queremos analisar a receita mensal.

Além de traçar o perfil dos dados com histogramas e análise de frequência, muitas vezes podemos detectar dados ausentes comparando valores em duas tabelas. Por exemplo, podemos esperar que cada cliente nas `transactions` também tenha um registro na tabela `customer`. Para verificar isso, consulte as tabelas usando um *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
;
```

Dados ausentes podem ser um sinal importante por si só, portanto, não assuma que eles sempre precisam ser corrigidos ou preenchidos. Dados ausentes podem revelar o design do sistema subjacente ou vieses no processo de coleta de dados.

Registros com campos ausentes podem ser totalmente filtrados, mas muitas vezes queremos mantê-los e, em vez disso, fazer alguns ajustes com base no que sabemos sobre valores esperados ou típicos. Temos algumas opções, chamadas de técnicas *imputação*, para preenchimento de dados faltantes. Estes incluem o preenchimento com uma média ou mediana do conjunto de dados, ou com o valor anterior. É importante documentar os dados ausentes e como eles foram substituídos, pois isso pode afetar a interpretação e o uso dos dados a jusante. Os valores imputados podem ser particularmente problemáticos quando os dados são usados em aprendizado de máquina, por exemplo.

Uma opção comum é preencher os dados ausentes com um valor constante. O preenchimento com um valor constante pode ser útil quando o valor é conhecido para alguns registros, mesmo que não tenham sido preenchidos no banco de dados. Por exemplo, imagine que houve um bug de software que impediu o preenchimento do `price` de um item chamado “xyz”, mas sabemos que o preço é sempre \$20. Uma instrução CASE pode ser adicionada à consulta para lidar com isso:

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

Outra opção é preencher com um valor derivado, seja uma função matemática em outras colunas ou uma declaração CASE. Por exemplo, imagine que temos um campo para o `net_sales` para cada transação. Devido a um bug, algumas linhas não têm esse campo preenchido, mas têm os campos `gross_sales` e `discount` preenchidos. Podemos calcular `net_sales` subtraindo `discount` de `gross_sales`:

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

Os valores ausentes também podem ser preenchidos com valores de outras linhas no conjunto de dados. Carregar um valor da linha anterior é chamado de *preenchimento para frente*, enquanto usar um valor da próxima linha é chamado de *preenchimento para trás*. Isso pode ser feito com as funções de janela `lag` e `lead`, respectivamente. Por exemplo, imagine que nossa tabela de transações tenha um campo `product_price` que armazena o preço sem desconto que um cliente paga por um `product`. Ocasionalmente este campo não é preenchido, mas podemos supor que o preço é o mesmo que o preço pago pelo último cliente para comprar aquele `product`. Podemos preencher com o valor anterior usando a função `lag`, `PARTITION BY` do `product` para garantir que o preço seja obtido apenas do mesmo `product` e `ORDER BY` a data apropriada para garantir que o preço seja retirado da transação anterior mais recente:

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

A função `lead` pode ser usada para preencher com `product_price` para a transação a seguir. Alternativamente, poderíamos pegar a `avg` dos preços do `product` e usá-la para preencher o valor ausente. Preencher com valores anteriores, próximos ou médios envolve fazer algumas suposições sobre valores típicos e o que é razoável incluir em uma análise. É sempre uma boa ideia verificar os resultados para certificar-se de que são plausíveis e observar que você interpolou os dados quando não estiverem disponíveis.

Para dados que estão disponíveis, mas não com a granularidade necessária, geralmente precisamos criar linhas adicionais no conjunto de dados. Por exemplo, imagine que temos uma `customer_subscriptions` com os campos `subscription_date` e `annual_amount`. Podemos dividir esse valor de assinatura anual em 12 valores iguais de receita mensal dividindo por 12, convertendo efetivamente ARR (receita recorrente anual) em MRR (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 fica um pouco tedioso, principalmente se os períodos de assinatura puderem ser de dois, três ou cinco anos, além de um ano. Também não é útil se o que queremos são as datas reais dos meses. Em teoria, 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 de clientes ao longo do tempo, a codificação dos nomes dos meses não será precisa. Poderíamos usar instruções CASE em combinação com nomes de meses codificados, mas, novamente, isso é tedioso e provavelmente será propenso a erros à medida que você adiciona uma lógica mais complicada. Em vez disso, criar novas linhas por meio de um *JOIN* para uma tabela, como uma dimensão de data, fornece uma solução elegante.

Uma *dimensão de data* é uma tabela estática que tem uma linha por dia, com atributos opcionais de data estendida, como dia da semana, nome do mês, final do mês e ano fiscal. As datas se estendem o suficiente no passado e no futuro o suficiente para cobrir todos os usos previstos. Como existem apenas 365 ou 366 dias por ano, tabelas que cobrem até 100 anos não ocupam muito espaço. Figura 2-3 mostra uma amostra dos dados em uma tabela de dimensão de data. O código de exemplo para criar uma dimensão de data usando funções SQL está no [site GitHub](#).

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 estiver usando um banco de dados Postgres, a função `generate_series` pode ser usada para criar uma dimensão de data para preencher a tabela inicialmente ou se a criação de uma tabela não for uma opção. Ela assume o seguinte formato:

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

Nesta função, *start* é a primeira data desejada na série, *stop* é a última data e *step interval* é o período de tempo entre os valores. O *step interval* pode ter 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, então “2000-01-01” é convertido como um TIMESTAMP. Podemos então criar uma consulta que resulte em uma linha para todos os dias, independentemente de um cliente fazer o pedido em um determinado dia. Isso é útil quando queremos garantir que um cliente seja contado para cada dia ou quando queremos especificamente contar ou analisar os dias em que 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 de assinatura, podemos usar a dimensão de data para criar um registro para cada mês por JOINing a dimensão de data em datas que estão entre o `subscription_date` e 11 meses depois (para 12 meses no total):

```
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 vários motivos, e entender a causa raiz é importante para decidir como lidar com isso. Há várias opções para localizar e substituir dados ausentes. Isso inclui o uso de instruções CASE para definir valores padrão, derivando valores executando cálculos em outros campos na mesma linha e interpolando de outros valores na mesma coluna.

A limpeza de dados é uma parte importante do processo de preparação de dados. Os dados podem precisar ser limpos por vários motivos diferentes.

Algumas limpezas de dados precisam ser feitas para corrigir a baixa qualidade dos dados, como quando há valores inconsistentes ou ausentes nos dados brutos, enquanto outras limpezas de dados são feitas para tornar as análises mais fáceis ou mais significativas. A flexibilidade do SQL nos permite realizar tarefas de limpeza de várias maneiras.

Depois que os dados são limpos, uma próxima etapa comum no processo de preparação é moldar o conjunto de dados.

## Preparando: Dados de modelagem

*Dados de modelagem* refere-se à manipulação da forma como os dados são representados em colunas e linhas. Cada tabela no banco de dados tem uma forma. O conjunto de resultados de cada consulta tem uma forma. A modelagem de dados pode parecer um conceito bastante 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 na modelagem de dados é descobrir a *granularidade* dos dados de que você precisa. Assim como as rochas podem variar em tamanho de pedregulhos gigantes até grãos de areia, e ainda mais até poeira microscópica, os dados também podem ter níveis variados de detalhes. Por exemplo, se a população de um país é uma pedra, então a população de uma cidade é uma pequena pedra, e a de uma casa é um grão de areia. Os dados em um nível menor de detalhes podem incluir nascimentos e mortes individuais ou mudanças de uma cidade ou país para outro.

O *achatamento de dados* é outro conceito importante na modelagem. Isso se refere à redução do número de linhas que representam uma entidade, incluindo uma única linha. Unir várias tabelas para criar um único conjunto de dados de saída é uma maneira de nivelar os dados. Outra forma é através da agregação.

Nesta seção, abordaremos primeiro algumas considerações para escolher formas de dados. Em seguida, veremos alguns casos de uso comuns: pivotar e não pivotar. Veremos exemplos de modelagem de dados para análises específicas nos capítulos restantes. O Capítulo 8 entrará em mais detalhes sobre como manter o SQL complexo organizado ao criar conjuntos de dados para análise posterior.

## Para qual saída: BI, visualização, estatísticas, ML

Decidir como moldar seus dados com SQL depende muito do que você planeja fazer com os dados posteriormente. Geralmente, é uma boa ideia gerar um conjunto de dados que tenha o menor número de linhas possível e ainda atender à sua necessidade de granularidade. Isso alavancará o poder de computação do banco de dados, reduzirá o tempo necessário para mover dados do banco de dados para outro lugar e reduzirá a quantidade de processamento que você ou outra pessoa precisa fazer em outras ferramentas. Algumas das outras ferramentas para as quais sua saída pode ir são uma ferramenta de BI para relatórios e painéis, uma planilha para usuários de negócios examinarem, uma ferramenta de estatística como R ou um modelo de aprendizado de máquina em Python – ou você pode gerar os dados diretamente para uma visualização criada com uma variedade de ferramentas.

Ao enviar dados para uma ferramenta de inteligência de negócios para relatórios e painéis, é importante entender o caso de uso. Os conjuntos de dados podem precisar ser muito detalhados para permitir a exploração e o fatiamento pelos usuários finais. Eles podem precisar ser pequenos e agregados e incluir cálculos específicos para permitir carregamento rápido e tempos de resposta em painéis executivos. É importante entender como a ferramenta funciona e se ela funciona melhor com conjuntos de dados menores ou é arquitetada para executar suas próprias agregações em conjuntos de dados maiores. Não existe uma resposta “tamanho único”. Quanto mais você souber sobre como os dados serão usados, mais preparado estará para moldá-los adequadamente.

Conjuntos de dados menores, agregados e altamente específicos geralmente funcionam melhor para visualizações, sejam eles criados em software comercial ou usando uma linguagem de programação como R, Python ou JavaScript. Pense no nível de agregação e fatias, ou vários elementos, que os usuários finais precisarão filtrar. Às vezes, os conjuntos de dados exigem uma linha para cada fatia, bem como uma fatia “tudo”. Você pode precisar *UNION* em duas consultas — uma no nível de detalhes e outra no nível de “tudo”.

Ao criar saída para pacotes de estatísticas ou modelos de aprendizado de máquina, é importante entender 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 um registro por transação com seus atributos associados, bem como atributos do cliente. Geralmente, a saída para modelagem seguirá a noção de “dados organizados” proposta por Hadley Wickham.<sup>2</sup> Tidy data tem estas propriedades:

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

Em seguida, veremos como usar o SQL para transformar dados da estrutura na qual eles existem em seu banco de dados em qualquer outra estrutura dinâmica ou não dinâmica necessária para análise.

## Dinamizando 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 colunas, de acordo com os valores de outro atributo. Na interseção de cada linha e coluna, uma estatística de resumo como `sum`, `count` ou `avg` é calculada. As tabelas dinâmicas geralmente são uma boa maneira de resumir dados para o público empresarial, pois remodelam os dados em um formato mais compacto e de fácil compreensão.

---

<sup>2</sup> Hadley Wickham, “Tidy Data”, *Journal of Statistical Software* 59, no. 10 (2014): 1–23,  
<https://doi.org/10.18637/jss.v059.i10>.

As tabelas dinâmicas são amplamente conhecidas por sua implementação no Microsoft Excel, que possui uma interface de arrastar e soltar para criar os resumos dos dados.

Tabelas dinâmicas, ou saída dinâmica, podem ser criadas em SQL usando uma instrução CASE junto com uma ou mais funções de agregação. Vimos instruções CASE várias vezes até agora, e remodelar dados é outro caso de uso importante para elas. Por exemplo, imagine que temos uma tabela `orders` com uma linha para cada compra realizada pelos clientes. Para achatar os dados, *GROUP BY* o `customer_id` e *sum* o `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 um pivô, também criaremos colunas para cada um dos valores de um atributo. Imagine que a tabela `orders` também tenha um campo `product` que contém o tipo de item comprado e o `order_date`. Para criar uma saída dinâmica, *GROUP BY* a `order_date` e *sum* o resultado de uma instrução CASE que retorna a `order_amount` sempre que a linha atende 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`, você pode usar opcionalmente “`else 0`” para evitar nulos no conjunto de resultados. Com `count` ou `count distinct`, no entanto, você não deve incluir uma instrução ELSE, pois isso inflaria o conjunto de resultados.

Isso ocorre porque o banco de dados não contará um nulo, mas contará um valor substituto, como zero.

Girar com instruções CASE é bastante útil, e ter essa capacidade abre designs de tabela de data warehouse que são longos e estreitos em vez de largos, o que pode ser melhor para armazenar dados esparsos, porque adicionar colunas a uma tabela pode ser uma operação cara. Por exemplo, em vez de armazenar vários atributos de cliente em muitas colunas diferentes, uma tabela pode conter vários 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 seu valor. Os dados podem então ser dinamizados conforme necessário para montar um registro de cliente com os atributos desejados. Esse design é eficiente quando há muitos atributos esparsos (apenas um subconjunto de clientes possui valores para muitos dos atributos).

Dinamização de dados com uma combinação de agregação e instruções CASE funciona bem quando há um número finito de itens a serem dinâmicos. Para pessoas que trabalharam com outras linguagens de programação, é essencialmente um loop, mas escrito explicitamente linha por linha. Isso lhe dá muito controle, por exemplo, se você quiser calcular métricas diferentes em cada coluna, mas também pode ser tedioso. Girar com instruções case não funciona bem quando novos valores chegam constantemente ou mudam rapidamente, pois o código SQL precisaria ser atualizado constantemente. Nesses casos, empurrar a computação para outra camada de sua pilha de análise, como uma ferramenta de BI ou linguagem estatística, pode ser mais apropriado.

## Desarticulando com instruções UNION

Às vezes, temos o problema oposto e precisamos mover os dados armazenados em colunas para linhas em vez de criar dados organizados. Essa operação é chamada de *unpivoting*. Os conjuntos de dados que podem precisar de desdinamização são aqueles que estão em um formato de tabela dinâmica. Como exemplo, as populações dos países norte-americanos em intervalos de 10 anos a partir de 1980 são mostradas na [Figura 2-4](#).

<b>Country</b>	<b>year_1980</b>	<b>year_1990</b>	<b>year_2000</b>	<b>year_2010</b>
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 do país por ano (em milhares)<sup>3</sup>*

---

<sup>3</sup> US Census Bureau, “International Data Base (IDB)”, atualizado pela última vez em dezembro de 2020, <https://www.census.gov/ data-tools/demo/idb>.

Para transformar isso em um conjunto de resultados com uma linha por país por ano, podemos usar um operador *UNION*. *UNION* é uma maneira de combinar conjuntos de dados de várias consultas em um único conjunto de resultados. Existem duas formas, *UNION* e *UNION ALL*. Ao usar *UNION* ou *UNION ALL*, os números de colunas em cada consulta de componente devem corresponder. Os tipos de dados devem corresponder ou ser compatíveis (inteiros e floats podem ser misturados, mas inteiros e strings não podem). Os nomes das colunas no conjunto de resultados vêm da primeira consulta. O alias dos campos nas consultas restantes é, portanto, opcional, mas pode tornar uma 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
...	...	...

Neste exemplo, usamos uma constante para codificar o ano, a fim de acompanhar o ano ao qual o valor da população corresponde. Os valores codificados podem ser de qualquer tipo, dependendo do seu caso de uso. Pode ser necessário converter explicitamente determinados valores codificados, como ao inserir uma data:

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

Qual é a diferença entre *UNION* e *UNION ALL*? Ambos podem ser usados para anexar ou empilhar dados dessa maneira, mas são um pouco diferentes. *UNION* remove duplicatas do conjunto de resultados, enquanto *UNION ALL* retém todos os registros, sejam duplicados ou não. *UNION ALL* é mais rápido, pois o banco de dados não precisa passar pelos dados para encontrar duplicatas. Ele também garante que cada registro termine no conjunto de resultados. Eu costumo usar *UNION ALL*, usando *UNION* apenas quando tenho um motivo para suspeitar de dados duplicados.

*UNIONing* os dados também podem ser úteis para reunir dados de diferentes fontes. Por exemplo, imagine que temos uma tabela `populations` com dados anuais por país e outra tabela `PIB` com produto interno bruto anual, ou PIB. Uma opção é *JOIN* as tabelas e obter um conjunto de resultados com uma coluna para população e outra para PIB:

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

Outra opção é *UNION ALL* os conjuntos de dados para que acabemos com um conjunto de dados 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
;
```

Qual abordagem você usa depende em grande parte da saída que você precisa para sua análise. A última opção pode ser útil quando você tem várias métricas diferentes em tabelas diferentes e nenhuma tabela tem um conjunto completo de entidades (neste caso, países). Esta é uma abordagem alternativa para um *FULL OUTER JOIN*.

## Funções pivot e unpivot

Reconhecendo que os casos de uso pivot e unpivot são comuns, alguns fornecedores de banco de dados implementaram funções para fazer isso com menos linhas de código. O Microsoft SQL Server e o Snowflake têm funções `pivot` que assumem a forma de expressões extras na cláusula *WHERE*. Aqui, agregação é qualquer função de agregação, como `sum` ou `avg`, o `value_column` é o campo a ser agregado e uma coluna será criada para cada valor da `label_column` listado como um rótulo:

```
SELECT...
FROM...
pivot(aggregation(value_column)
      FOR label_column IN (label_1, label_2, ...))
;
```

Poderíamos reescrever o exemplo de pivô anterior que usava instruções CASE da seguinte forma:

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

Embora esta sintaxe seja mais compacta do que a construção CASE que vimos anteriormente, as colunas desejadas ainda precisam ser especificadas. Como resultado, `pivot` não resolve o problema de novos conjuntos de valores que chegam ou mudam rapidamente campos que precisam ser transformados em colunas. Postgres tem uma função semelhante `crosstab`, disponível no módulo `tablefunc`.

Microsoft SQL Server e Snowflake também possuem funções `unpivot` que funcionam de forma semelhante às expressões na cláusula *WHERE* e transformam linhas em colunas:

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

Por exemplo, os `country_populations` do exemplo anterior podem ser reformulados da seguinte maneira:

```
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 *UNION* ou *UNION ALL* que vimos anteriormente, mas a lista de colunas deve ser especificada na consulta.

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

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

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

```
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 muitos formatos e formas diferentes, e são nem sempre no formato necessário em nossa saída. Existem várias opções para remodelar dados por meio de dinamização ou não dinamização, seja com instruções CASE ou *UNIONS*, ou com funções específicas do banco de dados. Entender como manipular seus dados para moldá-los da maneira que você deseja lhe dará maior flexibilidade em sua análise e na forma de apresentar seus resultados.

## Conclusão

Preparar dados para análise pode parecer o trabalho que você faz antes de chegar ao trabalho real de análise, mas é tão fundamental para entender os dados que sempre acho que é um tempo bem gasto. Compreender os diferentes tipos de dados que você provavelmente encontrará é fundamental, e você deve dedicar um tempo para entender os tipos de dados em cada tabela com a qual trabalha. A criação de perfil de dados nos ajuda a saber mais sobre o que está no conjunto de dados e a examiná-lo quanto à qualidade. Costumo retornar à criação de perfil em meus projetos de análise, pois aprendo mais sobre os dados é preciso verificar os resultados da minha consulta ao longo do caminho à medida que criou a complexidade. A qualidade dos dados provavelmente nunca deixará de ser um problema, por isso analisamos algumas maneiras de lidar com a limpeza e o aprimoramento dos conjuntos de dados. Por fim, é essencial saber como moldar os dados para criar o formato de saída correto. Veremos esses tópicos se repetirem no contexto de várias análises ao longo do livro. O próximo capítulo, sobre análise de séries temporais, inicia nossa jornada em técnicas específicas de análise.

## CAPÍTULO 3

# Análise de séries temporais

Agora que abordei SQL e bancos de dados e as principais etapas na preparação de dados para análise, é hora de nos voltarmos para tipos específicos de análise que podem ser feitos com SQL. Há um número aparentemente interminável de conjuntos de dados no mundo e, correspondentemente, infinitas maneiras de analisá-los. Neste e nos capítulos seguintes, organizei tipos de análise em temas que, espero, sejam úteis à medida que você desenvolve suas habilidades de análise e SQL. Muitas das técnicas a serem discutidas baseiam-se nas mostradas no [Capítulo 2](#) e depois nos capítulos anteriores à medida que o livro avança. As séries temporais de dados são tão predominantes e tão importantes que começarei a série de temas de análise aqui.

A análise de séries temporais é um dos tipos mais comuns de análise feita com SQL. Uma *série temporal* é uma sequência de medições ou pontos de dados registrados em ordem de tempo, geralmente em intervalos regularmente espaçados. Existem muitos exemplos de dados de séries temporais na vida diária, como a temperatura máxima diária, o valor de fechamento do índice de ações S&P 500 ou o número de passos diários registrados pelo seu rastreador de condicionamento físico. A análise de séries temporais é usada em uma ampla variedade de setores e disciplinas, desde estatísticas e engenharia até previsão do tempo e planejamento de negócios. A análise de séries temporais é uma maneira de entender e quantificar como as coisas mudam ao longo do tempo.

A previsão é um objetivo comum da análise de séries temporais. Como o tempo apenas avança, os valores futuros podem ser expressos em função dos valores passados, enquanto o inverso não é verdadeiro. No entanto, é importante notar que o passado não prevê perfeitamente o futuro. Qualquer número de mudanças nas condições de mercado mais amplas, tendências populares, lançamentos de produtos ou outras grandes mudanças dificultam a previsão. Ainda assim, olhar para dados históricos pode levar a insights, e desenvolver uma série de resultados plausíveis é útil para o planejamento. Enquanto escrevo isso, o mundo está no meio de uma pandemia global de COVID-19, algo que não é visto há 100 anos – anterior a todas as histórias das organizações, exceto as mais duradouras.

Assim, muitas organizações atuais não viram esse evento específico antes, mas existiram por meio de outras crises econômicas, como as que se seguiram à explosão das pontocom e os ataques de 11 de setembro de 2001, bem como a crise financeira global de 2007–2008. Com uma análise cuidadosa e compreensão do contexto, muitas vezes podemos extrair insights úteis.

Neste capítulo, abordaremos primeiro os blocos de construção SQL da análise de séries temporais: sintaxe e funções para trabalhar com datas, carimbos de data e hora e hora. A seguir, apresentarei o conjunto de dados de vendas no varejo usado para exemplos ao longo do restante do capítulo. Segue-se uma discussão de métodos para análise de tendências e, em seguida, abordarei o cálculo das janelas de tempo de rolagem. Em seguida, estão os cálculos de período a período para analisar dados com componentes de sazonalidade. Finalmente, vamos encerrar com algumas técnicas adicionais que são úteis para a análise de séries temporais.

## Manipulações de Date, Datetime e Time

Datas e horas vêm em uma ampla variedade de formatos, dependendo da fonte de dados. Muitas vezes precisamos ou queremos transformar o formato de dados brutos para nossa saída ou realizar cálculos para chegar a novas datas ou partes de datas. Por exemplo, o conjunto de dados pode conter registros de data e hora da transação, mas o objetivo da análise é gerar tendências de vendas mensais. Outras vezes, podemos querer saber quantos dias ou meses se passaram desde um determinado evento. Felizmente, o SQL tem funções poderosas e recursos de formatação que podem transformar praticamente qualquer entrada bruta em praticamente qualquer saída que possamos precisar para análise.

Nesta seção, mostrarei como converter entre fusos horários e, em seguida, aprofundarei a formatação de datas e datas. Em seguida, explorarei as manipulações matemáticas de data e hora, incluindo aquelas que fazem uso de intervalos. Um intervalo é um tipo de dados que contém um intervalo de tempo, como um número de meses, dias ou horas. Embora os dados possam ser armazenados em uma tabela de banco de dados como um tipo de intervalo, na prática raramente vejo isso sendo feito, então falarei sobre intervalos junto com as funções de data e hora com as quais você pode usá-los. Por último, discutirei algumas considerações especiais ao unir ou combinar dados de diferentes fontes.

## Conversões de fuso horário

Compreender o fuso horário padrão usado em um conjunto de dados pode evitar mal-entendidos e erros no processo de análise. Os fusos horários dividem o mundo em regiões norte-sul que observam o mesmo horário. Os fusos horários permitem que diferentes partes do mundo tenham horários semelhantes para o dia e a noite – então, por exemplo, o sol está no céu às 12h, onde quer que você esteja no mundo. As zonas seguem fronteiras irregulares que são tanto políticas quanto geográficas. A maioria tem uma hora de intervalo, mas alguns são deslocados apenas 30 ou 45 minutos, e assim há mais de 30 fusos horários abrangendo o globo. Muitos países distantes do equador também observam o horário de verão em algumas partes do ano, mas há exceções, como nos

Estados Unidos e na Austrália, onde alguns estados observam o horário de verão e outros não. Cada fuso horário tem uma abreviação padrão, como PST para Pacific Standard Time e PDT para Pacific Daylight Time.

Muitos bancos de dados são definidos para o *Tempo Universal Coordenado* (UTC), o padrão global usado para regular os relógios e registrar eventos nesse fuso horário. Ele substituiu o *Greenwich Mean Time* (GMT), que você ainda pode ver se seus dados vierem de um banco de dados mais antigo. O UTC não tem horário de verão, por isso permanece consistente durante todo o ano. Isso acaba sendo bastante útil para análise. Lembro-me de uma vez em que um gerente de produto em pânico me pediu para descobrir por que as vendas em um determinado domingo caíram tanto em comparação com o domingo anterior. Passei horas escrevendo consultas e investigando possíveis causas antes de finalmente descobrir que nossos dados foram registrados no horário do Pacífico (PT). O horário de verão começou no início da manhã de domingo, o relógio do banco de dados avançou 1 hora e o dia tinha apenas 23 horas em vez de 24 e, portanto, as vendas pareciam cair. Meio ano depois, tivemos um dia correspondente de 25 horas, quando as vendas pareciam extraordinariamente altas.



Muitas vezes, os carimbos de data/hora no banco de dados não são codificados com o fuso horário e você precisará consultar a fonte ou o desenvolvedor para descobrir como seus dados foram armazenados. O UTC tornou-se mais comum nos conjuntos de dados que vejo, mas isso certamente não é universal.

Uma desvantagem do UTC, ou realmente de qualquer registro de hora da máquina, é que perdemos informações sobre a hora local para o humano que está fazendo as ações que geraram o evento registrado no banco de dados. Eu gostaria de saber se as pessoas tendem a usar meu aplicativo móvel mais durante o dia de trabalho ou durante as noites e fins de semana. Se meu público estiver agrupado em um fuso horário, não é difícil descobrir isso. Mas se o público abrange vários fusos horários ou é internacional, torna-se uma tarefa de cálculo converter cada hora gravada para seu fuso horário local.

Todos os fusos horários locais têm um deslocamento UTC. Por exemplo, o deslocamento para PDT é UTC – 7 horas, enquanto o deslocamento para PST é UTC – 8 horas. Os carimbos de data/hora nos bancos de dados são armazenados no formato AAAA-MM-DD hh:mi:ss (para anos-meses-dias horas:minutos:segundos). Os carimbos de data/hora com o fuso horário têm uma informação adicional para o deslocamento UTC, expresso como um número positivo ou negativo. Conversão de um fuso horário para outro pode ser realizada com `at time zone` seguido pela abreviação do fuso horário de destino. Por exemplo, podemos converter um carimbo de data/hora em UTC (deslocamento – 0) para 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 de banco de dados, permitindo que essa conversão seja dinâmico para o conjunto de dados. Alguns bancos de dados têm uma função `convert_timezone` ou `convert_tz` que funciona de forma semelhante. Um argumento é o fuso horário do resultado e o outro argumento é o fuso horário do qual converter:

```
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 o nome exato e a ordem do fuso horário de destino e os argumentos de carimbo de data/hora de origem. Muitos bancos de dados contêm uma lista de fusos horários e suas abreviações em uma tabela de sistema. Alguns comuns são vistos na [Tabela 3-1](#). Estes podem ser consultados com `SELECT * FROM` o nome da tabela. A Wikipedia também tem uma lista útil de [abreviações de fuso horário padrão e seus deslocamentos UTC](#).

*Tabela 3-1. Tabelas do sistema de informações de fuso horário em bancos de dados comuns*

Postgres	<code>pg_timezone_names</code>
MySQL	<code>mysql.time_zone_names</code>
SQL Server	<code>sys.time_zone_info</code>
Redshift	<code>pg_timezone_names</code>

Os fusos horários são uma parte inata do trabalho com carimbos de data/hora. Com funções de conversão de fuso horário, é possível mover-se entre o fuso horário em que os dados foram gravados e qualquer outro fuso horário mundial. A seguir, mostrarei uma variedade de técnicas para manipular datas e carimbos de data/hora com SQL.

## Conversões de formato de Date e Timestamp

Datas e timestamps são fundamentais para a análise de séries temporais. Devido à grande variedade de maneiras pelas quais 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 das conversões mais comuns e como realizá-las com SQL: alterar o tipo de dados, extrair partes de uma data ou carimbo de data/hora e criar um carimbo de data ou hora a partir de partes. Vou começar apresentando algumas funções úteis que retornam a data e/ou hora atuais.

Retornar a data ou hora atual é uma tarefa de análise comum — por exemplo, incluir um carimbo de data/hora para o conjunto de resultados ou usar na matemática de datas, abordada na próxima seção. A data e a hora atuais são chamadas de *hora do sistema* e, embora seja fácil retorná-las com o SQL, existem algumas diferenças de sintaxe entre os bancos de dados.

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

```
SELECT current_date;
```

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

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

Finalmente, existem funções para retornar apenas a parte do timestamp da hora atual do sistema. Novamente, consulte a documentação ou experimente para descobrir quais funções usar com seu banco de dados:

```
current_time  
localtime  
timeofday()
```

SQL possui várias funções para alterar o formato de datas e horas. Para reduzir a granularidade de um timestamp, use a função `date_trunc`. O primeiro argumento é um valor de texto que indica o nível do período de tempo para truncar o carimbo de data/hora no segundo argumento. 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 variam de microssegundos a milênios, proporcionando muita flexibilidade. Bancos de dados que não suportam `date_trunc`, como 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 de período padrão

Time period arguments
microsecond
millisecond
second
minute
hour
day
week
month
quarter
year
decade
century
millennium

Em vez de retornar datas ou carimbos de data/hora, às vezes nossa análise pede partes de datas ou horas. Por exemplo, podemos querer agrupar as vendas por mês, dia da semana ou hora do dia.

O SQL fornece algumas funções para retornar apenas a parte da data ou do carimbo de data/hora necessários. Datas e carimbos de data/hora geralmente são intercambiáveis, exceto quando a solicitação é para retornar uma parte do tempo. Nesses casos, é claro que o tempo é necessário.

A função `date_part` recebe um valor de texto 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 parte decimal; dependendo de suas necessidades, você pode querer converter o valor para um tipo de dados inteiro:

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

Outra função que funciona de forma semelhante é `extract`, que recebe um nome de parte 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 observe que a parte solicitada deve corresponder às unidades do intervalo. Assim, por exemplo, solicitar dias de um intervalo indicado em dias retorna o valor esperado de 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 indicado em meses retorna um valor de 0.0:

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

date_part
-----
0.0
```



Uma lista completa de partes de datas pode ser encontrada na documentação do seu banco de dados ou pesquisando online, mas algumas das mais comuns são "dia", "mês" e "ano" para datas e "segundo", "minuto" e "hora" para carimbos de data/hora.

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

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



Se você encontrar carimbos de data/hora armazenados como épocas Unix (o número de segundos decorridos desde 1º de janeiro de 1970, às 00:00:00 UTC), poderá convertê-los em carimbos de data/hora usando a função `to_timestamp`.

Às vezes, a análise exige a criação de uma data a partir de peças de diferentes fontes. Isso pode ocorrer quando os valores de ano, mês e dia são armazenados em colunas diferentes no banco de dados.

Também pode ser necessário quando as partes foram analisadas fora do texto, um tópico que abordarei com mais profundidade no [Capítulo 5](#).

Uma maneira simples de criar um carimbo de data/hora a partir de componentes de data e hora separados é concatená-los com um sinal de mais (+):

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

Uma data pode ser montada usando a função `make_date`, `makedate`, `date_from_parts` ou `datefromparts`. Eles são equivalentes, mas bancos de dados diferentes nomeiam as funções de maneira diferente. A função recebe argumentos para as partes do ano, mês e 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 nomes de campo de referência e devem ser inteiros. Ainda outra maneira de montar uma data ou carimbo de data/hora é concatenar os valores e depois converter 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
```

SQL tem várias maneiras de formatar e converter datas e carimbos de data/hora e recuperar datas e horas do sistema. Na próxima seção, começarei a usá-los na matemática de datas.

## Date Math

SQL nos permite fazer várias operações matemáticas em datas. Isso pode ser surpreendente, pois, estritamente falando, 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 uma variedade de tarefas de análise.

Por exemplo, podemos usá-lo para encontrar a idade ou tempo de serviço de um cliente, quanto tempo decorreu entre dois eventos e quantas coisas ocorreram em uma janela de tempo.

A matemática de datas envolve dois tipos de dados: as próprias datas e os intervalos. Precisamos do conceito de intervalos porque os componentes de data e hora não se comportam como números 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 são 12 horas. Intervalos nos permitem mover suavemente entre unidades de tempo. Os intervalos são de dois tipos: intervalos ano-mês e intervalos diurnos. Começaremos com algumas operações que retornam valores inteiros e, em seguida, passaremos para funções que funcionam com ou retornam intervalos.

Primeiro, vamos encontrar os dias decorridos entre duas datas. Existem várias maneiras de fazer isso no SQL. A primeira maneira é usando um operador matemático, o sinal de menos (-):

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

Retorna o número de dias entre essas duas datas. Observe que a resposta é 30 dias e não 31. O número de dias inclui apenas um dos endpoints. Subtrair as datas ao contrário também funciona e retorna um intervalo de -30 dias:

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

Encontrar a diferença entre duas datas também pode ser feito com a função `dateiff`. O Postgres não oferece suporte, mas muitos outros bancos de dados populares o fazem, incluindo SQL Server, Redshift e Snowflake, e é bastante útil, principalmente quando o objetivo é retornar um intervalo diferente do número de dias. A função recebe três argumentos — as unidades do 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)
```

Então nosso exemplo anterior ficaria assim:

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

Também podemos encontrar o número de meses entre duas datas, e o banco de dados fará as contas corretas, mesmo que os comprimentos dos meses sejam 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 usando a função `age`, que calcula o intervalo entre duas datas:

```
SELECT age(date('2020-06-30'),date('2020-01-01'));
```

```
age
```

```
-----
```

```
5 mons 29 days
```

Podemos então encontrar o componente de número de meses 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 encontrar o tempo decorrido entre elas é bastante poderoso. Adicionar datas não funciona da mesma maneira. Para fazer adição com datas, precisamos alavancar intervalos ou funções especiais.

Por exemplo, podemos adicionar sete dias a uma data adicionando 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 exigem a sintaxe de intervalo e, em vez disso, convertem automaticamente o número fornecido em dias, embora geralmente é uma boa prática usar a notação de intervalo, tanto para compatibilidade entre bancos de dados quanto para facilitar a leitura do código:

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

```
new_date
```

```
-----
```

```
2020-06-08 00:00:00
```

Se você quiser adicionar uma unidade de tempo diferente, use a notação de intervalo com meses, anos, horas, ou outra data ou período de tempo. Observe que isso também pode ser usado para subtrair intervalos de datas usando um “`-`” em vez de um “`+`”.

Muitos, mas nem todos os bancos de dados têm uma função `date_add` ou `dateadd` que recebe o intervalo desejado, um valor e a data inicial e faz as contas:

```
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 experimente as consultas, para descobrir a sintaxe e as funções disponíveis e apropriadas para o seu projeto.

Qualquer uma dessas formulações pode ser usada na cláusula `WHERE` além da cláusula `SELECT`. Por exemplo, podemos filtrar para registros que ocorreram há pelo menos três meses:

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

Eles também podem ser usados em condições `JOIN`, mas observe que o desempenho do banco de dados geralmente será mais lento quando a condição `JOIN` contiver um cálculo em vez de uma igualdade ou desigualdade entre datas.

O uso de matemática de datas é comum na análise com SQL, tanto para encontrar o tempo decorrido entre datas ou carimbos de data/hora quanto para calcular novas datas com base em um intervalo de uma data conhecida. Existem várias maneiras de encontrar o tempo decorrido entre duas datas, adicionar intervalos a datas e subtrair intervalos de datas. Em seguida, vamos nos voltar para as manipulações de tempo, que são semelhantes.

## Matemática do Tempo

A matemática do tempo é menos comum em muitas áreas de análise, mas pode ser útil em algumas situações. Por exemplo, podemos querer saber quanto tempo leva para um representante de suporte atender uma chamada telefônica em uma central de atendimento ou responder a um e-mail solicitando assistência. Sempre que o tempo decorrido entre dois eventos for inferior a um dia, ou quando o arredondamento do resultado para um número de dias não fornecer informações suficientes, a manipulação do tempo entra em ação. A matemática do tempo funciona de forma semelhante à matemática da data, aproveitando os intervalos. Podemos adicionar intervalos de tempo aos tempos:

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

new\_time

-----

08:00:00

Podemos subtrair intervalos de times:

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

Também podemos subtrair tempos, resultando em um intervalo:

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

Os tempos, ao contrário das datas, podem ser multiplicados:

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

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

```
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 usam valores constantes, mas você também pode incluir nomes de campos de banco de dados ou cálculos na consulta SQL para tornar os cálculos dinâmicos. Em seguida, discutirei considerações de datas especiais a serem lembradas ao combinar conjuntos de dados de diferentes sistemas de origem.

## Unindo Dados de Diferentes Fontes

Combinar dados de diferentes fontes é um dos casos de uso mais atraentes para um data warehouse. No entanto, sistemas de origem diferentes podem gravar datas e horas em formatos diferentes ou fusos horários diferentes ou até mesmo estar um pouco desligados devido a problemas com a hora do relógio interno do servidor. Mesmo as tabelas da mesma fonte de dados podem ter diferenças, embora isso seja menos comum. Reconciliar e padronizar datas e carimbos de data/hora é um passo importante antes de avançar na análise.

Datas e carimbos de data/hora que estão em formatos diferentes podem ser padronizados com SQL. *JOIN* em datas ou incluir campos de data em *UNIONs* geralmente requer que as datas ou timestamps estejam no mesmo formato. No início do capítulo, mostrei técnicas para formatar datas e carimbos de data/hora que servirão bem para esses problemas. Tome cuidado com os fusos horários ao combinar dados de diferentes fontes. Por exemplo, um banco de dados interno pode usar o horário UTC, mas os dados de terceiros podem estar em um fuso horário local. Vi dados provenientes de software como serviço (SaaS) que foram registrados em vários horários locais. Observe que os próprios valores de carimbo de data/hora não terão necessariamente o fuso horário incorporado. Pode ser necessário consultar a documentação do fornecedor e converter os dados para UTC se o restante dos dados estiver armazenado dessa maneira. Outra opção é armazenar o fuso horário em um campo para que o valor do carimbo de data/hora possa ser convertido conforme necessário.

Outra coisa a ser observada ao trabalhar com dados de fontes diferentes são os carimbos de data/hora ligeiramente fora de sincronia. Isso pode acontecer quando os carimbos de data/hora são gravados de dispositivos clientes — por exemplo, de um laptop ou celular em uma fonte de dados e um servidor na outra fonte de dados. Certa vez, vi uma série de resultados de experimentos serem calculados incorretamente porque o dispositivo móvel cliente que registrou a ação de um usuário foi deslocado em alguns minutos do 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, então alguns eventos foram excluídos inadvertidamente. Uma correção para algo assim é relativamente simples: em vez de filtrar por timestamps de ação maiores do que o timestamp do grupo de tratamento, permita que eventos dentro de um curto intervalo ou janela de tempo antes do timestamp do tratamento sejam incluídos nos resultados. Isso pode ser feito com uma cláusula *BETWEEN* e date math, como visto na última seção.

Ao trabalhar com dados de aplicativos móveis, preste atenção especial se os carimbos de data/hora representam quando a ação aconteceu no dispositivo *ou* quando o evento chegou ao banco de dados. A diferença pode variar de insignificante até dias, dependendo se o aplicativo móvel permite o uso offline e de como ele lida com o envio de dados durante períodos de baixa intensidade de sinal. Os dados de aplicativos móveis podem chegar atrasados ou podem chegar ao banco de dados dias após a ocorrência no dispositivo. Datas e carimbos de data/hora também podem ser corrompidos no caminho, e você pode ver aqueles que estão impossivelmente distantes no passado ou no futuro como resultado.

Agora que mostrei como manipular datas, datas e horas alterando os formatos, convertendo fusos horários, realizando cálculos de data e trabalhando em conjuntos de dados de diferentes fontes, estamos prontos para ver alguns exemplos de séries temporais. Primeiro, apresentarei o conjunto de dados para exemplos no restante do capítulo.

## O conjunto de dados de vendas no varejo

Os exemplos no restante deste capítulo usam um conjunto de dados de vendas mensais no varejo dos EUA do [Monthly Retail Trade Report: Retail and Food Services Sales: Excel \(1992– present\)](#) <https://www.census.gov/retail/index.html#mrts>, disponível no site [Census.gov](#). Os dados deste relatório são usados como um indicador econômico para entender as tendências nos padrões de gastos do consumidor nos EUA. Embora os números do produto interno bruto (PIB) sejam publicados trimestralmente, esses dados de vendas no varejo são publicados mensalmente, por isso também são usados para ajudar a prever o PIB. Por essas duas razões, os números mais recentes geralmente são cobertos pela imprensa de negócios quando são divulgados.

Os dados vão de 1992 a 2020 e incluem vendas totais e detalhes de subcategorias de vendas no varejo. Ele contém números não ajustados e ajustados sazonalmente. Este capítulo utilizará os números não ajustados, pois um dos objetivos é analisar a sazonalidade. Os números de vendas estão em milhões de dólares americanos. O formato original do arquivo é um arquivo Excel, com uma guia para cada ano e com os meses como colunas. O [site do GitHub](#) para este livro [https://github.com/cathytnanimura/sql\\_book/tree/master/Chapter%203%20Time%20Series%20Analysis](https://github.com/cathytnanimura/sql_book/tree/master/Chapter%203%20Time%20Series%20Analysis) tem os dados em um formato que é mais fácil de importar para um banco de dados, juntamente com o código específico para importação no Postgres. [Figura 3-1](#) mostra 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 fun, 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	Suppressed	(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 no varejo dos EUA*

## Tendência dos dados

Com dados de séries temporais, muitas vezes queremos procurar tendências nos dados. Uma tendência é simplesmente a direção na qual os dados estão se movendo. Pode estar subindo ou aumentando ao longo do tempo, ou pode estar descendo ou diminuindo ao longo do tempo. Pode permanecer mais ou menos plano, ou pode haver tanto ruído, ou movimento para cima e para baixo, que é difícil determinar uma tendência. Esta seção abordará várias técnicas para dados de séries temporais de tendências, desde tendências simples para gráficos até a comparação de componentes de uma tendência, usando cálculos de porcentagem do total para comparar partes com o todo e, finalmente, indexação para ver a alteração percentual de um período de tempo de referência.

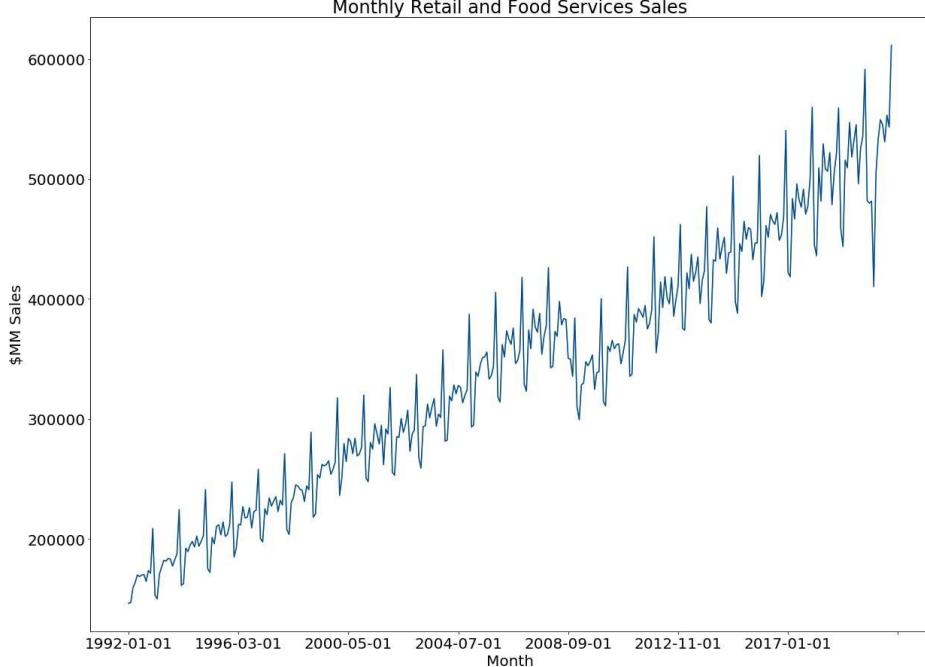
### Tendências Simples

A criação de uma tendência pode ser um passo na criação de perfis e na compreensão dos dados, ou pode ser o resultado final. O conjunto de resultados é uma série de datas ou carimbos de data/hora e um valor numérico. Ao representar graficamente uma série temporal, as datas ou carimbos de data/hora se tornarão o eixo x e o valor numérico será o eixo y. Por exemplo, podemos verificar a tendência das vendas totais de varejo e serviços de alimentação 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 representados graficamente na [Figura 3 -2](#).



*Figura 3-2. Tendência das vendas mensais de varejo e serviços de alimentação*

Esses dados claramente têm alguns padrões, mas também têm algum ruído. Transformar os dados e agregar no nível anual pode nos ajudar a obter uma melhor compreensão. Primeiro, vamos usar a função `date_part` para retornar apenas o ano do campo `sales_month` e depois `sum` a `sales`. Os resultados são filtrados para “Vendas de varejo e serviços de alimentação, total” `kind_of_business` 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
...
...
```

Depois de representar graficamente esses dados, como na [Figura 3-3](#), agora temos uma série temporal mais suave que geralmente está aumentando ao longo do tempo, como seria de esperar, uma vez que os valores de vendas não são ajustados pela inflação. As vendas de todos os serviços de varejo e alimentação caíram em 2009, durante a crise financeira global.

Depois de crescer a cada ano ao longo da década de 2010, as vendas ficaram estáveis em 2020 em relação a 2019, devido ao impacto da pandemia de COVID-19.

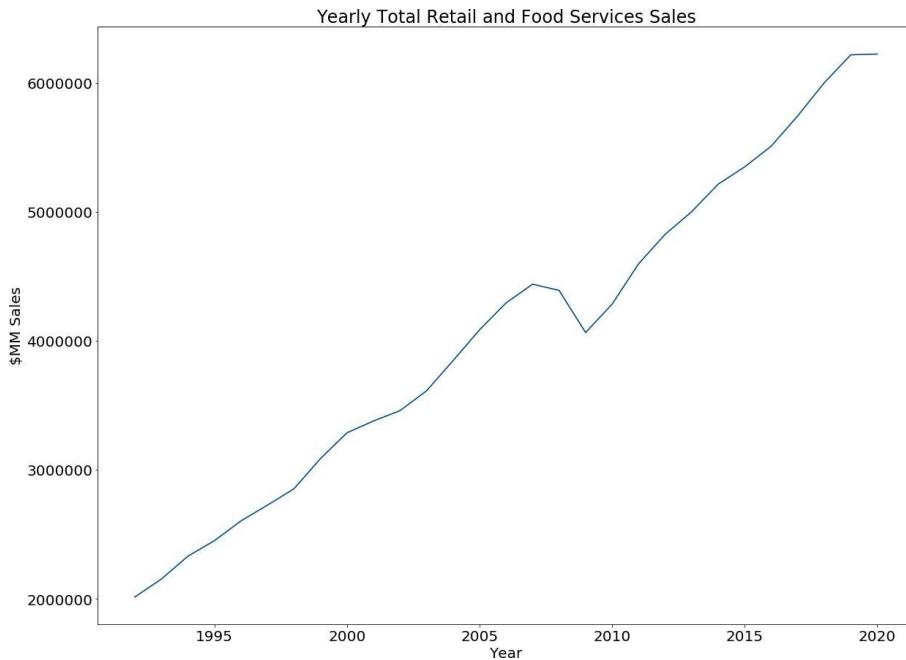


Figura 3-3. Tendência das vendas totais anuais de varejo e serviços de alimentação

A representação gráfica de dados de séries temporais em diferentes níveis de agregação, como semanal, mensal ou anual, é uma boa maneira de entender as tendências. Esta etapa pode ser usada para simplesmente traçar o perfil dos dados, mas também pode ser a saída final, dependendo dos objetivos da análise. Em seguida, vamos usar SQL para comparar componentes de uma série temporal.

## Comparando componentes

Muitas vezes, os conjuntos de dados contêm não apenas uma única série temporal, mas várias fatias ou componentes de um total no mesmo intervalo de tempo. Comparar essas fatias geralmente revela padrões interessantes. No conjunto de dados de vendas no varejo, existem valores para vendas totais, mas também várias subcategorias. Vamos comparar a tendência anual de vendas de algumas categorias associadas a atividades de lazer: livrarias, lojas de artigos esportivos e lojas de hobby. Esta consulta adiciona `kind_of_business` na cláusula `SELECT` e, como é outro atributo em vez de uma agregação, adiciona-o à cláusula `GROUP BY` também:

```
SELECT date_part('year',sales_month) as sales_year
,kind_of_business
,sum(sales) as sales
FROM retail_sales
```

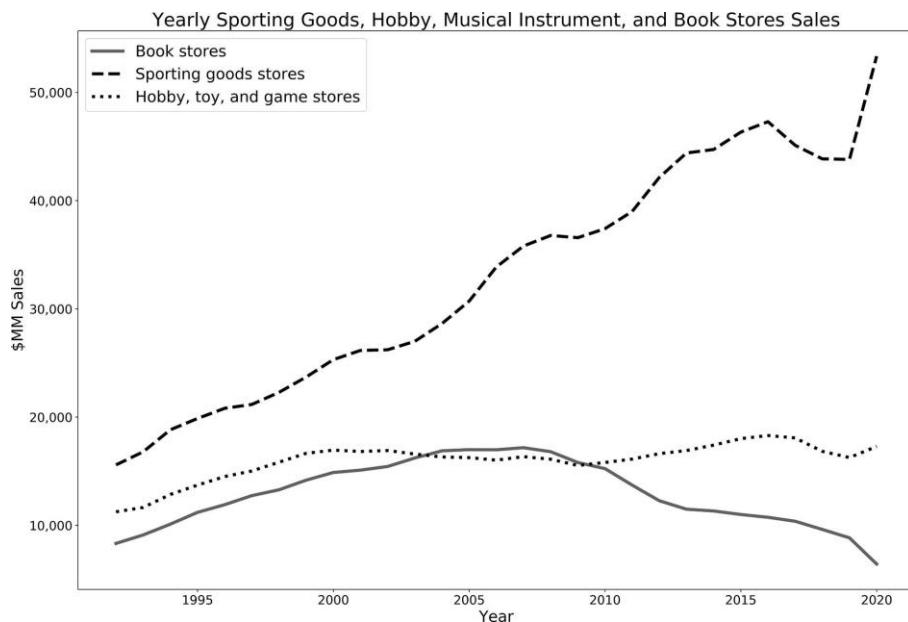
```

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 graficamente na [Figura 3-4](#). As vendas nos varejistas de artigos esportivos começaram as mais altas entre as três categorias e cresceram muito mais rápido durante o período e, no final da série histórica, essas vendas foram substancialmente maiores. As vendas nas lojas de artigos esportivos começaram a cair em 2017, mas tiveram uma grande recuperação em 2020. As vendas nas lojas de hobby, brinquedos e jogos ficaram relativamente estáveis nesse período, com uma ligeira queda em meados dos anos 2000 e outro ligeiro declínio antes de uma recuperação em 2020. As vendas nas livrarias cresceram até meados dos anos 2000 e estão em declínio desde então. Todas essas categorias foram impactadas pelo crescimento dos varejistas online, mas o momento e a magnitude parecem ser diferentes.



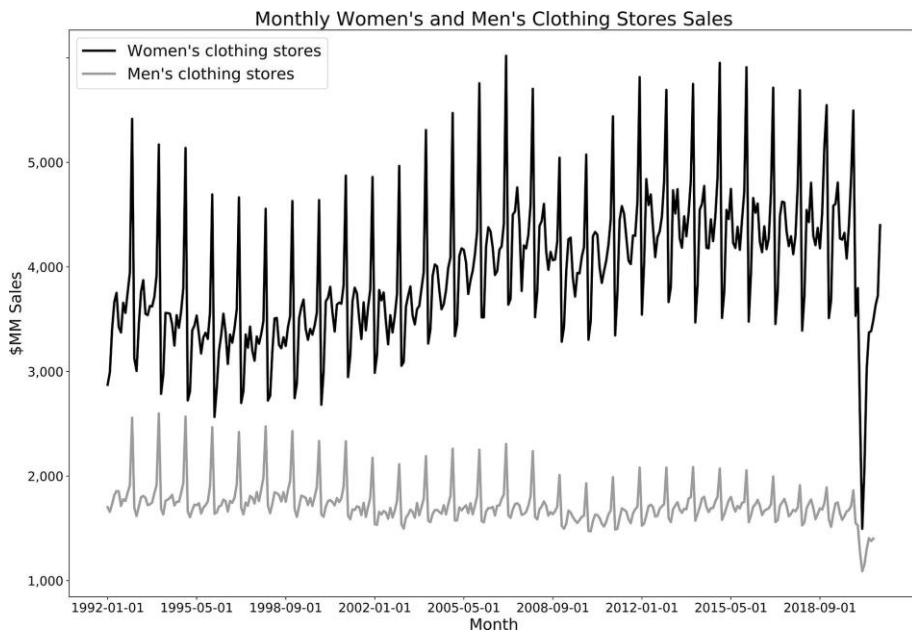
*Figura 3-4. Tendência das vendas anuais no varejo para lojas de artigos esportivos; lojas de passatempos, brinquedos e jogos; e livrarias*

Além de observar tendências simples, podemos querer realizar comparações mais complexas entre partes da série temporal. Para os próximos exemplos, veremos as vendas nas lojas de roupas femininas e nas lojas de roupas masculinas. Observe que, como os nomes contêm apóstrofos, o caractere usado para indicar o início e o fim das strings, precisamos escapá-los com um apóstrofo extra. Isso permite que o banco de dados saiba que o apóstrofo é parte da string e não o final. Embora possamos considerar adicionar uma etapa em um pipeline de carregamento de dados que remova apóstrofos extras em nomes, eu os deixei aqui como uma demonstração dos tipos de ajustes de código que são frequentemente necessários no mundo real. Primeiro, analisaremos os dados para 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 graficamente na [Figura 3-5](#). As vendas dos varejistas de roupas femininas são muito maiores do que as dos varejistas de roupas masculinas. Ambos os tipos de lojas apresentam sazonalidade, tópico que abordarei com profundidade em “[Análise com Sazonalidade](#)” na [página 107](#). Ambos tiveram quedas significativas em 2020 devido ao fechamento de lojas e redução nas compras por causa da pandemia COVID-19.



*Figura 3-5. Tendência mensal de vendas em lojas de roupas femininas e masculinas*

Os dados mensais têm padrões intrigantes, mas são barulhentos, então usaremos agregados anuais para os próximos exemplos. Vimos este formato de consulta anteriormente ao acumular vendas totais e vendas para 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 maiores do que nas lojas de roupas masculinas? Na tendência anual mostrada na [Figura 3-6](#), a diferença entre as vendas de homens e mulheres não parece constante, mas estava aumentando durante o início e meados dos anos 2000. As vendas de roupas femininas, em particular, caíram durante a crise financeira global de 2008–2009, e as vendas em ambas as categorias caíram muito durante a pandemia em 2020.



Figura 3-6. Tendência anual de vendas em lojas de roupas femininas e masculinas

No entanto, não precisamos confiar em estimativas visuais. Para obter mais precisão sobre essa lacuna, podemos calcular a lacuna entre as duas categorias, a proporção e a diferença percentual entre elas. Para isso, o primeiro passo é organizar os dados de forma que haja uma única linha para cada mês, com uma coluna para cada categoria. Girando os dados com funções agregadas combinadas com instruções CASE reliza isso:

```

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 este cálculo de bloco de construção, podemos encontrar a diferença, proporção e diferença percentual entre as séries temporais no conjunto de dados. A diferença pode ser calculada subtraindo um valor do outro usando o operador matemático “-”. Dependendo dos objetivos da análise, encontrar a diferença nas vendas dos homens ou encontrar a diferença nas vendas das mulheres pode ser apropriado. Ambos são mostrados 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 da execução da consulta, pois as agregações podem ser adicionadas ou subtraídas umas das outras. Uma subconsulta geralmente é mais legível, mas adiciona mais linhas ao código. Dependendo de quanto longo ou complexo é o restante de sua consulta SQL, você pode preferir colocar o cálculo intermediário em uma subconsulta ou apenas calculá-lo na consulta principal. Aqui está um exemplo sem a subconsulta, subtraindo as vendas masculinas das vendas femininas, com um filtro de cláusula adicionado *WHERE* para remover 2020, já que alguns meses têm valores nulos:<sup>1</sup>

```

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

---

<sup>1</sup> Os pontos de dados de outubro e novembro de 2020 foram suprimidos pelo editor dos dados, devido a preocupações com a qualidade dos dados. A coleta de dados provavelmente se tornou mais difícil devido ao fechamento de lojas durante a pandemia de 2020.

```

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
...

```

**Figura 3-7** mostra que a diferença diminuiu entre 1992 e cerca de 1997, começou um longo aumento por volta de 2001 (com uma breve queda em 2007) e depois ficou mais ou menos estável até 2019.



*Figura 3-7. Diferença anual entre as vendas em lojas de roupas femininas e masculinas*

Vamos continuar nossa investigação e observar a proporção dessas categorias. Usaremos as vendas masculinas como linha de base ou denominador, mas observe que poderíamos usar as vendas da loja feminina com a mesma facilidade:

```

SELECT sales_year
,omens_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
...
...

```



SQL retorna muito decimal dígitos ao realizar a divisão. Em geral, você deve considerar arredondar o resultado antes de apresentar a análise. Use o nível de precisão (número de casas decimais) que conta a história.

Plotar o resultado, mostrado na [Figura 3-8](#), revela que a tendência é semelhante à tendência da diferença, mas enquanto houve uma queda na diferença em 2009, a razão na verdade aumentou.

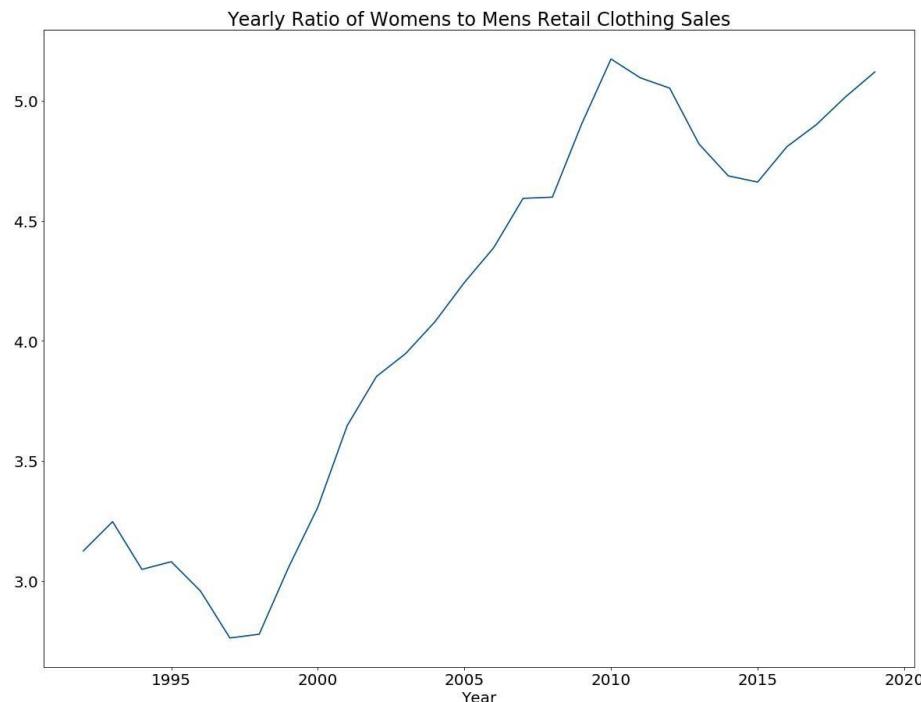


Figura 3-8. Proporção anual de vendas de roupas femininas e masculinas

Em seguida, podemos calcular 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 para esta saída são diferentes das do exemplo anterior, a forma deste gráfico é a mesma do gráfico de razão. A escolha de qual usar depende do seu público e das normas em seu domínio. Todas essas declarações estão corretas: em 2009, as vendas nas lojas de roupas femininas foram US\$28,7 bilhões superiores às vendas nas lojas masculinas; em 2009, as vendas nas lojas de roupas femininas foram 4,9 vezes as vendas nas lojas masculinas; em 2009, as vendas nas lojas femininas foram 390% superiores às vendas nas lojas masculinas. A versão a ser selecionada depende da história que você deseja contar com a análise.

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

## Porcentagem do total de cálculos

Ao trabalhar com dados de séries temporais que possuem várias partes ou atributos que constituem um todo, geralmente é útil analisar a contribuição de cada parte para o todo e se isso mudou ao longo do tempo. A menos que os dados já contenham uma série temporal dos valores totais, precisaremos calcular o total geral para calcular a porcentagem do total para cada linha. Isso pode ser feito com um self-*JOIN*, ou uma função window, que, como vimos no [Capítulo 2](#), é um tipo especial de função SQL que pode referenciar qualquer linha dentro de uma partição especificada da tabela.

Primeiro vou mostrar o método self-*JOIN*. Um self-*JOIN* é sempre que uma tabela é unida a ela mesma. Desde que cada instância da tabela na consulta receba um alias diferente, o banco de dados as tratará como tabelas distintas. Por exemplo, para encontrar a porcentagem de vendas combinadas de roupas masculinas e femininas que cada série representa, podemos *JOIN* `retail_sales`, alias como `a`, para `retail_sales`, alias como `b`, no campo `sales_month`. Nós então *SELECT* os valores de nomes de séries individuais (`kind_of_business`) e `sales` de alias `a`. Então, a partir do pseudônimo `b` nós `sum` o `sales` para ambas as categorias e chame o resultado `total_sales`. Observe que o entre *JOIN* as tabelas no campo `sales_month` cria um JOIN cartesiano parcial, que resulta em duas linhas do alias `b` para cada linha no alias `a`. Agrupando por `a.sales_month`, `a.kind_of_business`, e `a.sales` e agregando `b.sales` retorna exatamente os resultados necessários, no entanto. Na consulta externa, a porcentagem do total para cada linha é calculada dividindo `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, pois o mesmo resultado poderia ser obtido sem ela, mas torna o código um pouco mais fácil de seguir. Uma segunda maneira de calcular a porcentagem do total de vendas para cada categoria é usar a função `sum` e *PARTITION BY* o `sales_month`. Lembre-se de que a cláusula *PARTITION BY* indica a seção da tabela na qual a função deve calcular. A cláusula *ORDER BY* não é necessária nesta função `sum`, pois a ordem de cálculo não importa. Além disso, a consulta não precisa de uma cláusula *GROUP BY*, porque as funções de janela examinam várias linhas, mas não reduzem o número de linhas no conjunto de resultados:

```

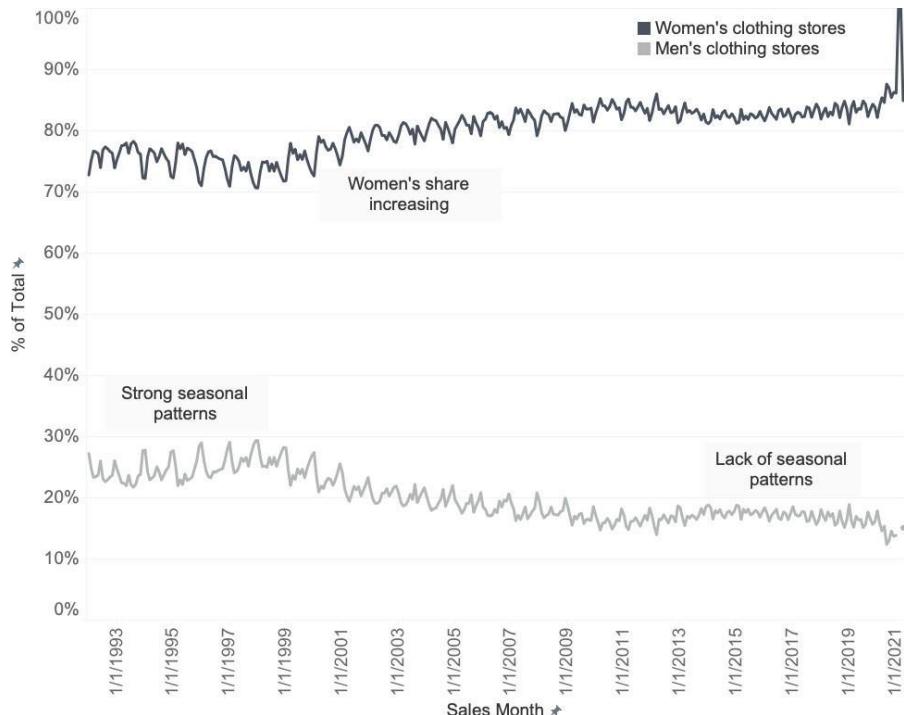
SELECT sales_month, kind_of_business, sales
    ,sum(sales) over (partition by sales_month) as total_sales
    ,sales * 100 / sum(sales) over (partition by sales_month) as pct_total
    FROM retail_sales
    WHERE kind_of_business in ('Men''s clothing stores'
        , 'Women''s clothing stores')
    ;

```

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 gráfica desses dados, como na [Figura 3-9](#), revela algumas tendências interessantes. Primeiro, a partir do final da década de 1990, as vendas das lojas de roupas femininas tornaram-se uma porcentagem crescente do total. Em segundo lugar, no início da série, um padrão sazonal é evidente, onde as vendas masculinas aumentam como porcentagem das vendas totais em dezembro e janeiro.

Na primeira década do século 21, aparecem dois picos sazonais, no verão e no inverno, mas no final da década de 2010, os padrões sazonais são atenuados quase ao ponto da aleatoriedade. Veremos a análise da sazonalidade em maior profundidade mais adiante neste capítulo.



*Figura 3-9. Vendas de lojas de roupas masculinas e femininas como porcentagem do total mensal*

Outra porcentagem do total que podemos querer encontrar é a porcentagem de vendas dentro de um período de tempo mais longo, como a porcentagem de vendas anuais que cada mês representa. Novamente, um self-JOIN ou uma função de janela farão o trabalho. Neste exemplo, usaremos um 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, ampliados para 2019, são mostrados em [Figura 3-10](#). As duas séries temporais acompanham bastante de perto, mas as lojas masculinas tiveram uma porcentagem maior de suas vendas em janeiro do que as lojas femininas. As lojas masculinas tiveram uma queda de verão em julho, enquanto a queda correspondente nas vendas das lojas femininas só ocorreu em setembro.

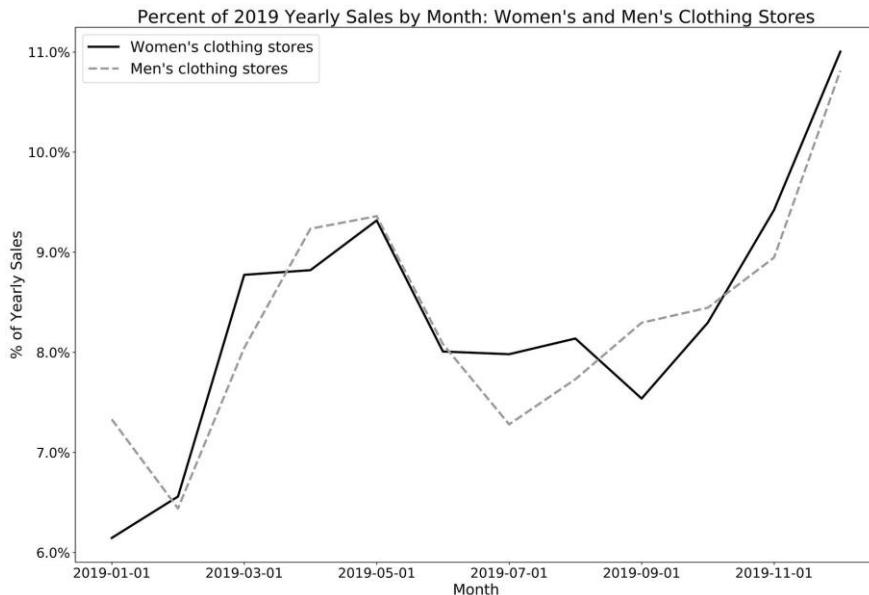


Figura 3-10. Porcentagem das vendas anuais de 2019 para vendas de roupas femininas e masculinas

Agora que mostrei como usar o SQL para a porcentagem do total de cálculos e os tipos de análise que podem ser realizados, passarei a indexar e calcular a alteração percentual ao longo do tempo.

## Indexação para ver a mudança percentual ao longo do tempo

Os valores nas séries temporais geralmente flutuam ao longo do tempo. As vendas aumentam com a crescente popularidade e disponibilidade de um produto, enquanto o tempo de resposta da página da Web diminui com os esforços dos engenheiros para otimizar o código. A indexação de dados é uma maneira de entender as mudanças em uma série temporal em relação a um período base (ponto de partida). Os índices são amplamente utilizados em economia, bem como em ambientes de negócios. Um dos índices mais famosos é o Índice de Preços ao Consumidor (CPI), que rastreia a mudança nos preços dos itens que um consumidor típico compra e é usado para acompanhar a inflação, decidir aumentos salariais e muitas outras aplicações. O IPC é uma medida estatística complexa que usa vários pesos e entradas de dados, mas a premissa básica é direta. Escolha um período base e calcule a variação percentual no valor 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-JOINs. Como exemplo, indexamos as vendas de lojas de roupas femininas para o primeiro ano da série, 1992. A primeira etapa é agregar as `sales` por `sales_year` em uma subconsulta, como fizemos anteriormente. Na consulta externa, a função de janela `first_value` encontra o valor associado à primeira linha na `PARTITION BY`, de acordo com a classificação na cláusula `ORDER BY`. Neste exemplo, podemos omitir a cláusula `PARTITION BY`, porque queremos devolver as vendas `value` para a primeira linha em todo o conjunto de dados 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 esta amostra de dados, podemos verificar visualmente se o valor do índice está definido corretamente no valor de 1992. Em seguida, encontre a porcentagem de alteração deste 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 variação percentual pode ser positiva ou negativa, e veremos que de fato ocorre nesta série temporal. A função de janela `last_value` pode ser substituída por `first_value` nesta consulta. A indexação do último valor em uma série é muito menos comum, no entanto, uma vez que as questões de análise geralmente se relacionam com a mudança de um ponto de partida, em vez de olhar para trás a partir de um ponto final arbitrário; ainda assim, a opção está lá. Além disso, a ordem de classificação pode ser usada para obter a indexação do primeiro ou do último valor alternando entre `ASC` e `DESC`:

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

As funções de janela fornecem muita flexibilidade. Indexação pode ser realizada sem eles por meio de uma série de auto-*JOINS*, embora sejam necessárias mais linhas de código:

```
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 cláusula `JOIN` incomum = 1 1 entre alias `aa` e subconsulta `bb`. Como queremos que o valor `index_sales` seja preenchido para cada linha no conjunto de resultados, não podemos `JOIN` no ano ou em qualquer outro valor, o que 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 resulte em TRUE para criar o `JOIN`. Qualquer outra instrução TRUE, como `on 2 = 2` ou `on 'apples' = 'apples'`, pode ser usada.



Cuidado com os zeros no denominador das operações de divisão como `sales / index_sales` no último exemplo. Os bancos de dados retornam um erro quando encontram a divisão por zero, o que pode ser frustrante. Mesmo quando você acha que um zero no campo do denominador é improvável, é uma boa prática evitar isso dizendo ao banco de dados para retornar um valor padrão alternativo quando encontrar um zero. Isso pode ser feito com uma instrução CASE. Os exemplos nesta seção não têm zeros no denominador, então omitirei este código extra para facilitar a legibilidade.

Para encerrar esta seção, vejamos um gráfico da série temporal indexada para lojas de roupas masculinas e femininas, mostrada na [Figura 3-11](#). O código SQL se parece com:

```

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 de lojas de roupas masculinas e femininas, indexadas às vendas de 1992

É evidente, a partir deste gráfico, que 1992 foi um ponto alto para as vendas em lojas de roupas masculinas. Depois de 1992, as vendas caíram, depois retornaram brevemente ao mesmo nível em 1998 e vêm caindo desde então. Isso é impressionante, pois o conjunto de dados não é ajustado pela inflação, a tendência dos preços subirem ao longo do tempo. As vendas nas lojas de roupas femininas caíram inicialmente em relação aos níveis de 1992, mas voltaram ao nível de 1992 em 2003. Elas aumentaram desde então, com exceção da queda durante a crise financeira que diminuiu as vendas em 2009 e 2010. Uma explicação para essas tendências é que os homens simplesmente diminuíram os gastos com roupas ao longo do tempo, talvez tornando-se menos conscientes da moda em relação às mulheres. Talvez as roupas masculinas simplesmente tenham se tornado menos caras à medida que as cadeias de suprimentos globais diminuíssem os custos. Ainda outra explicação pode ser que os homens mudaram suas compras de roupas de varejistas categorizados como “lojas de roupas masculinas” para outros tipos de varejistas, como lojas de artigos esportivos ou varejistas online.

A indexação de dados de séries temporais é uma técnica de análise poderosa, permitindo-nos encontrar uma variedade de insights nos dados. O SQL é adequado 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 rolantes para encontrar padrões em séries temporais com ruído.

## Rolling Time Windows

Os dados de séries temporais geralmente são ruidosos, um desafio para um de nossos principais objetivos de encontrar padrões. Vimos como a agregação de dados, como mensal para anual, pode suavizar os resultados e torná-los mais fáceis de interpretar. Outra técnica para suavizar os dados são *as janelas de tempo contínuo*, também conhecidas como cálculos móveis, que levam em consideração vários períodos. As médias móveis são provavelmente as mais comuns, mas com o poder do SQL, qualquer função agregada está disponível para análise. As janelas de tempo contínuo são usadas em uma ampla variedade de áreas de análise, incluindo mercados de ações, tendências macroeconômicas e medição de audiência. Alguns cálculos são tão comumente usados que têm seus próprios acrônimos: últimos doze meses (LTM), doze meses à direita (TTM) e acumulado no ano (YTD).

A Figura 3-12 mostra um exemplo de janela de tempo contínuo e um cálculo cumulativo, relativo ao mês de outubro na série temporal.

Month	Sales
Nov	100
Dec	110
Jan	95
Feb	85
Mar	90
Apr	90
May	95
Jun	100
Jul	105
Aug	110
Sep	120
Oct	130

The diagram shows a table of monthly sales data from November to October. A bracket on the left side of the table, spanning from November to October, is labeled "LTM = 1,230". A bracket on the right side of the table, spanning from November to October, is labeled "YTD = 1,020".

Figura 3-12. Exemplo de soma de vendas contínuas de LTM e YTD

Existem várias partes importantes de qualquer cálculo de série temporal contínua. O primeiro é 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 de suavização maior, mas correm o risco de perder a sensibilidade a importantes alterações de curto prazo nos dados. Janelas mais curtas com menos períodos de tempo fazem menos suavização e, portanto, são mais sensíveis a mudanças de curto prazo, mas correm o risco de reduzir muito pouco o ruído.

A segunda parte dos cálculos de séries temporais é a função agregada usada. Como observado anteriormente, as médias móveis são provavelmente as mais comuns.

Somas móveis, contagens, mínimos e máximos também podem ser calculados com SQL. As contagens móveis são úteis nas métricas de população de usuários (consulte a barra lateral a seguir). A movimentação de mínimos e máximos pode ajudar na compreensão dos extremos dos dados, útil para o planejamento de análises.

A terceira parte dos cálculos de séries temporais é escolher o particionamento ou agrupamento dos dados incluídos na janela. A análise pode exigir a redefinição da janela todos os anos. Ou a análise pode precisar de uma série móvel diferente para cada componente ou grupo de usuários. O Capítulo 4 entrará em mais detalhes sobre a análise de coorte de grupos de usuários, onde consideraremos como a retenção e os valores cumulativos, como gastos, diferem entre as populações ao longo do tempo. O particionamento será controlado através do agrupamento, bem como da instrução *PARTITION BY* das funções da janela.

Com essas três partes em mente, passaremos para o código SQL e cálculos para períodos de tempo móveis, continuando com o conjunto de dados de vendas no varejo dos EUA para exemplos.

## Medindo “Usuários ativos”: DAU, WAU e MAU

Muitos aplicativos SaaS de consumidor e alguns B2B usam cálculos de usuários ativos, como usuários ativos diários (DAU), usuários ativos semanais (WAU) e usuários ativos mensais (MAU) para estimar seu público Tamanho. Como cada uma dessas janelas é contínua, elas podem ser calculadas diariamente. Muitas vezes me perguntam qual é a métrica certa ou melhor a ser usada, e minha resposta é sempre “depende”.

A DAU ajuda as empresas com planejamento de capacidade, como estimar quanta carga esperar nos servidores. Dependendo do serviço, no entanto, dados ainda mais detalhados podem ser necessários, como informações de usuários simultâneos por hora ou até mesmo minuto a minuto.

MAU é comumente usado para estimar tamanhos relativos de aplicativos ou serviços. É útil para medir populações de usuários razoavelmente estáveis ou crescentes que têm padrões de uso regular que não são necessariamente diários, como maior uso no fim de semana para produtos de lazer ou maior uso durante a semana para produtos relacionados ao trabalho ou à escola. O MAU não é tão adequado para detectar alterações no churn subjacente de usuários que param de usar um aplicativo. Como um usuário leva 30 dias, a janela mais comum, para passar pelo MAU, um usuário pode estar ausente do produto por 29 dias antes de desencadear uma queda no MAU.

WAU, calculado ao longo de 7 dias, pode ser um meio termo entre DAU e MAU. O WAU é mais sensível a flutuações de curto prazo, alertando as equipes sobre mudanças no churn mais rapidamente do que o MAU, enquanto suaviza as flutuações do dia da semana que são rastreadas pelo DAU. Uma desvantagem da WAU é que ela ainda é sensível a flutuações de curto prazo causadas por eventos como feriados.

## Calculando Rolling Time Windows

Agora que sabemos o que são janelas de tempo contínuo, como são úteis e seus principais componentes, vamos calculá-los usando o conjunto de dados de vendas no varejo dos EUA. Começaremos com o caso mais simples, quando o conjunto de dados contém um registro para cada período que deveria estar na janela e, na próxima seção, veremos o que fazer quando esse não for o caso.

Existem dois métodos principais para calcular uma janela de tempo de rolagem: um self-*JOIN*, que pode ser usado em qualquer banco de dados, e uma função de janela, que como vimos não está disponível em alguns bancos de dados. Em ambos os casos, precisamos do mesmo resultado: uma data e um número de pontos de dados que correspondam ao tamanho da janela à qual aplicaremos uma média ou outra função agregada.

Para este exemplo, usaremos uma janela de 12 meses para obter vendas anuais contínuas, pois os dados estão em um nível mensal de granularidade. Em seguida, aplicaremos uma média para obter uma média móvel de 12 meses das vendas no varejo. Primeiro, vamos desenvolver a intuição para o que entrará no cálculo. Nesta consulta, alias *a* da tabela é a nossa tabela “âncora”, aquela da qual recolhemos as datas. Para começar, analisaremos um único mês, dezembro de 2019. A partir do alias *b*, a consulta reúne os 12 meses individuais de vendas que entrarão na média móvel. Isso é feito com a cláusula *JOIN b.sales\_month between a.sales\_month - interval '11 months' and a.sales\_month*, que cria uma intencionalidade cartesiana *JOIN*:

```

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 as figuras `sales_month` e `sales` de alias `a` são repetidos para cada linha dos 12 meses no window.



Lembre-se de que as datas em uma cláusula *BETWEEN* são inclusivas (ambas serão retornadas no conjunto de resultados). É um erro comum usar 12 em vez de 11 na consulta anterior. Em caso de dúvida, verifique os resultados da consulta intermediária como fiz aqui para garantir que o número pretendido de períodos termine no cálculo da janela.

A próxima etapa é aplicar a agregação — neste caso, `avg`, pois queremos uma média móvel. A `count` de registros retornados do alias `b` é incluída para confirmar que cada linha tem uma média de 12 pontos de dados, uma verificação de qualidade de dados útil. 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
;
  
```

<code>sales_month</code>	<code>sales</code>	<code>moving_avg</code>	<code>records_count</code>
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 graficamente na [Figura 3-13](#). Embora a tendência mensal seja ruidosa, a tendência da média móvel suavizada facilita a detecção de mudanças como o aumento de 2003 a 2007 e a queda subsequente até 2011. Observe que a queda extrema no início de 2020 puxa a média móvel para baixo mesmo depois que as vendas começam a se recuperar no final do ano.

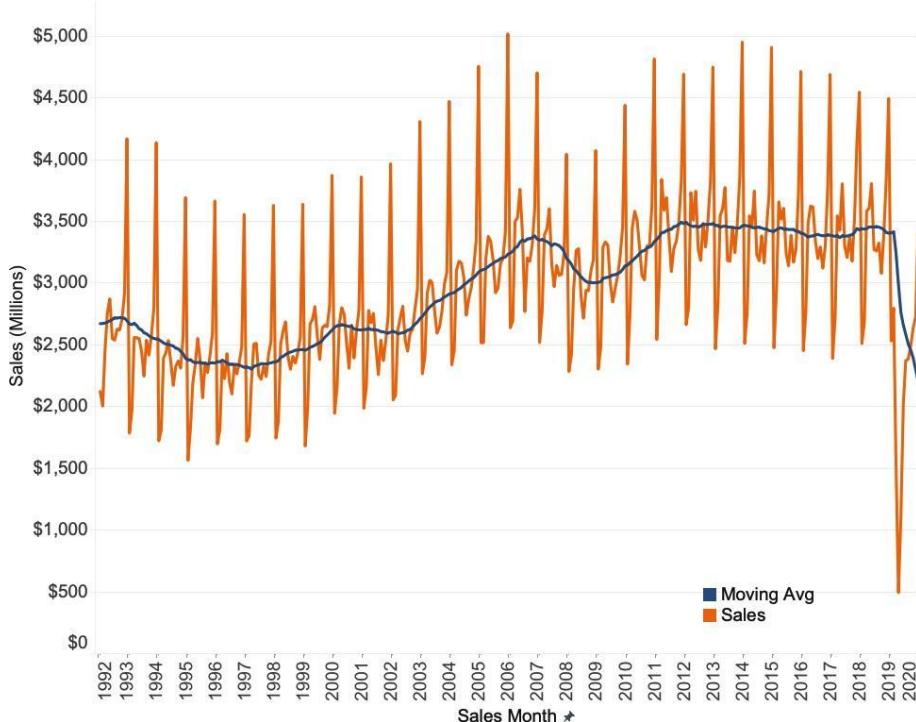


Figura 3-13. Vendas mensais e vendas médias móveis de 12 meses para lojas de roupas femininas



Adicionar o filtro `kind_of_business = 'Women's clothing stores'` a cada alias não é estritamente necessário. Como a consulta usa um *INNER JOIN*, a filtragem em uma tabela filtrará automaticamente a outra. No entanto, a filtragem em ambas as tabelas geralmente faz com que as consultas sejam executadas mais rapidamente, principalmente quando as tabelas são grandes.

As funções de janela são outra maneira de calcular as janelas de tempo de rolagem. Para fazer uma janela rolante, precisamos usar outra parte opcional de um cálculo de janela: a *cláusula frame*. A cláusula frame permite especificar quais registros incluir na janela. Por padrão, todos os registros na partição são incluídos e, em muitos casos, isso funciona bem. No entanto, controlar os registros incluídos em um nível mais refinado é útil para casos como cálculos de janela móvel. A sintaxe é simples e ainda pode ser confusa ao encontrá-la pela primeira vez. A cláusula frame pode ser especificada como:

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

Dentro das chaves há três opções para o tipo de quadro: intervalo, linhas e grupos. Essas são as maneiras pelas quais você pode especificar quais registros incluir no resultado, em relação à linha atual. Os registros são sempre escolhidos da partição atual e seguem o *ORDER BY*. A classificação padrão é crescente (*ASC*), mas pode ser alterada para decrescente (*DESC*). *Rows* é o mais simples e permitirá que você especifique o número exato de linhas que devem ser retornadas. O *intervalo* inclui registros que estão dentro de algum limite de valores relativos à linha atual. Os *grupos* podem ser usados quando há vários registros com o mesmo valor *ORDER BY*, como quando um conjunto de dados inclui várias linhas por mês de vendas, uma para cada cliente.

O *frame\_start* e o *frame\_end* podem ser qualquer um dos seguintes:

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

*Preceding* significa incluir linhas antes da linha atual, de acordo com a classificação *ORDER BY*. A *linha atual* é apenas isso, e *seguir* significa incluir linhas que ocorrem após a linha atual de acordo com a classificação *ORDER BY*. A palavra-chave *UNBOUNDED* significa incluir todos os registros na partição antes ou depois da linha atual. O *deslocamento* é o número de registros, geralmente apenas uma constante inteira, embora um campo ou uma expressão que retorne um inteiro também possa ser usado. As cláusulas de quadro também têm uma opção opcional *frame\_exclusion*, que está além do escopo da discussão aqui. [Figura 3-14](#) mostra um exemplo das linhas que cada uma das opções de moldura de janela selecionará.

The diagram shows a table of monthly sales data from November to October. A specific row for May is highlighted in blue and labeled as the 'Current row'. Brackets on the left side group rows into three categories: 'Unbounded preceding' (Nov-Oct), 'Unbounded following' (Jan-Oct), and the single 'Current row' (May). Brackets on the right side group rows into three groups: '3 preceding' (Nov-Apr), 'Current row' (May), and '3 following' (Jun-Oct).

Month	Sales
Nov	100
Dec	110
Jan	95
Feb	85
Mar	90
Apr	90
May	95
Jun	100
Jul	105
Aug	110
Sep	120
Oct	130

Figura 3-14. Cláusulas de moldura de janela e as linhas que elas incluem

Da partição à ordenação de molduras de janela, as funções de janela têm uma variedade de opções que controlam os cálculos, tornando-as incrivelmente poderosas e adequadas para lidar com cálculos complexos com uma sintaxe relativamente simples. Voltando ao nosso exemplo de vendas no varejo, a média móvel que calculamos usando um self-*JOIN* pode ser realizada com funções de janela em 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
...
...
```

Nesta consulta, a janela ordena as vendas por mês (crescente) para garantir que os registros da janela estejam em ordem cronológica. A cláusula frame são `rows between 11 preceding and current row`, pois sei que tenho um registro para cada mês e quero que os 11 meses anteriores e o mês da linha atual sejam incluídos nos cálculos de média e contagem. A consulta retorna todos os meses, incluindo aqueles que não têm 11 meses anteriores, e talvez querímos filtrá-los colocando essa consulta em uma subconsulta e filtrando por mês ou número de registros na consulta externa.



Embora o cálculo de médias móveis de períodos de tempo anteriores seja comum em muitos contextos de negócios, as funções de janela SQL são flexíveis o suficiente para incluir também períodos de tempo futuros. Eles também podem ser usados em qualquer cenário em que os dados tenham alguma ordenação, não apenas na análise de séries temporais.

O cálculo de médias móveis ou outras agregações móveis pode ser realizado com self-*JOINS* ou funções de janela quando existirem registros no conjunto de dados para cada período de tempo na 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 deles terá desempenho ou dar conselhos gerais sobre qual usar.

Vale a pena tentar os dois métodos e prestar atenção em quanto tempo leva para retornar os resultados da consulta; em seguida, faça o que parece funcionar mais rápido como sua escolha padrão. Agora que vimos como calcular janelas de tempo de rolagem, mostrarei como calcular janelas de rolagem com conjuntos de dados esparsos.

## Rolling Time Windows com Sparse Data

Os conjuntos de dados no mundo real podem não conter um registro para cada período de tempo que cai dentro da janela. A medição de interesse pode ser sazonal ou intermitente por natureza. Por exemplo, os clientes podem voltar a comprar em um site em intervalos irregulares, ou um determinado produto pode entrar e sair do estoque. Isso resulta em dados esparsos.

Na última seção, mostrei como calcular uma janela rolante com um self-JOIN e um intervalo de data na cláusula JOIN. Você pode estar pensando que isso coletará todos os registros dentro da janela de 12 meses, estejam todos no conjunto de dados ou não, e você estará correto. O problema com essa abordagem ocorre quando não há registro para o mês (ou dia ou ano) em si. Por exemplo, imagine que eu queira calcular as vendas contínuas de 12 meses para cada modelo de sapato que minha loja estoca em dezembro de 2019. No entanto, alguns dos sapatos ficaram sem estoque antes de dezembro e, portanto, não têm registros de vendas em naquele mês. O uso de uma função self-JOIN ou de janela retornará um conjunto de dados de vendas contínuas para todos os sapatos vendidos em dezembro, mas os dados não incluirão os sapatos que ficaram sem 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 garantir 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. Como os `retail_sales` incluem linhas para todos os meses, simulei um conjunto de dados esparsos adicionando uma subconsulta para filtrar a tabela apenas para `sales_months` de janeiro e julho (1 e 7). Vejamos os resultados quando JOINed para `date_dim`, mas antes da agregação, para desenvolver a intuição sobre os dados antes de aplicar os 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 para fevereiro e março `dates` além de janeiro, embora 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 associação a todas as datas. A construção desta consulta é muito semelhante à consulta self-JOIN na última seção, com a cláusula `JOIN on b.sales_month between a.date - interval '11 months' and a.date` da mesma forma que a cláusula `JOIN` no self-JOIN. Agora que desenvolvemos uma compreensão do 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 acima, o conjunto de resultados inclui uma linha para cada mês; no entanto, a média móvel permanece constante até que um novo ponto de dados (neste caso, janeiro ou julho) seja adicionado. Cada média móvel consiste em dois pontos de dados subjacentes. Em um caso de uso real, é provável que o número de pontos de dados subjacentes varie. Para retornar o valor do mês atual ao usar uma dimensão de dados, uma agregação com uma instrução CASE pode ser usada — por exemplo:

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

As condições dentro da instrução CASE pode ser alterado para retornar qualquer um dos registros subjacentes que a análise requer por meio do uso de igualdade, desigualdade ou deslocamentos com matemática de data. Se uma dimensão de data não estiver disponível em seu banco de dados, outra técnica poderá ser usada para simular uma. Em uma subconsulta, *SELECIONE* as *DISTINCT* datas necessárias e *JOIN* para sua tabela da mesma forma que 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, esteja ou não relacionada à tabela a partir da qual você deseja calcular a agregação contínua.

O cálculo de janelas de tempo de rolagem com dados esparsos ou ausentes pode ser feito em SQL com aplicação controlada de JOINs cartesianos. Em seguida, veremos como calcular valores cumulativos que são frequentemente usados em análise.

## Calculando valores cumulativos

Cálculos de janela contínua, como médias móveis, normalmente usam janelas de tamanho fixo, como 12 meses, como vimos na seção anterior. Outro tipo de cálculo comumente usado é o valor cumulativo, como YTD, quarter-to-date (QTD) e month-to-date (MTD). Em vez de uma janela de comprimento fixo, eles contam com 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 total de vendas no acumulado do ano em cada mês. Outras análises podem exigir um YTD médio mensal ou um YTD máximo mensal, que pode ser realizado trocando `sum` por `avg` ou `max`. A janela é reiniciada de acordo com a cláusula *PARTITION BY*, neste caso o ano do mês de vendas. A cláusula *ORDER BY* normalmente inclui um campo de data na análise de série temporal. Omitir o *ORDER BY* pode levar a resultados incorretos devido à maneira como os dados são classificados na tabela subjacente, portanto, é uma boa ideia incluí-lo 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 `sales` para aquele mês, e o total corrente `sales_ytd`. A série começa em 1992 e é reiniciada em janeiro de 1993, como será para todos os anos no conjunto de dados. Os resultados para os anos de 2016 a 2020 estão representados graficamente na [Figura 3-15](#). Os primeiros quatro anos mostram padrões semelhantes ao longo do ano, mas é claro que 2020 parece muito diferente.

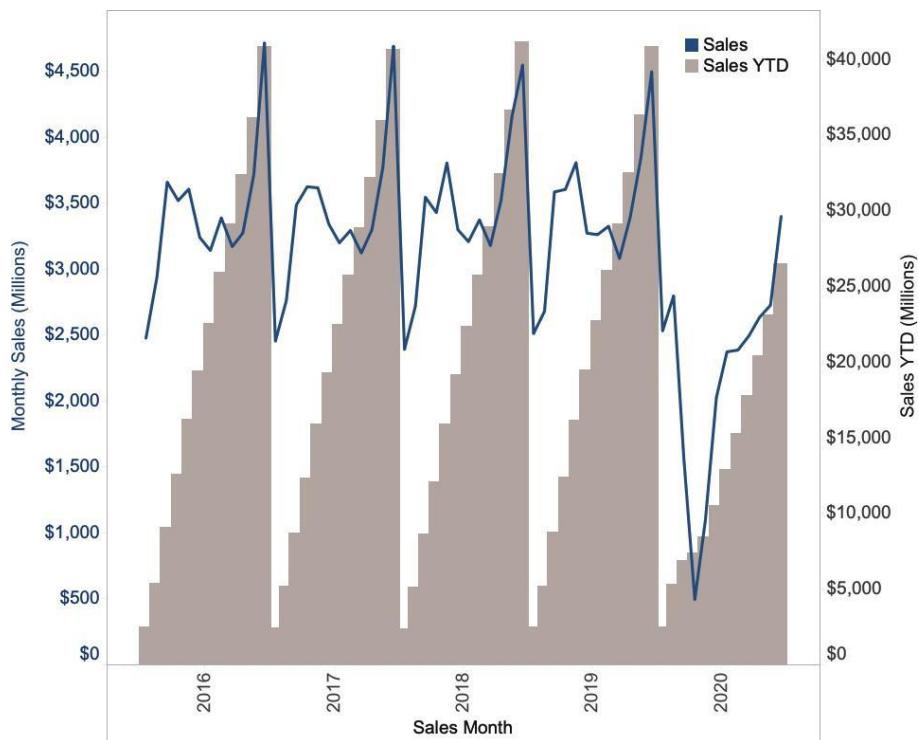


Figura 3-15. Vendas mensais e vendas anuais cumulativas para lojas de roupas femininas

Os mesmos resultados podem ser alcançados sem as funções de janela, usando um self-*JOIN* cartesiano *JOIN*. Neste exemplo, os dois aliases de tabela são *JOINed* no ano do *sales\_month* para garantir que os valores agregados sejam para o mesmo ano, redefinindo a cada ano. A cláusula *JOIN* também especifica que os resultados devem incluir *sales\_months* do alias *b* que são menores ou iguais a *sales\_month* no alias *a*. Em janeiro de 1992, apenas a linha de janeiro de 1992 do alias *b* atende a esse critério; em fevereiro de 1992, tanto em janeiro quanto em fevereiro de 1992; 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 para acompanhar exatamente o que eles estão calculando quando você estiver familiarizado com a sintaxe. Geralmente, há mais de uma maneira de abordar um problema no SQL, e as janelas de tempo contínuo são um bom exemplo disso. Acho útil conhecer várias abordagens, porque de vez em quando me deparo com um problema complicado que, na verdade, é melhor resolvido com uma abordagem que parece menos eficiente em outros contextos. Agora que abordamos as janelas de tempo rolantes, passaremos para nosso tópico final na análise de séries temporais com SQL: sazonalidade.

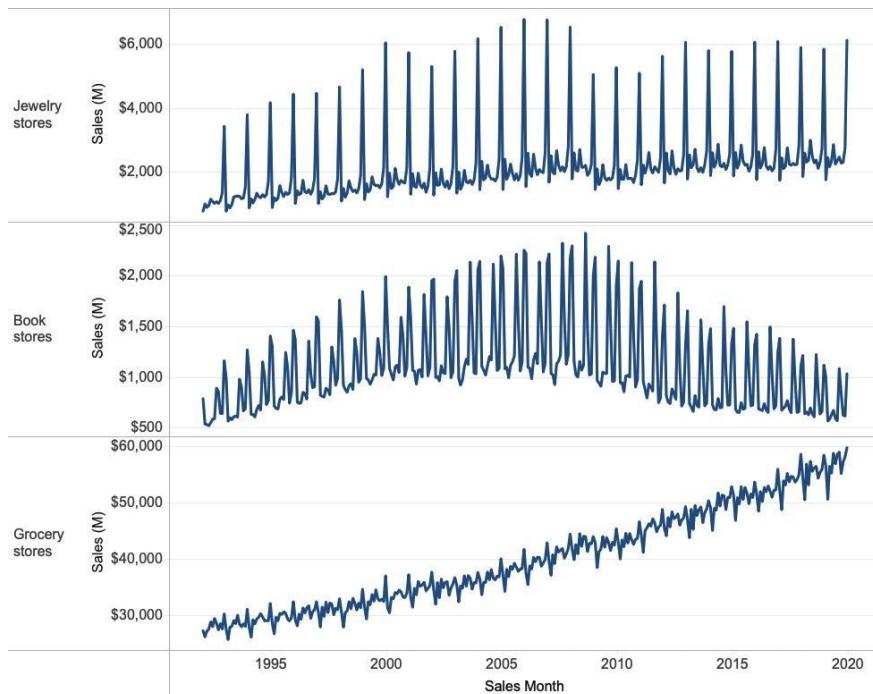
## Analisando com Sazonalidade

*Sazonalidade* é qualquer padrão que se repete em intervalos regulares. Ao contrário de outros ruídos nos dados, a sazonalidade pode ser prevista. A palavra *sazonalidade* traz à mente as quatro estações do ano – primavera, verão, outono, inverno – e alguns conjuntos de dados incluem esses padrões. Os padrões de compras mudam com as estações, desde as roupas e alimentos que as pessoas compram até o dinheiro gasto em lazer e viagens. A temporada de compras de inverno pode ser decisiva para muitos varejistas. A sazonalidade também pode existir em outras escalas de tempo, de anos a minutos. As eleições presidenciais nos Estados Unidos acontecem a cada quatro anos, levando a padrões distintos na cobertura da mídia. A ciclicidade dos dias da semana é comum, pois o trabalho e a escola dominam de segunda a sexta-feira, enquanto as tarefas domésticas e as atividades de lazer dominam o fim de semana. A hora do dia é outro tipo de sazonalidade que os restaurantes experimentam, com corridas na hora do almoço e do jantar e vendas mais lentas no meio.

Para entender se a sazonalidade existe em uma série temporal e em que escala, é útil grafá-la e, em seguida, inspecionar visualmente os padrões. Tente agregar em diferentes níveis, de hora em hora a diária, semanal e mensal. Você também deve incorporar conhecimento sobre o conjunto de dados. Existem padrões que você pode adivinhar com base no que você sabe sobre a entidade ou processo que ela representa? Consulte especialistas no assunto, se disponível.

Vamos dar uma olhada em alguns padrões sazonais no conjunto de dados de vendas no varejo, mostrado na [Figura 3-16](#). As joalherias têm um padrão altamente sazonal, com picos anuais em dezembro relacionados à oferta de presentes de Natal. As livrarias têm dois picos a cada ano: um pico é em agosto, correspondendo ao período de volta às aulas nos Estados Unidos; o outro pico começa em dezembro e dura até janeiro, incluindo o período de presentes de fim de ano e o período de volta às aulas no semestre da primavera.

Um terceiro exemplo são os supermercados, que têm muito menos sazonalidade mensal do que as outras duas séries temporais (embora provavelmente tenham sazonalidade no nível do dia da semana e da hora do dia). Isso não é surpreendente: as pessoas precisam comer o ano todo. As vendas dos supermercados aumentam um pouco em dezembro para os feriados e diminuem em fevereiro, já que esse mês simplesmente tem menos dias.



*Figura 3-16. Exemplos de padrões de sazonalidade nas vendas de livrarias, mercearias e joalherias*

A sazonalidade pode assumir muitas formas, embora existam algumas abordagens comuns para analisá-la independentemente. Uma maneira de lidar com a sazonalidade é suavizá-la, agregando os dados a um período de tempo menos granular ou usando janelas contínuas, como vimos anteriormente. Outra maneira de trabalhar com dados sazonais é comparar com períodos de tempo semelhantes e analisar a diferença. Mostrarei várias maneiras de fazer isso a seguir.

## Comparações de período a período: YoY e MoM

As comparações de período a período podem assumir várias formas. A primeira é comparar um período de tempo com o valor anterior da série, prática tão comum na análise que existem siglas para as comparações mais utilizadas. Dependendo do nível de agregação, a comparação pode ser ano a ano (YoY), mês a mês (MoM), dia a dia (DoD) e assim por diante.

Para esses cálculos usaremos a função `lag`, outra das funções da janela. A função `lag` retorna um valor anterior ou atrasado de uma série. A função `lag` tem a seguinte forma:

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

O `return_value` é qualquer campo do conjunto de dados e, portanto, pode ser qualquer tipo de dados. O opcional `OFFSET` indica quantas linhas de volta na partição para levar o `return_value`. O padrão é 1, mas qualquer valor inteiro pode ser usado. Você também pode opcionalmente especificar um valor `default` para usar se não houver nenhum registro atrasado para recuperar um valor. Como outras funções de janela, `lag` também é calculado sobre uma partição, com ordenação determinada pela cláusula `ORDER BY`. Se nenhuma cláusula `PARTITION BY` é especificado, `lag` analisa todo o conjunto de dados e, da mesma forma, se não houver cláusula `ORDER BY` for especificado, a ordem do banco de dados será usada. Geralmente é uma boa ideia incluir pelo menos uma cláusula `ORDER BY` em uma função do window `lag` para controlar a saída.



A função window `lead` funciona da mesma forma que a `lag`, exceto que ela retorna um valor subsequente conforme determinado pelo deslocamento. Alterar o `ORDER BY` de ascendente (`ASC`) para descendente (`DESC`) em uma série temporal tem o efeito de transformar uma instrução `lag` no equivalente a uma instrução `lead`. Alternativamente, um inteiro negativo pode ser usado como o `OFFSET` para retornar um valor de uma linha subsequente.

Vamos aplicar isso ao nosso conjunto de dados de vendas no varejo para calcular o crescimento MoM e YoY. Nesta seção, vamos nos concentrar nas vendas de livrarias, já que sou um verdadeiro nerd de livrarias. Primeiro, desenvolveremos nossa intuição sobre o que é retornado pela `lag` retornando os valores do mês de atraso e dos valores de vendas atrasados:

```
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` é retornado, assim como as `sales` desse mês, e podemos confirmar isso inspecionando as primeiras linhas do conjunto de resultados. A primeira linha tem null para `prev_month` e `prev_month_sales` uma vez que não há registro anterior neste conjunto de dados. Com a compreensão dos valores devolvidos pela função `lag`, podemos calcular a variação percentual do 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
...
...
```

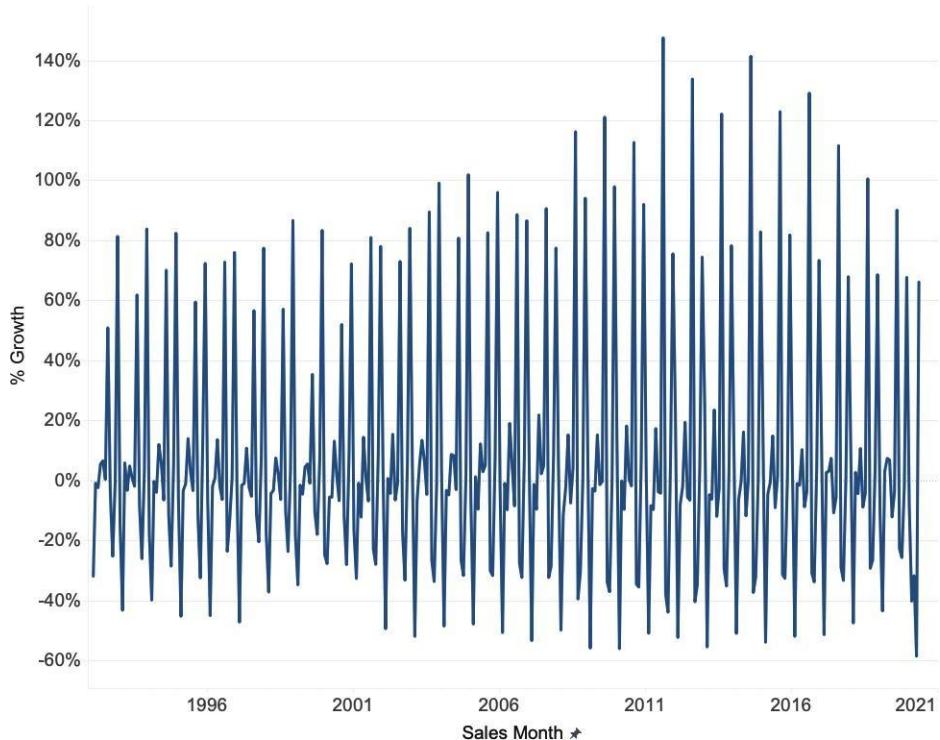
As vendas caíram 31,8% de janeiro a fevereiro, devido, pelo menos em parte, à queda sazonal após as férias e ao retorno às aulas para o semestre da primavera. As vendas caíram apenas 0,7% de fevereiro a março.

O cálculo para a comparação YoY é semelhante, mas primeiro precisamos agregar as vendas ao nível anual. Como estamos analisando apenas um `kind_of_business`, retirarei 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% desde 1992 a 1993, e quase 11% de 1993 a 1994. Esses cálculos período a período são úteis, mas não nos permitem analisar a sazonalidade no conjunto de dados. Por exemplo, na [Figura 3-17](#), os valores de crescimento MoM percentual são plotados e contêm tanta sazonalidade quanto a série temporal original.



*Figura 3-17. Crescimento percentual em relação ao mês anterior nas vendas de livrarias nos EUA*

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

## Comparações de período a período: mesmo mês versus ano passado

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

Para realizar essa comparação, podemos usar a função `lag` junto com um particionamento inteligente: a unidade de tempo com a qual queremos comparar o valor atual. Neste caso, vamos comparar mensalmente `sales` para o `sales` para o mesmo mês do ano anterior. Por exemplo, janeiro `sales` será comparado a janeiro do ano anterior `sales`, Fevereiro `sales` será comparado ao ano anterior fevereiro `sales`, e assim por diante.

Primeiro, lembre-se de que a função `date_part` retorna um valor numérico quando usada 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, incluímos o `date_part` na cláusula `PARTITION BY` para que a função de janela procure o valor do número do mês correspondente do ano anterior.

Este é um exemplo de como as cláusulas de função de janela podem incluir cálculos além dos campos do banco de dados, dando-lhes ainda mais versatilidade. Acho útil verificar resultados intermediários para construir a intuição sobre o que a consulta final retornará, então primeiro vamos confirmar que a função `lag` with `partition by date_part('month', sales_month)` retorna os valores pretendidos:

```
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'
 ;
```

<code>sales_month</code>	<code>sales</code>	<code>prev_year_month</code>	<code>prev_year_sales</code>
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 observando o valor `prev_year_month`. A fila para o 1993-01-01 `sales_month` retorna 01-01-1992 para o `prev_year_month` como pretendido, e o `prev_year_sales` de 790 correspondem ao `sales` podemos ver na linha 1992-01-01. Observe que o `prev_year_month` e `prev_year_sales` são nulos para 1992, pois não há registros anteriores no conjunto de dados.

Agora que estamos confiantes de que a função `lag` conforme escrito retorna os valores corretos, podemos calcular as métricas de comparação, como diferença absoluta e alteração percentual em relação ao anterior:

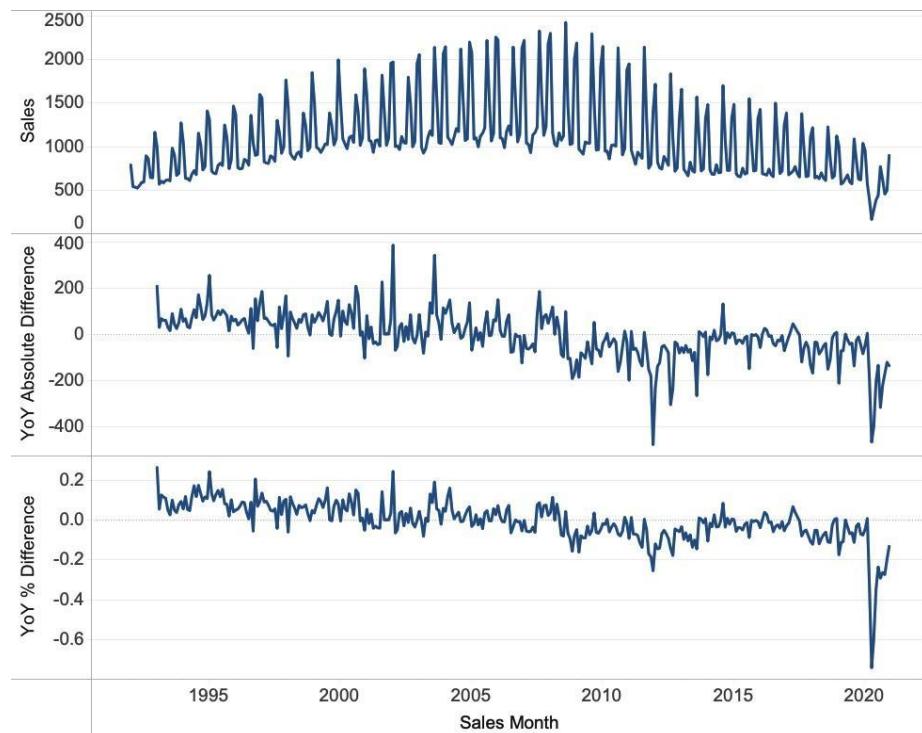
```

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 agora representar graficamente os resultados na [Figura 3-18](#) e ver mais facilmente os meses em que o crescimento foi excepcionalmente alta, como janeiro de 2002, ou excepcionalmente baixa, como dezembro de 2001.



*Figura 3-18. Vendas em livrarias, diferença absoluta nas vendas ano a ano e crescimento percentual ano a ano*

Outra ferramenta de análise útil é criar um gráfico que alinhe o mesmo período — neste caso, meses — com uma linha para cada série temporal — neste caso, anos . Para fazer isso, criaremos um conjunto de resultados que tenha uma linha para cada número ou nome do mês e uma coluna para cada um dos anos que desejamos considerar. Para obter o mês, podemos usar o `date_part` ou a função `to_char`, dependendo se queremos valores numéricos ou de texto para os meses. Em seguida, dinamizaremos os dados usando uma função de agregação.

Este exemplo usa o agregado `max`, mas dependendo da análise, um `sum`, `count`, ou outra agregação pode ser apropriada. Vamos ampliar de 1992 a 1994 para este exemplo:

```
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 desta forma, podemos ver algumas tendências imediatamente. As vendas de dezembro são as maiores vendas mensais do ano. As vendas em 1994 foram maiores a cada mês do que as vendas em 1992 e 1993. O aumento nas vendas de agosto a setembro é visível e particularmente fácil de detectar em 1994.

Com um gráfico dos dados, como na [Figura 3-19](#), as tendências são muito mais fácil de identificar. As vendas aumentaram ano a ano em todos os meses, embora os aumentos tenham sido maiores em alguns meses do que em outros. Com esses dados e gráficos em mãos, podemos começar a construir uma história sobre vendas de livrarias que pode ajudar no planejamento de estoque ou programação de promoções de marketing ou pode servir como evidência em uma história mais ampla sobre vendas no varejo nos EUA.

Com o SQL, existem várias técnicas para eliminar o ruído da sazonalidade para comparar dados em séries temporais. Nesta seção, vimos como comparar valores atuais com períodos comparáveis anteriores usando funções `lag` e como dinamizar os dados com `date_part`, `to_char`, e funções agregadas. A seguir, mostrarei algumas técnicas para comparar vários períodos anteriores para controlar ainda mais os dados de séries temporais com ruído.

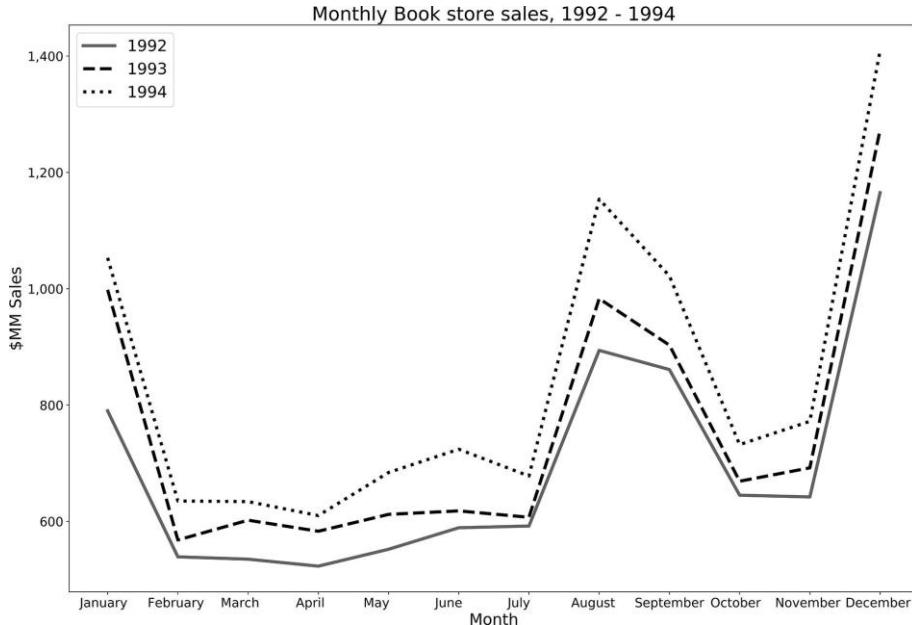


Figura 3-19. Vendas de livrarias para 1992–1994, alinhadas por mês

## Comparação com vários períodos anteriores

A comparação de dados com períodos anteriores comparáveis é uma maneira útil de reduzir o ruído que surge da sazonalidade. Às vezes, a comparação com um único período anterior é insuficiente, principalmente se esse período anterior foi impactado por eventos incomuns. Comparar uma segunda-feira com a segunda-feira anterior é difícil se uma delas for feriado. O mês do ano anterior pode ser incomum devido a eventos econômicos, condições climáticas severas ou uma interrupção do site que alterou o comportamento típico. Comparar os valores atuais com um agregado de vários períodos anteriores pode ajudar a suavizar essas flutuações. Essas técnicas também combinam o que aprendemos sobre o uso do SQL para calcular períodos de tempo contínuo e resultados comparáveis de períodos anteriores.

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

Para este exemplo, compararemos as vendas do mês atual com as vendas do mesmo mês em três anos anteriores. Como de costume, primeiro inspecionaremos os valores retornados para confirmar que o SQL está funcionando conforme o 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

```

Nulo é retornado quando não existe registro anterior, e podemos confirmar que o mesmo mês correto, ano anterior valor aparece. A partir daqui, podemos calcular qualquer métrica de comparação que a análise exigir - neste caso, a porcentagem 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 pelo resultado desse livro as vendas cresceram em relação à média móvel de três anos anterior em meados da década de 1990, mas o quadro era diferente no final da década de 2010, quando as vendas eram uma porcentagem cada vez menor dessa média móvel de três anos a cada ano.

Você deve ter notado que esse problema se assemelha a um que vimos anteriormente ao calcular as janelas de tempo de rolagem. Como alternativa ao último exemplo, podemos usar uma função de window `avg` com uma cláusula frame. Para realizar isso, o *PARTITION BY* usará a mesma função `date_part`, e o *ORDER BY* é o mesmo. Uma cláusula frame é adicionada para incluir `rows between 3 preceding and 1 preceding`. Isso inclui os valores nas linhas 1, 2 e 3 anteriores, mas exclui o valor na 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 correspondem aos do exemplo anterior, confirmando que o código alternativo é equivalente.



Se você olhar de perto, notará que os valores das casas decimais são ligeiramente diferentes no resultado usando as três `lag` e no resultado usando a função de window `avg`. Isso se deve ao modo como o banco de dados trata o arredondamento decimal em cálculos intermediários. Para muitas análises, a diferença não importa, mas preste muita atenção se estiver trabalhando com dados financeiros ou outros dados altamente regulamentados.

Analizar dados com sazonalidade geralmente envolve tentar reduzir o ruído para tirar conclusões claras sobre as tendências subjacentes nos dados. Comparar pontos de dados com vários períodos anteriores pode nos dar uma tendência ainda mais suave para comparar e determinar o que realmente está acontecendo no período atual. Isso requer que os dados incluam histórico suficiente para fazer essas comparações, mas quando temos uma série temporal suficientemente longa, pode ser perspicaz.

## Conclusão

A análise de séries temporais é uma maneira poderosa de analisar conjuntos de dados. Vimos como configurar nossos dados para análise com manipulações de data e hora. Conversamos sobre dimensões de data e vimos como aplicá-las ao cálculo de janelas de tempo contínuo. Analisamos os cálculos de período a período e como analisar dados com padrões de sazonalidade. No próximo capítulo, aprofundaremos um tópico relacionado que se estende à análise de séries temporais: análise de coorte.

## CAPÍTULO 4

# Análise de Coorte

No [Capítulo 3](#), abordamos a análise de séries temporais. Com essas técnicas em mãos, passaremos agora a um tipo de análise relacionado com muitos negócios e outras aplicações: análise de coorte.

Lembro-me da primeira vez que encontrei uma análise de coorte. Eu estava trabalhando no meu primeiro emprego de analista de dados, em uma pequena startup. Eu estava revisando uma análise de compra na qual trabalhei com o CEO, e ele sugeriu que eu dividisse a base de clientes em grupos para ver se o comportamento estava mudando ao longo do tempo. Presumi que fosse alguma coisa chique da escola de negócios e provavelmente inútil, mas ele era o CEO, então é claro que eu o agradava. Acontece que não foi apenas uma brincadeira. Dividir populações em coortes e segui-las ao longo do tempo é uma maneira poderosa de analisar seus dados e evitar vários vieses. As coortes podem fornecer pistas sobre como as subpopulações diferemumas das outras e como elas mudam ao longo do tempo.

Neste capítulo, veremos primeiro o que são coortes e os blocos de construção de certos tipos de análise de coortes. Após uma introdução ao conjunto de dados do legislador usado para os exemplos, aprenderemos como construir uma análise de retenção e lidar com vários desafios, como definir a coorte e lidar com dados esparsos. Em seguida, abordaremos os cálculos de sobrevivência, devolução e cumulativos, todos semelhantes à análise de retenção na forma como o código SQL é estruturado. Por fim, veremos como combinar a análise de coorte com a análise transversal para entender a composição das populações ao longo do tempo.

## Coortes: uma estrutura de análise útil

Antes de entrarmos no código, definirei o que são coortes, considerarei os tipos de perguntas que podemos responder com esse tipo de análise e descreverei os componentes de qualquer análise de coorte.

Uma *coorte* é um grupo de indivíduos que compartilham alguma característica de interesse, descrita a seguir, no momento em que começamos a observá-los. Os membros da coorte geralmente são pessoas, mas podem ser qualquer tipo de entidade que queremos estudar: empresas, produtos ou fenômenos do mundo físico. Indivíduos em uma coorte podem estar cientes de sua participação, assim como as crianças em uma classe de primeira série estão cientes de que fazem parte de um grupo de colegas de primeira série, ou os participantes de um teste de drogas estão cientes de que fazem parte de um grupo que recebe um tratamento. . Outras vezes, as entidades são agrupadas em coortes virtualmente, como quando uma empresa de software agrupa todos os clientes adquiridos em determinado ano para estudar por quanto tempo permanecem clientes. É sempre importante considerar as implicações éticas de coorte de entidades sem seu conhecimento, se algum tratamento diferente for aplicado a elas.

A *análise de coorte* é uma maneira útil de comparar grupos de entidades ao longo 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 mudanças. A análise de coorte fornece uma estrutura para detectar 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 adquiridos por meio de uma campanha de marketing podem ter padrões de compra de longo prazo diferentes daqueles 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 às coortes anteriores. Esse monitoramento pode fornecer um sinal de alerta antecipado de que algo deu errado (ou certo) para novos clientes. A análise de coorte também é usada para extrair dados históricos. Os testes A/B, discutidos no [Capítulo 7](#), são o padrão-ouro para determinar a causalidade, mas não podemos voltar no tempo e executar todos os testes para todas as perguntas sobre o passado que nos interessam. É claro que devemos ser cautelosos ao atribuir significado causal à análise de coorte e, em vez disso, usar a análise de coorte como uma forma de entender os clientes e gerar hipóteses que possam ser testadas rigorosamente no futuro.

As análises de coorte têm três componentes: o agrupamento de coorte, uma série temporal de dados sobre a qual a coorte é observada e uma métrica agregada que mede uma ação realizada pelos membros da coorte.

O *agrupamento de coortes* geralmente é baseado em uma data de início: a primeira compra do cliente ou a data da assinatura, a data em que um aluno iniciou a escola e assim por diante. No entanto, as coortes também podem ser formadas em torno de outras características que são inatas ou mudam ao longo do tempo. As qualidades inatas incluem o ano de nascimento e o país de origem, ou o ano em que a empresa foi fundada. As características que podem mudar ao longo do tempo incluem cidade de residência e estado civil. Quando estes são usados, precisamos ter o cuidado de coorte apenas no valor na data de início, ou então as entidades podem pular entre os grupos de coorte.

## Coorte ou Segmento?



Esses dois termos são frequentemente usados de maneira semelhante, ou até mesmo de forma intercambiável, mas vale a pena fazer uma distinção entre eles por uma questão de clareza. Uma *coorte* é um grupo de usuários (ou outras entidades) que têm uma data de início comum e são acompanhados ao longo do tempo. Um *segmento* é um agrupamento de usuários que compartilham uma característica comum ou um conjunto de características em um determinado momento, independentemente de sua data de início. Semelhante às coortes, os segmentos podem ser baseados em fatores inatos, como idade ou características comportamentais. Um segmento de usuários que se inscreve no mesmo mês pode ser colocado em uma coorte e acompanhado ao longo do tempo. Ou diferentes agrupamentos de usuários podem ser explorados com análise de coorte para que você possa ver quais têm as características mais valiosas. As análises que abordaremos neste capítulo, como retenção, podem ajudar a colocar dados concretos por trás dos segmentos de marketing.

O segundo componente de qualquer análise de coorte é a *série temporal*. Esta é uma série de compras, logins, interações ou outras ações que são realizadas pelos clientes ou entidades a serem coorte. É importante que a série temporal cubra toda a vida útil das entidades, ou haverá *viés de sobrevivência* nas primeiras coortes. O viés de sobrevivência ocorre quando apenas os clientes que permaneceram estão no conjunto de dados; os clientes churned são excluídos porque não estão mais por perto, então o resto dos clientes parece ser de maior qualidade ou se encaixam em comparação com coortes mais recentes (consulte “[Viés de Sobrevivência](#)” na [página 167](#)). Também é importante ter uma série temporal suficientemente longa para que as entidades concluam a ação de interesse. Por exemplo, se os clientes tendem a comprar uma vez por mês, é necessária uma série temporal de vários meses. Se, por outro lado, as compras ocorrerem apenas uma vez por ano, seria preferível uma série temporal de vários anos. Inevitavelmente, os clientes adquiridos mais recentemente não terão tanto tempo para concluir as ações quanto os clientes que foram adquiridos no passado. Para normalizar, a análise de coorte geralmente mede o número de períodos decorridos a partir de uma data de início, em vez de meses do calendário. Dessa forma, as coortes podem ser comparadas no período 1, período 2 e assim por diante para ver como elas evoluem ao longo do tempo, independentemente do mês em que a ação realmente ocorreu. Os intervalos podem ser dias, semanas, meses ou anos.

A *métrica agregada* deve estar relacionada às ações que são importantes para a saúde da organização, como clientes que continuam a usar ou comprar o produto. Os valores de métrica são agregados em toda a coorte, geralmente com `sum`, `count` ou `average`, embora qualquer agregação relevante funcione. O resultado é uma série temporal que pode ser usada para entender as mudanças no comportamento ao longo do tempo.

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

### *Retenção*

Retenção está relacionada com se o membro da coorte tem um registro na série temporal em uma determinada data, expressa como um número de períodos a partir da data de início. Isso é útil em qualquer tipo de organização em que são esperadas ações repetidas, desde jogar um jogo online até usar um produto ou renovar uma assinatura, e ajuda a responder perguntas sobre a aderência ou o envolvimento de um produto e quantas entidades podem ser esperadas para aparecer em datas futuras.

### *Sobrevida*

Sobrevida se preocupa com quantas entidades permaneceram no conjunto de dados por um determinado período de tempo ou mais, independentemente do número ou frequência de ações até aquele momento. A sobrevida é útil para responder a perguntas sobre a proporção da população que se espera que permaneça – seja em um sentido positivo, por não se mudar ou falecer, ou em um sentido negativo, por não se formar ou cumprir algum requisito.

### *Devolução*

Retorno ou comportamento de compra repetida está relacionado ao fato de uma ação ter acontecido mais do que um limite mínimo de vezes - geralmente simplesmente mais de uma vez - durante uma janela fixa de tempo. Esse tipo de análise é útil em situações em que o comportamento é intermitente e imprevisível, como no varejo, onde caracteriza a participação de compradores repetidos em cada coorte dentro de uma janela de tempo fixa.

### *Cumulativo*

Os cálculos cumulativos dizem respeito ao número total ou valores medidos em uma ou mais janelas de tempo fixas, independentemente de quando ocorreram durante essa janela. Os cálculos cumulativos são frequentemente usados em cálculos do valor da vida útil do cliente (LTV ou CLTV).

Os quatro tipos de análise de coorte nos permitem comparar subgrupos e entender como eles diferem ao longo do tempo para tomar melhores decisões sobre produtos, marketing e finanças. Os cálculos para os diferentes tipos são semelhantes, portanto, definiremos o estágio com retenção e, em seguida, mostrarei como modificar o código de retenção para calcular os outros tipos. Antes de mergulharmos na construção de nossa análise de coorte, vamos dar uma olhada no conjunto de dados que usaremos para os exemplos deste capítulo.

## O conjunto de dados dos legisladores

Os exemplos de SQL neste capítulo usarão um conjunto de dados de membros anteriores e atuais do Congresso dos Estados Unidos mantidos em um [repositório do GitHub](#). Nos EUA, o Congresso é responsável por redigir leis ou legislação, por isso seus membros também são conhecidos como legisladores. Como o conjunto de dados é um arquivo JSON, apliquei algumas transformações para produzir um modelo de dados mais adequado para análise e publiquei os dados em um formato adequado para acompanhar os exemplos na [pasta legisladores](#) ([https://github.com/cathytanimura/sql\\_book/tree/master/Chapter%204:%20Cohorts](https://github.com/cathytanimura/sql_book/tree/master/Chapter%204:%20Cohorts)).

O repositório de origem tem um excelente dicionário de dados, então não vou repetir todos os detalhes aqui. No entanto, fornecerei alguns detalhes que devem ajudar aqueles que não estão familiarizados com o governo dos EUA a acompanhar as análises deste capítulo.

O Congresso tem duas câmaras, o Senado (“sen” no conjunto de dados) e a Câmara dos Deputados (“rep”). Cada estado tem dois senadores, e eles são eleitos para mandatos de seis anos. Os representantes são alocados aos estados com base na população; cada representante tem um distrito que ele representa sozinho. Os representantes são eleitos para mandatos de dois anos. Os mandatos reais em qualquer uma das câmaras podem ser mais curtos caso o legislador morra ou seja eleito ou nomeado para um cargo superior. Os legisladores acumulam poder e influência por meio de cargos de liderança quanto mais tempo estão no cargo e, portanto, é comum concorrer à reeleição. Finalmente, um legislador pode pertencer a um partido político, ou pode ser um “independente”. Na era moderna, a grande maioria dos legisladores são democratas ou republicanos, e a rivalidade entre os dois partidos é bem conhecida. Os legisladores ocasionalmente mudam de partido enquanto estão no cargo.

Para as análises, utilizaremos duas tabelas: `legislators` e `legislators_terms`. A tabela `legislators` contém uma lista de todas as pessoas incluídas no conjunto de dados, com data de nascimento, sexo e um conjunto de campos de ID que podem ser usados para pesquisar a pessoa em outros conjuntos de dados. A tabela `legislators_terms` contém um registro para cada mandato de cada legislador, com data de início e término, e outros atributos como câmara e partido. O campo `id_bioguide` é usado como identificador único de um legislador e aparece em cada tabela. A Figura 4-1 mostra uma amostra dos dados `legislators`. [Figura 4-2](#) mostra uma amostra dos dados `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 Comyn	John	Comyn	1952-02-02	M	C001056	300027

Figura 4-1. Amostra da tabela legislators

*	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

Agora que entendemos o que é análise de coorte e do conjunto de dados que usaremos como exemplos, vamos ver como escrever SQL para análise de retenção. A pergunta-chave que o SQL nos ajudará a responder é: uma vez que os representantes assumem o cargo, por quanto tempo eles mantêm seus empregos?

## 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 serem mantidas. As empresas geralmente querem que seus clientes continuem comprando seus produtos ou usando seus serviços, já que reter clientes é mais lucrativo do que adquirir novos. Os empregadores querem reter seus funcionários, porque o recrutamento de substitutos é caro e demorado. Os funcionários eleitos buscam a reeleição para continuar trabalhando nas prioridades de seus eleitores.

A principal questão na análise de retenção é se o tamanho inicial da coorte – número de assinantes ou funcionários, valor gasto ou outra métrica importante – permanecerá constante, diminuirá ou aumentará ao longo do tempo. Quando há um aumento ou uma diminuição, a quantidade e a velocidade da mudança também são questões interessantes. Na maioria das análises de retenção, o tamanho inicial tende a diminuir com o tempo, uma vez que uma coorte pode perder, mas não pode ganhar novos membros depois de formada. A receita é uma exceção interessante, já que um grupo de clientes pode gastar mais nos meses subsequentes do que no primeiro mês coletivamente, mesmo que alguns deles se desfaçam.

A análise de retenção usa a `count` de entidades ou `sum` de dinheiro ou ações presentes no conjunto de dados para cada período a partir da data de início e normaliza dividindo esse número pela `count` ou `sum` de entidades, dinheiro ou ações no primeiro período. O resultado é expresso em porcentagem e a retenção no período inicial é sempre 100%. Com o tempo, a retenção baseada em contagens geralmente diminui e nunca pode exceder 100%, enquanto a retenção baseada em dinheiro ou ação, embora muitas vezes decrescente, pode aumentar e ser superior a 100% em um período de tempo. A saída da análise de retenção geralmente é exibida em forma de tabela ou gráfico, que é chamada de curva de retenção. Veremos vários exemplos de curvas de retenção mais adiante neste capítulo.

Gráficos de curvas de retenção podem ser usados para comparar coortes. A primeira característica a ser observada é a forma da curva nos poucos períodos iniciais, onde muitas vezes há uma queda inicial acentuada. Para muitos aplicativos de consumo, perder metade de uma coorte nos primeiros meses é comum. Uma coorte com uma curva mais ou menos inclinada que outras pode indicar mudanças no produto ou na fonte de aquisição de clientes que merecem uma investigação mais aprofundada. Uma segunda característica a ser observada é se a curva se achata após certo número de períodos ou continua declinando rapidamente até zero. Uma curva de achatamento indica que há um ponto no tempo a partir do qual a maior parte da coorte que permanece permanece indefinidamente. Uma curva de retenção que flexiona para cima, às vezes chamada de curva de sorriso, pode ocorrer se os membros da coorte retornarem ou forem reativados após ficarem fora do conjunto de dados por algum período. Por fim, as curvas de retenção que medem a receita de assinaturas são monitoradas quanto a sinais de aumento da receita por cliente ao longo do tempo, um sinal de um negócio de software SaaS saudável.

Esta seção mostrará como criar uma análise de retenção, adicionar agrupamentos de coorte da própria série temporal e de outras tabelas e lidar com dados ausentes e esparsos que podem ocorrer em dados de série temporal. Com essa estrutura em mãos, você aprenderá na seção seguinte como fazer modificações para criar os outros tipos relacionados de análise de coorte. Como resultado, esta seção sobre retenção será a mais longa do capítulo, à medida que você constrói o código e desenvolve sua intuição sobre os cálculos.

## SQL para uma curva de retenção básica

Para análise de retenção, como em outras análises de coorte, precisamos de três componentes: a definição de coorte, uma série temporal de ações e uma métrica agregada que mede algo relevante para a organização ou processo. No nosso caso, os membros da coorte serão os legisladores, a série temporal serão os mandatos de cada legislador e a métrica de interesse será a `count` dos que ainda estão no cargo a cada período a partir da data de início.

Começamos calculando a retenção básica, antes de passar para exemplos que incluem vários agrupamentos de coorte. O primeiro passo é encontrar a primeira data de posse de cada legislador (`first_term`). Usaremos essa data para calcular o número de períodos para cada data subsequente na série temporal. Para fazer isso, pegue o `min` do `term_start` e `GROUP BY` cada `id_bioguide`, o identificador exclusivo para 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
...          ...
```

O próximo passo é colocar este código em uma subconsulta e juntá-lo à série temporal. A função `age` é aplicada para calcular os intervalos entre cada `term_start` e o `first_term` para cada legislador. Aplicando as funções `date_part` ao resultado, com ano, transforma isso no número de períodos anuais. Como as eleições acontecem a cada dois ou seis anos, usaremos anos como intervalo de tempo para calcular o `periods`. Poderíamos usar um intervalo mais curto, mas neste conjunto de dados há pouca flutuação diária ou semanal. A `count` de legisladores com registros para esse período é 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 construção `date_part` e `age` podem ser substituído por esta função mais simples:

```
datediff('year',first_term,term_start)
```

Alguns bancos de dados, como o Oracle, colocam a `datediff` último:

```
datediff(first_term,term_start,'year')
```

Agora que temos os períodos e o número de legisladores retidos em cada, a etapa final é calcular o tamanho total do `cohort` e preenchê-lo em cada linha para que o `cohort_retained` possa ser dividido por ele. A função de window `first_value` retorna o primeiro registro na cláusula *PARTITION BY*, de acordo com a ordenação definida no *ORDER BY*, uma maneira conveniente de obter o tamanho da coorte em cada linha. Nesse caso, o `cohort_size` vem do primeiro registro em todo o conjunto de dados, portanto, *PARTITION BY* é omitido:

```
first_value(cohort_retained) over (order by period) as cohort_size
```

Para encontrar o percentual retido, divida o valor `cohort_retained` por este 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 legislators_terms
    GROUP BY 1
) a
JOIN legislators_terms b on a.id_bioguide = b.id_bioguide

```

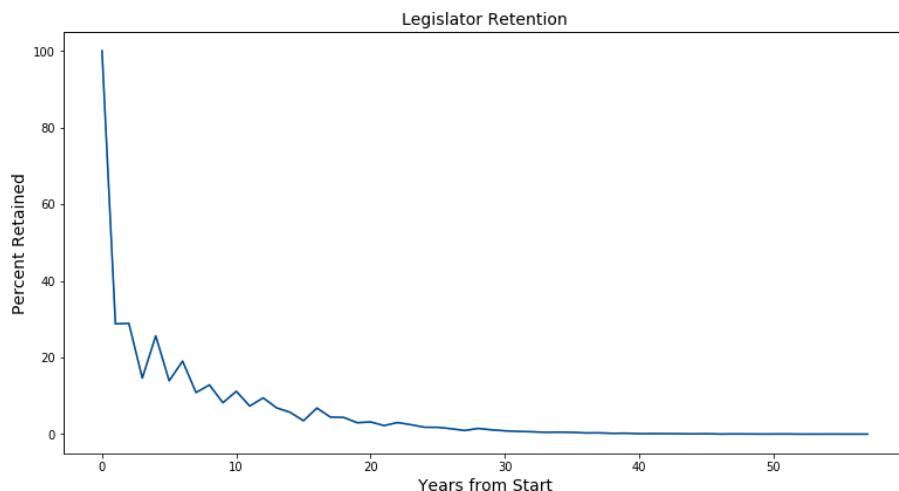
```

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
...     ...        ...        ...

```

Temos agora um cálculo de retenção, e podemos ver que há uma grande queda entre os 100% de legisladores retidos no período 0, ou em sua data de início, e o compartilhamento com outro registro de prazo que começa um ano depois. A representação gráfica dos resultados, como na [Figura 4-3](#), demonstra como a curva se achata e eventualmente vai para zero, já que mesmo os legisladores mais antigos acabam se aposentando ou morrem.



*Figura 4-3. Retenção desde o início do primeiro mandato para legisladores dos EUA*

Podemos pegar o resultado da retenção de coorte e remodelar os dados para mostrá-lo em formato de tabela. Dinamize e nivele os resultados usando uma função de agregação com uma instrução CASE; `max` é usado neste exemplo, mas outras agregações, como `min` ou `avg`, retornaria o mesmo resultado. A retenção é calculada para os anos de 0 a 4, mas anos adicionais podem ser adicionados seguindo 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 bastante baixa e, pelo gráfico, podemos ver que ela é irregular nos primeiros anos. Uma razão para isso é que o mandato dos deputados dura dois anos e os dos senadores seis anos, mas o conjunto de dados contém apenas registros para o início de novos mandatos; portanto, faltam dados de anos em que um legislador ainda estava no cargo, mas não iniciou um novo mandato. Medir a retenção a cada ano é enganoso neste caso. Uma opção é medir a retenção apenas em um ciclo de dois ou seis anos, mas também há outra estratégia que podemos empregar para preencher os dados “faltantes”. Abordarei isso a seguir antes de retornar ao tópico da formação de grupos de coorte.

## Ajustando Séries Temporais para Aumentar a Precisão de Retenção

Discutimos técnicas para limpar dados “ausentes” no [Capítulo 2](#), e vamos recorrer a essas técnicas nesta seção para chegar a uma curva de retenção mais suave e verdadeira para os legisladores. Ao trabalhar com dados de séries temporais, como na análise de coorte, é importante considerar não apenas os dados presentes, mas também se esses dados refletem com precisão a presença ou ausência de entidades em cada período de tempo. Isso é particularmente um problema em contextos em que um evento capturado nos dados leva a entidade a persistir por algum período de tempo que não é capturado nos dados.

Por exemplo, um cliente que compra uma assinatura de software é representado nos dados no momento da transação, mas esse cliente tem o direito de usar o software por meses ou anos e não é necessariamente representado nos dados durante esse período. Para corrigir isso, precisamos de uma maneira de derivar o período de tempo em que a entidade ainda está presente, seja com uma data de término explícita ou com conhecimento da duração da assinatura ou prazo. Então podemos dizer que a entidade estava presente em qualquer data entre essas datas de início e término.

No conjunto de dados dos legisladores, temos um registro para a data de início de um mandato, mas nos falta a noção de que isso “dá direito” a um legislador a servir por dois ou seis anos, dependendo da câmara. Para corrigir isso e suavizar a curva, precisamos preencher os valores “faltantes” para os anos em que os legisladores ainda estão no cargo entre novos mandatos. Como esse conjunto de dados inclui um valor `term_end` para cada termo, mostrarei como criar uma análise de retenção de coorte mais precisa preenchendo as datas entre os valores inicial e final. Em seguida, mostrarei como você pode imputar datas de término quando o conjunto de dados não inclui uma data de término.

Calcular a retenção usando uma data de início e término definida nos dados é a abordagem mais precisa. Para os exemplos a seguir, consideraremos os legisladores retidos em um determinado ano se ainda estivessem no cargo no último dia do ano, 31 de dezembro. Antes da Vigésima Emenda à Constituição dos EUA, os mandatos começavam em 4 de março, mas depois a data de início mudou para 3 de janeiro ou para um dia da semana subsequente se o terceiro cair em um fim de semana. Os legisladores podem prestar juramento em outros dias do ano devido a eleições especiais fora do ciclo ou nomeações para preencher assentos vagos. Como resultado, as datas `term_start` agrupam-se em janeiro, mas estão espalhadas ao longo do ano. Embora possamos escolher outro dia, 31 de dezembro é uma estratégia para normalizar essas datas de início variadas.

O primeiro passo é criar um conjunto de dados que contenha um registro para cada 31 de dezembro em que cada legislador esteve no cargo. Isso pode ser feito juntando-se a subconsulta que encontrou o `first_term` para a tabela `legislators_terms` para encontrar o `term_start` e `term_end` para cada termo. Um segundo JOIN para o `date_dim` recupera as datas que caem entre as datas de início e término, restringindo os valores retornados a `c.month_name = 'December' and c.day_of_month = 31`. O `period` é calculado como os anos entre o `date` de `date_dim` e o `first_term`. Observe que, embora tenham decorrido mais de 11 meses entre a tomada de posse em janeiro e 31 de dezembro, o primeiro ano ainda aparece 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 uma data dimensão não estiver disponível, você pode criar uma subconsulta com as datas necessárias de duas maneiras. Se seu banco de dados suportar o `generate_series`, você pode 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')

```

Você pode querer salvá-lo como uma tabela ou visualização para uso posterior. Como alternativa, você pode consultar o conjunto de dados ou qualquer outra tabela no banco de dados que tenha um conjunto completo de datas. Nesse caso, a tabela tem todos os anos necessários, mas faremos uma data de 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

```

Existem várias maneiras criativas de obter a série de datas necessárias. Use qualquer método disponível e mais simples em suas consultas.

Agora temos uma linha para cada `date` (final do ano) para a qual gostaríamos de calcular a retenção. O próximo passo é calcular o `cohort_retained` para cada período, o que é feito com uma `count` de `id_bioguide`. Uma função `coalesce` é usada no `period` para definir um valor padrão de 0 quando nulo. Isso lida com os casos em que o mandato de um legislador começa e termina no mesmo ano, dando crédito por servir naquele 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
...

```

O passo final é calcular o `cohort_size` e `pct_retained` como fizemos anteriormente usando funções da 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
...

```

Os resultados, representados graficamente na [Figura 4-4](#), agora são muito mais precisos. Quase todos os legisladores ainda estão no cargo no ano 1, e a primeira grande queda ocorre no ano 2, quando alguns deputados não serão reeleitos.

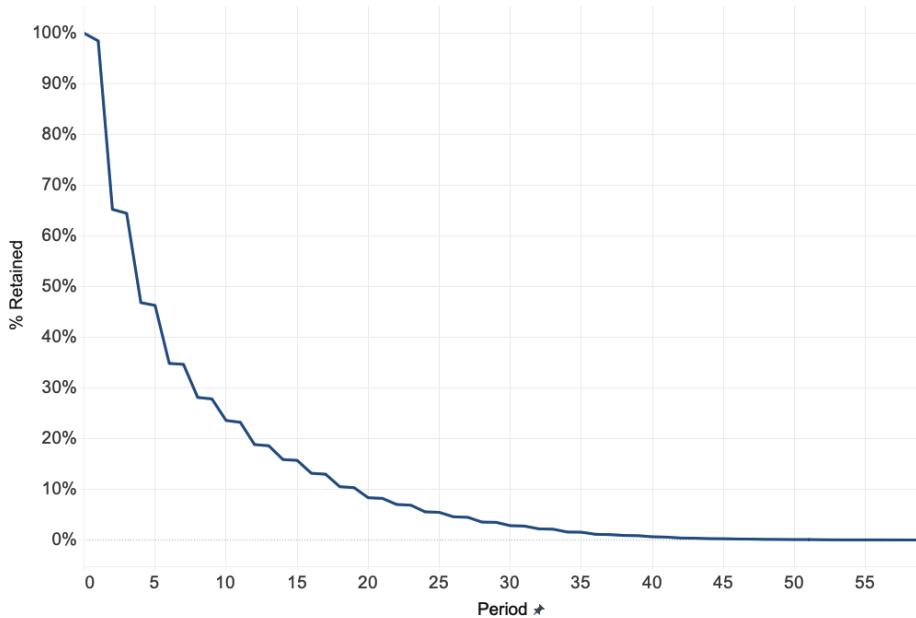


Figura 4-4. Retenção do legislador após ajuste para anos reais no cargo

Se o conjunto de dados não contiver uma data de término, há algumas opções para imputar uma. Uma opção é adicionar um intervalo fixo à data de início, quando a duração de uma assinatura ou prazo for conhecida. Isso pode ser feito com a matemática de datas adicionando um intervalo constante ao `term_start`. Aqui, uma instrução CASE trata da adição para os dois `term_types`:

```

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
...	...	...	...

Este bloco de código pode então ser conectado ao código de retenção para derivar o `período` e `pct_retained`. A desvantagem desse método é que ele não consegue capturar os casos em que um legislador não cumpriu um mandato completo, o que pode acontecer em caso de morte ou nomeação para um cargo superior.

Uma segunda opção é usar a data de início subsequente, menos um dia, como a data `term_end`. Isso pode ser calculado com a função de window `lead`. Esta função é semelhante à função `lag` usamos anteriormente, mas em vez de retornar um valor de uma linha anterior na partição, ele retorna um valor de uma linha posterior na partição, conforme determinado na cláusula *ORDER BY*. O padrão é uma linha, que usaremos aqui, mas a função possui um argumento opcional indicando um número diferente de linhas. Aqui encontramos a data `term_start` do termo subsequente usando `lead` e subtraia o intervalo '1 day' para derivar o `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
...          ...          ...          ...

```

Este bloco de código pode então ser conectado ao código de retenção. Este método tem algumas desvantagens. Primeiro, quando não há termo subsequente, a `lead` retorna null, deixando esse termo sem `term_end`. Um valor padrão, como um intervalo padrão mostrado no último exemplo, pode ser usado nesses casos. A segunda desvantagem é que esse método pressupõe que os mandatos sejam sempre consecutivos, sem tempo de afastamento. Embora a maioria dos legisladores tenda a servir continuamente até o fim de suas carreiras no Congresso, certamente há exemplos de lacunas entre mandatos que abrangem vários anos.

Sempre que fazemos ajustes para preencher dados ausentes, precisamos ter cuidado com as suposições que fazemos. Em contextos baseados em assinatura ou termo, as datas de início e término explícitas tendem a ser mais precisas.

Qualquer um dos outros dois métodos mostrados — adicionar um intervalo fixo ou definir a data de término em relação à próxima data de início — pode ser usado quando nenhuma data de término estiver presente e tivermos uma expectativa razoável de que a maioria dos clientes ou usuários permanecerá pelo período presumido .

Agora que vimos como calcular uma curva de retenção básica e corrigir datas ausentes, podemos começar a adicionar grupos de coorte. Comparar a retenção entre diferentes grupos é uma das principais razões para fazer uma análise de coorte. Em seguida, discutirei a formação de grupos da própria série temporal e, depois disso, discutirei a formação de grupos de coorte a partir de dados em outras tabelas.

## Coortes derivadas da própria série temporal

Agora que temos o código SQL para calcular a retenção, podemos começar a dividir as entidades em coortes. Nesta seção, mostrarei como derivar agrupamentos de coortes da própria série temporal. Primeiro, discutirei coortes com base no tempo com base na primeira data e explicarei como fazer coortes com base em outros atributos da série temporal.

A maneira mais comum de criar as coortes é baseada na primeira ou mínima data ou hora em que a entidade aparece na série temporal. Isso significa que apenas uma tabela é necessária para a análise de retenção de coorte: a própria série temporal. Coorte pela primeira aparição ou ação é interessante porque muitas vezes os grupos que começam em momentos diferentes se comportam de maneira diferente. Para serviços ao consumidor, os adotantes iniciais geralmente são mais entusiasmados e retêm de forma diferente dos adotantes posteriores, enquanto no software SaaS, os adotantes posteriores podem reter melhor porque o produto é mais maduro. As coortes baseadas em tempo podem ser agrupadas por qualquer granularidade de tempo que seja significativa para a organização, embora as coortes semanais, mensais ou anuais sejam comuns. Se você não tiver certeza de qual agrupamento usar, tente executar a análise de coorte com diferentes agrupamentos, sem diminuir o tamanho dos coortes, para ver onde surgem padrões significativos. Felizmente, uma vez que você saiba como construir as coortes e a análise de retenção, a substituição de diferentes granularidades de tempo é simples.

O primeiro exemplo usará coortes anuais e, em seguida, demonstrarei a troca em séculos. A questão-chave que consideraremos é se a época em que um legislador assumiu o cargo tem alguma correlação com sua retenção. As tendências políticas e o humor do público mudam com o tempo, mas em quanto?

Para calcular coortes anuais, primeiro adicionamos o ano do `first_term` anteriormente à consulta que encontra o `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
...

```

Esta consulta é então usada como subconsulta, e `cohort_size` e `pct_retained` são calculados na consulta externa como anteriormente. Nesse caso, no entanto, precisamos de uma cláusula `PARTITION BY` que inclua `first_year` para que o `first_value` seja calculado apenas dentro do conjunto de linhas para esse `first_year`, em vez de em 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, muitos para representar graficamente ou examinar em uma tabela. Em seguida, vamos olhar para um intervalo menos granular e coorte os legisladores por século do `first_term`. Essa mudança é feita facilmente substituindo `century` para `year` na função `date_part` na subconsulta `aa`. Lembre-se que os nomes dos séculos estão deslocados dos anos que representam, de modo que o século XVIII durou de 1700 a 1799, o século XIX durou de 1800 a 1899, e assim por diante. O particionamento na função `first_value` mudanças no 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 graficamente na [Figura 4- 5](#). A retenção nos primeiros anos foi maior para os primeiros eleitos no século 20 ou 21. O século 21 ainda está em andamento e, portanto, muitos desses legisladores não tiveram a oportunidade de permanecer no cargo por cinco ou mais anos, embora ainda estejam incluídos no denominador. Podemos querer considerar a remoção do século 21 da análise, mas deixei aqui para demonstrar como a curva de retenção cai artificialmente devido a essa circunstância.

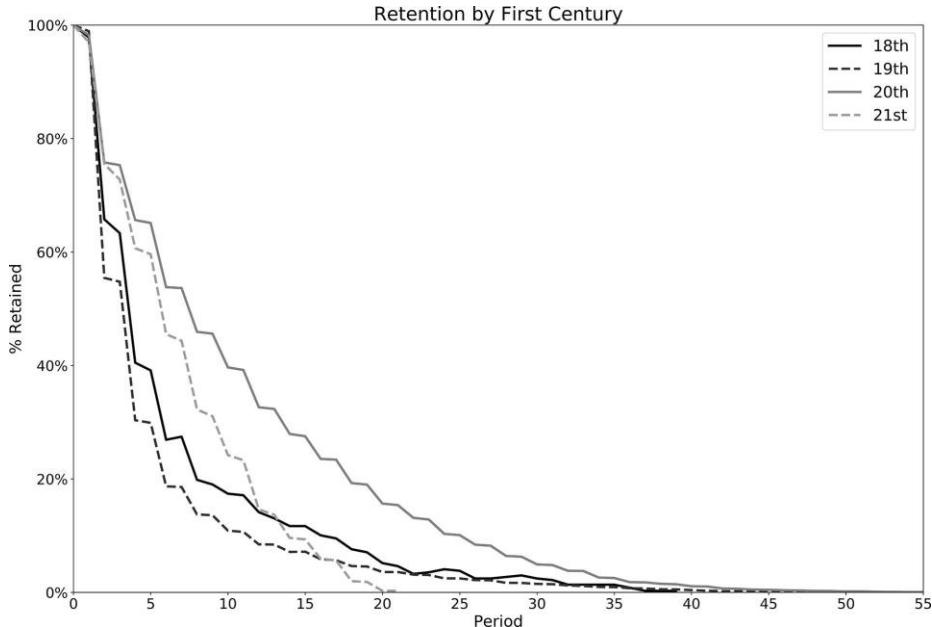


Figura 4-5. Retenção do legislador por século de início do primeiro mandato

As coortes podem ser definidas a partir de outros atributos em uma série temporal além da primeira data, com opções dependendo dos valores da tabela. A tabela `legislators_terms` possui um campo `state`, indicando qual estado a pessoa está representando para aquele termo. Podemos usar isso para criar coortes e as basearemos no primeiro estado para garantir que qualquer pessoa que tenha representado vários estados apareça nos dados apenas uma vez.



Ao coorte em um atributo que pode mudar ao longo do tempo, é importante garantir que cada entidade receba apenas um valor. Caso contrário, a entidade pode ser representada em várias coortes, introduzindo viés na análise. Normalmente, é usado o valor do registro mais antigo no conjunto de dados.

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 longa *GROUP BY*:

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

Podemos então inserir este código em nosso 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 para os cinco estados com maior o número total de legisladores está representado graficamente na [Figura 4-6](#). Os eleitos em Illinois e Massachusetts têm a maior retenção, enquanto os nova-iorquinos têm a menor retenção. Determinar as razões pelas quais seria um desdobramento interessante desta análise.

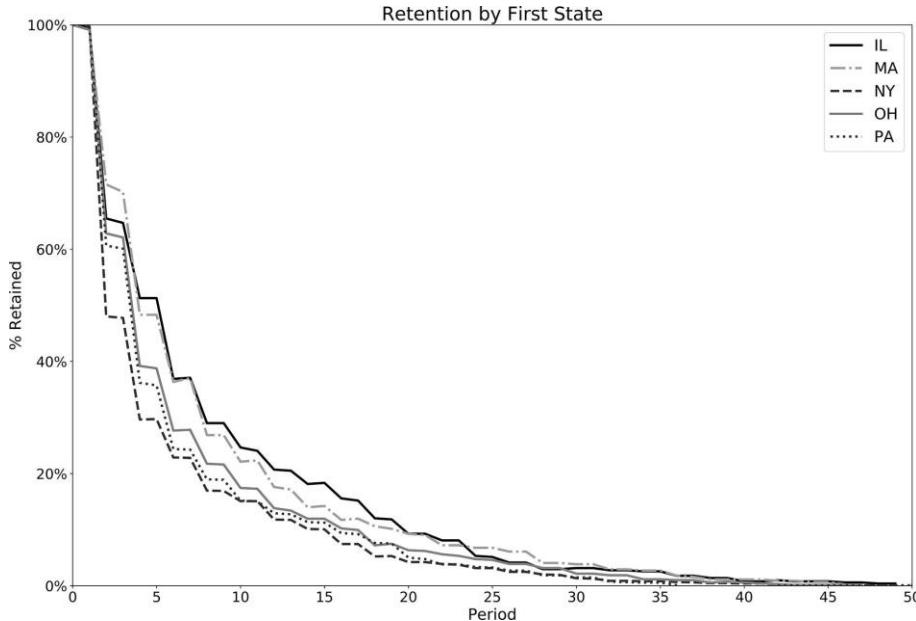


Figura 4-6. Retenção de legisladores por primeiro estado: os cinco principais estados pelo total de legisladores

Definir coortes da série temporal é relativamente simples usando uma `min` para cada entidade e, em seguida, convertendo essa data em um mês, ano ou século, conforme apropriado para a análise. Alternar entre mês e ano ou outros níveis de granularidade também é simples, permitindo que várias opções sejam testadas para encontrar um agrupamento significativo para a organização. Outros atributos podem ser usados para coorte com a função de janela `first_value`. A seguir, vamos nos voltar para os casos em que o atributo de coorte vem de uma tabela diferente daquela da série temporal.

## Definindo a Coorte a partir de uma Tabela Separada

Muitas vezes, as características que definem uma coorte existem em uma tabela separada daquela que contém a série temporal. Por exemplo, um banco de dados pode ter uma tabela de clientes com informações como fonte de aquisição ou data de registro pela qual os clientes podem ser agrupados. Adicionar atributos de outras tabelas, ou mesmo subconsultas, é relativamente simples e pode ser feito na análise de retenção e análises relacionadas discutidas posteriormente neste capítulo.

Para este exemplo, consideraremos se o gênero do legislador tem algum impacto em sua retenção. A tabela `legislators` tem um campo `gender`, onde F significa feminino e M significa masculino, que podemos usar para coorte dos legisladores. Para fazer isso, vamos `JOIN` as

tabela `legisladores` como alias `d` para adicionar `gender` ao cálculo de `cohort_retained`, no lugar de ano ou século:

```

SELECT d.gender
,coalesce(date_part('year',age(c.date,a.first_term)),0) as period
,coalesce(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 muito mais homens do que mulheres cumpriram mandatos legislativos. Podemos agora calcular a `percent_retained` para podermos comparar a retenção para estes 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
    ,coalesce(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
)

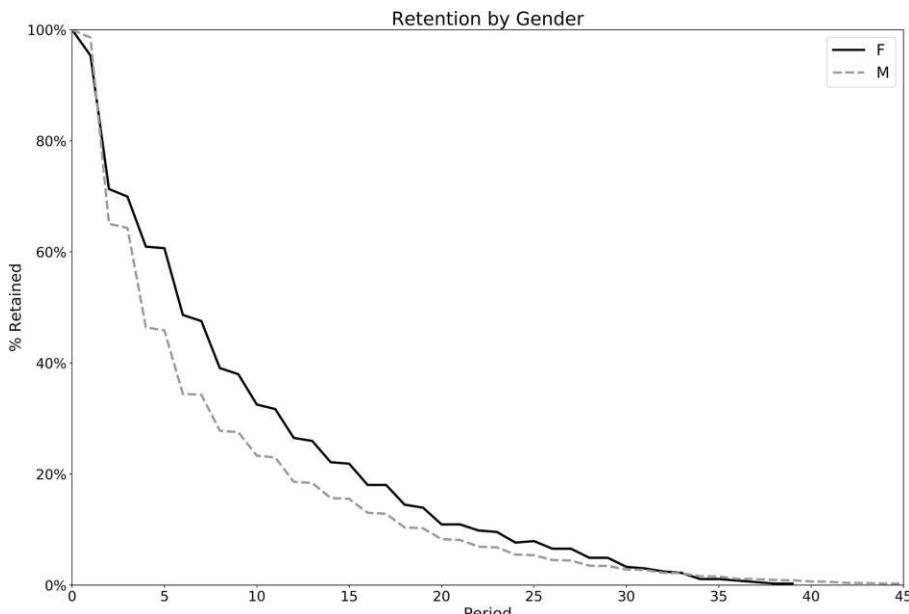
```

```

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
All	All	All	All	All

Podemos ver pelos resultados gráficos na Figura 4-7 que a retenção é maior para legisladoras mulheres do que para seus colegas homens nos períodos de 2 a 29. A primeira legisladora só assumiu o cargo em 1917, quando Jeannette Rankin ingressou na Câmara como republicana, representante de Montana. Como vimos anteriormente, a retenção aumentou nos séculos mais recentes.



*Figura 4-7. Retenção de legisladores por gênero*

Para fazer uma comparação mais justa, podemos restringir os legisladores incluídos na análise apenas àqueles cujos `first_term` começou desde que há mulheres no Congresso. Podemos fazer isso adicionando um filtro `WHERE` à subconsulta `aa`. Aqui os resultados também são restritos àqueles que começaram antes de 2000, para garantir que as coortes tenham tido pelo menos 20 anos possíveis para permanecer 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 homens ainda superam os legisladores femininos, mas por um margem menor. A retenção para as coortes está representada graficamente na [Figura 4-8](#). Com as coortes revisadas, os legisladores homens têm maior retenção até o 7º ano, mas a partir do 12º ano, as legisladoras têm maior retenção. A diferença entre as duas análises de coorte baseadas em gênero ressalta a importância de estabelecer coortes apropriadas e garantir que elas tenham um tempo comparável para estar presente ou concluir outras ações de interesse. Para melhorar ainda mais essa análise, poderíamos coorte por ano ou década de início e gênero, a fim de controlar mudanças adicionais na retenção ao longo do século 20 e no século 21.

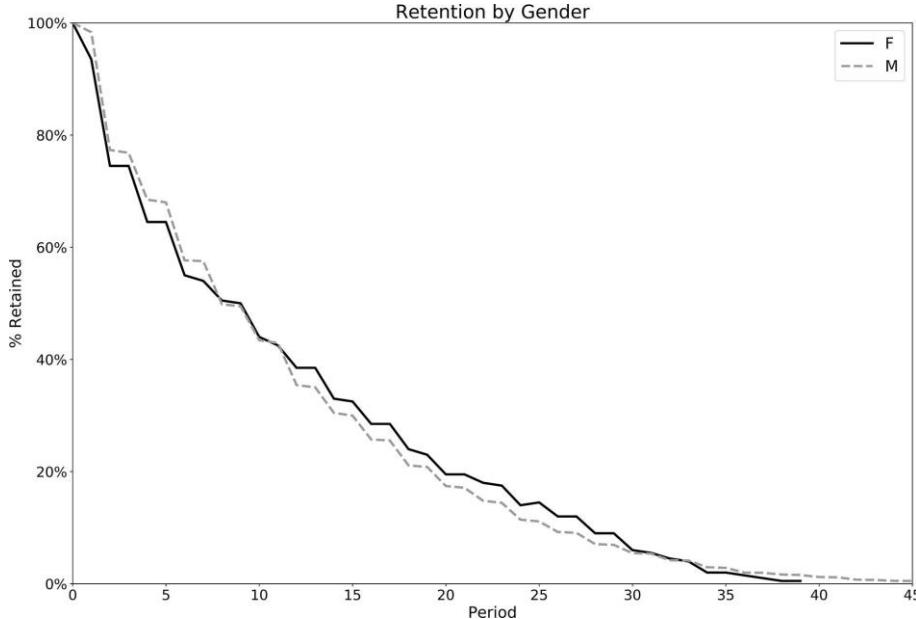


Figura 4-8. Retenção do legislador por gênero: coortes de 1917 a 1999

As coortes podem ser definidas de várias maneiras, a partir da série temporal e de outras tabelas. Com a estrutura que desenvolvemos, subconsultas, visualizações ou outras tabelas derivadas podem ser trocadas, abrindo toda uma gama de cálculos para serem a base de uma coorte. Vários critérios, como ano de início e sexo, podem ser usados. Um cuidado ao dividir populações em coortes com base em vários critérios é que isso pode levar a coortes esparsas, em que alguns dos grupos definidos são muito pequenos e não são representados no conjunto de dados para todos os períodos de tempo. A próxima seção discutirá métodos para superar esse desafio.

## Lidando com coortes esparsas

No conjunto de dados ideal, cada coorte tem alguma ação ou registro na série temporal para cada período de interesse. Já vimos como datas “ausentes” podem ocorrer devido a assinaturas ou termos que duram vários períodos e analisamos como corrigi-las usando uma dimensão de data para inferir datas intermediárias. Outro problema pode surgir quando, devido a critérios de agrupamento, a coorte se torna muito pequena e, como resultado, é representada apenas esporadicamente nos dados. Uma coorte pode desaparecer do conjunto de resultados, quando preferirmos que apareça com um valor de retenção zero. Esse problema é chamado *coortes esparsas* e pode ser contornado com o uso cuidadoso de *LEFT JOINs*.

Para demonstrar isso, vamos tentar coortar as legisladoras pelo primeiro estado que elas representaram para ver se há alguma diferença na retenção.

Já vimos que houve relativamente poucas legisladoras. Coorte-os ainda mais por estado é altamente provável que crie algumas coortes esparsas nas quais há muito poucos membros. Antes de fazer ajustes no código, vamos adicionar `first_state` (calculado na seção sobre derivação de coortes da série temporal) em nosso exemplo de gênero anterior e observar os resultados:

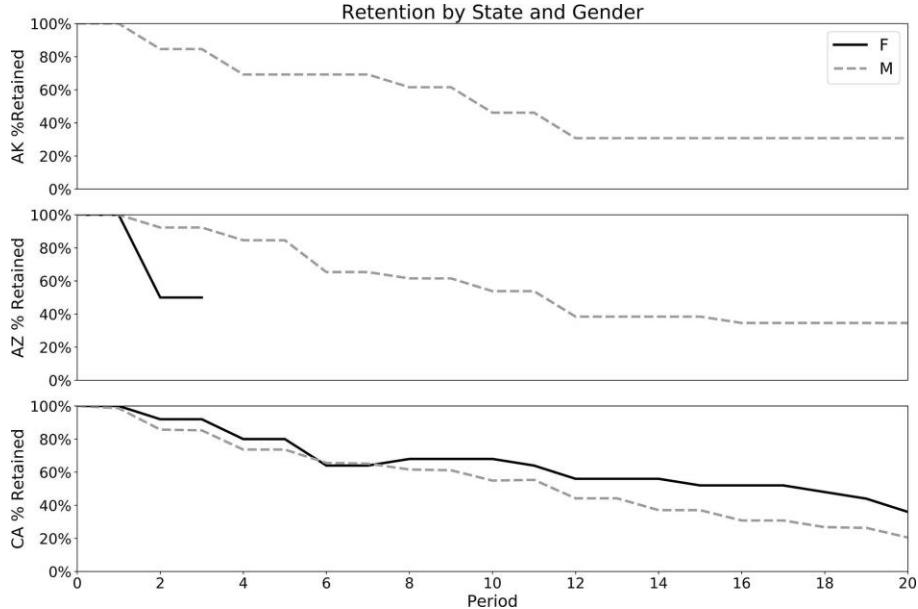
```

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
...	...	...	...	...	...

A representação gráfica dos resultados para os primeiros 20 períodos, como na [Figura 4-9](#), revela as coortes esparsas. O Alasca não teve nenhuma legisladora do sexo feminino, enquanto a curva de retenção feminina do Arizona desaparece após o ano 3. Apenas a Califórnia, um grande estado com muitos legisladores, possui curvas de retenção completas para ambos os sexos. Esse padrão se repete para outros estados pequenos e grandes.



*Figura 4-9. Retenção do legislador por sexo e primeiro estado*

Agora vamos ver como garantir um registro para cada período para que a consulta retorne valores zero para retenção em vez de nulos. O primeiro passo é consultar todas as combinações de `periods` e atributos de coorte, neste caso `first_state` e `gender`, com o início `cohort_size` para cada combinação. Isso pode ser feito pela subconsulta JOINing `aa`, que calcula a coorte, com uma subconsulta `generate_series` que retorna todos os inteiros de 0 a 20, com os critérios `on 1 = 1`. Esta é uma maneira prática de forçar um JOIN cartesiano quando as duas subconsultas não têm 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 *JOIN* de volta aos períodos reais no escritório, com um *LEFT JOIN* para garantir 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

... ... ... ... ... Podemos então dinamizar os resultados e confirmar que existe um valor para cada coorte para 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, neste ponto, o código SQL ficou bastante longo. Uma das partes mais difíceis de escrever SQL para análise de retenção de coorte é manter toda a lógica correta e o código organizado, um tópico que discutirei mais no [Capítulo 8](#). Ao criar o código de retenção, acho útil seguir passo a passo, verificando os resultados ao longo do caminho. Eu também verifico coortes individuais para validar se o resultado final é preciso.

As coortes podem ser definidas de várias maneiras. Até agora, normalizamos todas as nossas coortes para a primeira data em que aparecem nos dados da série temporal. Esta não é a única opção, no entanto, e uma análise interessante pode ser feita começando no meio da vida útil de uma entidade. Antes de concluir nosso trabalho sobre análise de retenção, vamos dar uma olhada nesta forma adicional de definir coortes.

## Definindo Coortes de Datas Diferentes da Primeira Data

Geralmente, as coortes baseadas em tempo são definidas a partir da primeira aparição da entidade na série temporal ou de alguma outra data anterior, como uma data de registro. No entanto, coorte em uma data diferente pode ser útil e perspicaz. Por exemplo, podemos querer analisar a retenção de todos os clientes que usam um serviço a partir de uma determinada data. Esse tipo de análise pode ser usado para entender se as mudanças no produto ou no marketing tiveram um impacto de longo prazo nos clientes existentes.

Ao usar uma data diferente da primeira, precisamos ter o cuidado de definir com precisão os critérios de inclusão em cada coorte. Uma opção é escolher entidades presentes em uma determinada data do calendário. Isso é relativamente simples de colocar no código SQL, mas pode ser problemático se uma grande parte da população de usuários regulares não aparecer todos os dias, fazendo com que a retenção varie dependendo do dia exato escolhido. Uma opção para corrigir isso é calcular a retenção para várias datas de início e, em seguida, calcular a média dos resultados.

Outra opção é usar uma janela de tempo, como uma semana ou um mês. Qualquer entidade que apareça no conjunto de dados durante essa janela é incluída na coorte. Embora essa abordagem geralmente seja mais representativa do negócio ou processo, a desvantagem é que o código SQL se tornará mais complexo e o tempo de consulta poderá ser mais lento devido a cálculos mais intensos do banco de dados. Encontrar o equilíbrio certo entre o desempenho da consulta e a precisão dos resultados é uma arte.

Vamos dar uma olhada em como calcular essa análise midstream com o conjunto de dados dos legisladores considerando a retenção de legisladores que estavam no cargo no ano 2000. Faremos uma coorte pelo `term_type`, que tem valores de “sen” para senadores e “rep” para os representantes. A definição incluirá qualquer legislador em exercício em qualquer momento durante o ano 2000: aqueles que começaram antes de 2000 e cujos mandatos terminaram durante ou após 2000 se qualificam, assim como aqueles que iniciaram um mandato em 2000. Podemos codificar qualquer data em 2000 como o `first_term`, visto que mais tarde verificaremos se eles estiveram no cargo em algum momento de 2000. O `min_start` dos termos que caem nesta janela também é calculado 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 conectar isso ao nosso código de retenção, com dois ajustes. Primeiro, um critério JOIN adicional entre a subconsulta `a` e a tabela `legislators_terms` é adicionado para retornar apenas os termos que começaram na data ou após a data `min_start`. Em segundo lugar, um filtro adicional é adicionado ao `date_dim` para que só retorne `dates` em 2000 ou mais tarde:

```

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
...	...	...	...	...

**Figura 4-10** mostra que apesar mandatos mais longos para os senadores, a retenção entre as duas coortes foi semelhante e, na verdade, pior para os senadores após 10 anos. Uma análise mais aprofundada comparando os diferentes anos em que foram eleitos pela primeira vez, ou outros atributos da coorte, pode produzir alguns insights interessantes.

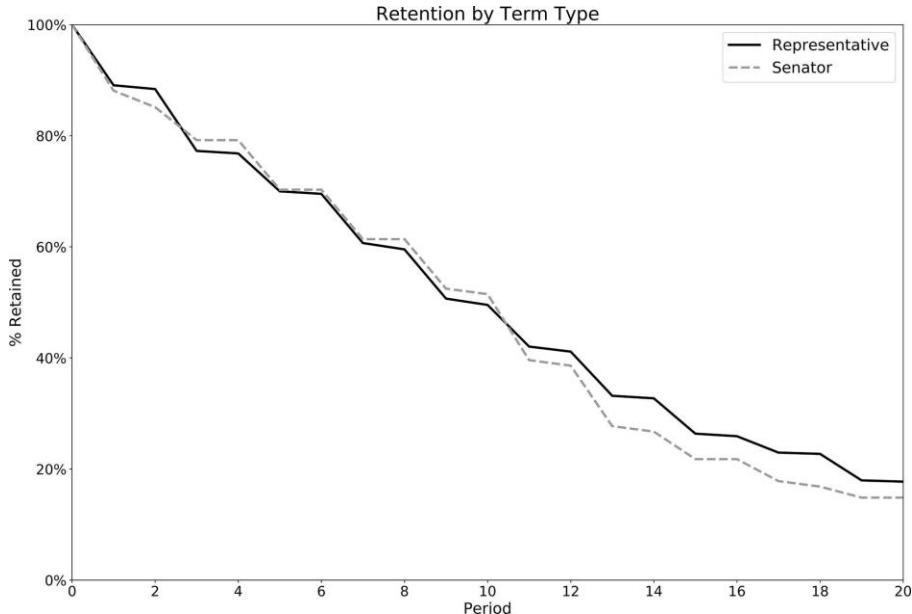


Figura 4-10. Retenção por tipo de mandato para legisladores em exercício durante o ano 2000

Um caso de uso comum para coorte em um valor diferente de um valor inicial é quando se tenta analisar a retenção depois que uma entidade atingiu um limite, como um certo número de compras ou um determinado valor gasto. Como em qualquer coorte, é importante ter cuidado ao definir o que qualifica uma entidade para fazer parte de uma coorte e qual data será usada como data de início.

A retenção de coorte é uma maneira poderosa de entender o comportamento de entidades em um conjunto de dados de série temporal. Vimos como calcular a retenção com SQL e como coorte com base na própria série temporal ou em outras tabelas e a partir de pontos no meio da vida útil da entidade. Também analisamos como usar funções e *JOINS* para ajustar datas em séries temporais e compensar coortes esparsas. Existem vários tipos de análises relacionadas à retenção de coorte: análise, sobrevivência, retorno e cálculos cumulativos, todos baseados no código SQL que desenvolvemos para retenção. Vamos nos voltar para eles a seguir.

## Análises de coorte relacionadas

Na última seção, aprendemos como escrever SQL para análise de retenção de coorte. A retenção captura se uma entidade estava em um conjunto de dados de série temporal em uma data ou janela de tempo específica. Além da presença em uma data específica, a análise geralmente está interessada em questões de quanto tempo uma entidade durou, se uma entidade realizou várias ações e quantas dessas ações ocorreram.

Tudo isso pode ser respondido com um código semelhante à retenção e adequado para praticamente qualquer critério de coorte que você desejar. Vamos dar uma olhada no primeiro deles, a sobrevivência.

## Survivorship

*Survivorship*, também chamado *análise de sobrevivência*, está preocupado com questões sobre quanto tempo algo dura, ou a duração de tempo até um determinado evento, como churn ou morte. A análise de sobrevivência pode responder a perguntas sobre a parcela da população que provavelmente permanecerá após um determinado período de tempo. As coortes podem ajudar a identificar ou pelo menos fornecer hipóteses sobre quais características ou circunstâncias 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 determinado período, calculamos se a entidade está presente nesse período ou posteriormente na série temporal. Em seguida, a parcela da coorte total é calculada. Normalmente, um ou mais períodos são escolhidos dependendo da natureza do conjunto de dados analisado. Por exemplo, se quisermos saber a proporção de jogadores que sobrevivem por uma semana ou mais, podemos verificar as ações que ocorrem após uma semana do início e considerar esses jogadores ainda sobreviventes. Por outro lado, se estivermos preocupados com o número de alunos que ainda estão na escola após um certo número de anos, poderíamos procurar a ausência de um evento de graduação em um conjunto de dados. O número de períodos pode ser *SELECTed* calculando uma duração média ou típica ou escolhendo períodos de tempo que sejam significativos para a organização ou processo analisado, como um mês, ano ou período de tempo mais longo.

Neste exemplo, veremos a parcela de legisladores que sobreviveram no cargo por uma década ou mais após o início de seu primeiro mandato. Como não precisamos saber as datas específicas de cada termo, podemos começar calculando a primeira e a última data `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, adicionamos à consulta uma função `date_part` para encontrar o século do `min(term_start)`, e calculamos a `tenure` como o número de anos entre o `min` e `max(term_start)` encontrados 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
...          ...        ...

```

Finalmente, calculamos o `cohort_size` com um `count` de todos os legisladores, bem como calculamos o número que sobreviveu por pelo menos 10 anos usando uma instrução `CASE` e agregação `count`. A porcentagem que sobreviveu é encontrada dividindo estes 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

```

Como os mandatos podem ou não ser consecutivos, também podemos calcular a proporção de legisladores em cada século que sobreviveram por cinco ou mais mandatos totais. Na subconsulta, adicione um `count` to find the total number of terms per legislator. Then in the outer query, divide the number of legislators with five or more terms by the total cohort size:

```

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 algo arbitrário. Também podemos calcular a sobrevivência para cada número de anos ou períodos e exibir os resultados em forma de gráfico ou tabela. Aqui, calculamos a sobrevivência para cada número de termos de 1 a 20. Isso é feito através de um `JOIN` 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
LATERAL generate_series(1, 20) b;
```

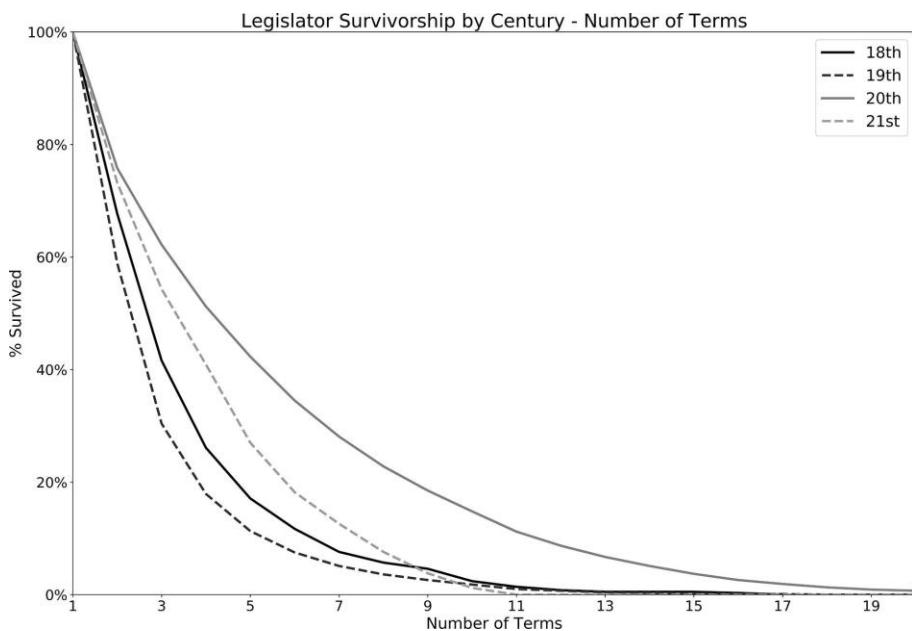
```

(
    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 graficamente na [Figura 4-11](#). A sobrevivência foi maior no século 20, um resultado que concorda com os resultados que vimos anteriormente em que a retenção também foi maior no século 20.



*Figura 4-11. Sobrevivência para legisladores: parcela da coorte que permaneceu no cargo por tantos mandatos ou mais*

A sobrevivência está intimamente relacionada à retenção. Enquanto a retenção conta as entidades presentes em um número específico de períodos desde o início, a sobrevivência considera apenas se uma entidade estava presente em um período específico ou posterior. Como resultado, o código é mais simples, pois precisa apenas da primeira e da última data na série temporal ou de uma contagem de datas.

A coorte é feita de forma semelhante à coorte para retenção, e as definições de coorte podem vir de dentro da série temporal ou ser derivadas de outra tabela ou subconsulta.

A seguir, consideraremos outro tipo de análise que é, de certa forma, o inverso da sobrevivência. Em vez de calcular se uma entidade está presente no conjunto de dados em um determinado momento ou mais tarde, calcularemos se uma entidade retorna ou repete uma ação em um determinado período ou antes. Isso é chamado de retorno ou comportamento de compra repetida.

## Devolução ou Comportamento de Compra Repetida

A sobrevivência é útil para entender por quanto tempo uma coorte provavelmente permanecerá. Outro tipo útil de análise de coorte procura entender se um membro da coorte pode retornar dentro de uma determinada janela de tempo e a intensidade da atividade durante essa janela. Isso é chamado *de retorno ou comportamento de compra repetida*.

Por exemplo, um site de comércio eletrônico pode querer saber não apenas quantos novos compradores foram adquiridos por meio de uma campanha de marketing, mas também se esses compradores se tornaram compradores recorrentes. Uma maneira de descobrir isso é simplesmente calcular o total de compras por cliente. No entanto, comparar os clientes adquiridos há dois anos com os adquiridos há um mês não é justo, pois os primeiros têm muito mais tempo para retornar. A coorte mais antiga quase certamente pareceria mais valiosa do que a mais nova. Embora isso seja verdade em certo sentido, fornece uma imagem incompleta de como as coortes provavelmente se comportarão ao longo de toda a vida.

Para fazer comparações justas entre coortes com datas de início diferentes, precisamos criar uma análise com base em uma *caixa de tempo* ou em uma janela de tempo fixa a partir da primeira data e considerar se os membros da coorte retornaram dentro dessa janela. Dessa forma, cada coorte tem uma quantidade igual de tempo em consideração, desde que incluimos apenas aquelas coortes para as quais a janela completa tenha decorrido. A análise de retorno é comum para organizações de varejo, mas também pode ser aplicada em outros domínios. Por exemplo, uma universidade pode querer ver quantos alunos matriculados em um segundo curso, ou um hospital pode estar interessado em quantos pacientes precisam de tratamentos médicos de acompanhamento após um incidente inicial.

Para demonstrar a análise de retorno, podemos fazer uma nova pergunta ao conjunto de dados dos legisladores: quantos legisladores têm mais de um tipo de mandato e, especificamente, qual parte deles começa como deputado e se torna senador (alguns senadores depois se tornam deputados, mas isso é muito menos comum). Como relativamente poucos fazem essa transição, agruparemos os legisladores pelo século em que se tornaram representantes pela primeira vez.

O primeiro passo é encontrar o tamanho da coorte para cada século, usando a subconsulta e os cálculos `date_part` vistos anteriormente, apenas para aqueles 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

```

A seguir faremos um cálculo semelhante, com um *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

```

Finalmente, *JOIN* essas duas subconsultas juntas e calcular o percentual de deputados que se tornaram senadores. Um *LEFT JOIN* é usado; esta cláusula é normalmente recomendada para garantir que todas as coortes sejam incluídas, independentemente de o evento subsequente ter ocorrido ou não. Se houver um século em que nenhum deputado se tornou senador, ainda queremos incluir esse século 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

```

Representantes do século 18 eram mais propensos a se tornarem senadores . No entanto, ainda não aplicamos uma caixa de tempo para garantir uma comparação justa. Embora possamos supor com segurança que todos os legisladores que serviram nos séculos 18 e 19 não estão mais vivos, muitos daqueles que foram eleitos pela primeira vez nos séculos 20 e 21 ainda estão no meio de suas carreiras. Adicionar o filtro `WHERE age(c.term_start, b.first_term) <= interval '10 years'` à subconsulta `bb` cria uma caixa de tempo de 10 anos. Observe que a janela pode ser facilmente maior ou menor alterando a constante no intervalo. Um filtro adicional aplicado à subconsulta `a`, `WHERE first_term <= '2009-12-31'`, exclui aqueles 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

```

Com esse novo ajuste, o século XVIII ainda teve a maior parcela de representantes tornando-se senadores em 10 anos, mas o século 21 tem a segunda maior participação, e o século 20 teve uma participação maior do que o 19.

Como 10 anos é um pouco arbitrário, também podemos querer comparar várias janelas de tempo. Uma opção é executar a consulta várias vezes com intervalos diferentes e anotar os resultados. Outra opção é calcular 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 esse resultado, podemos ver como a proporção de deputados que se tornaram senadores evoluiu ao longo do tempo, tanto dentro de cada coorte quanto entre coortes. Além do formato de tabela, o gráfico da saída geralmente revela tendências interessantes. Na [Figura 4-12](#), as coortes baseadas em século são substituídas por coortes baseadas na primeira década, e as tendências ao longo de 10 e 20 anos são mostradas. A conversão de deputados em senadores durante as primeiras décadas da nova legislatura dos Estados Unidos foi claramente diferente dos padrões dos anos seguintes.

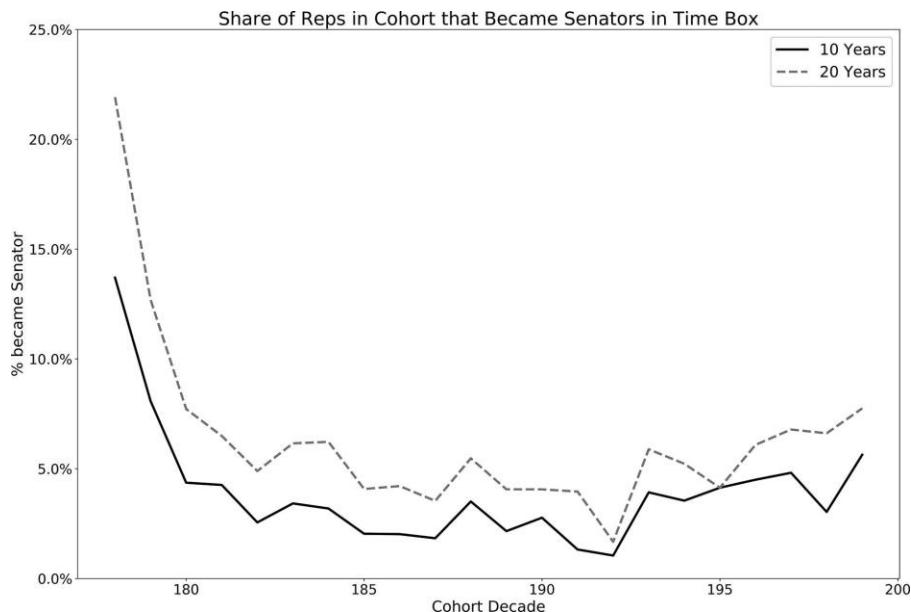


Figura 4-12. Tendência da proporção de representantes para cada coorte, definida por década inicial, que posteriormente se tornaram senadores

Encontrar o comportamento repetido dentro de uma caixa de tempo fixo é uma ferramenta útil para comparar coortes. Isso é particularmente verdadeiro quando os comportamentos são de natureza intermitente, como comportamento de compra ou consumo de conteúdo ou serviço. Na próxima seção, veremos como calcular não apenas se uma entidade teve uma ação subsequente, mas também quantas ações subsequentes ela teve, e as agregaremos com cálculos cumulativos.

## Cálculos Cumulativos

A análise de coorte cumulativa pode ser usada para estabelecer o *valor de vida útil cumulativo*, também chamado de *valor de vida útil do cliente* (os acrônimos CLTV e LTV são usados de forma intercambiável) e para monitorar coortes mais recentes para poder prever qual será seu LTV completo. Isso é possível porque o comportamento inicial geralmente está altamente correlacionado com o comportamento de longo prazo. Os usuários de um serviço que retornam com frequência nos primeiros dias ou semanas de uso tendem a ser os mais propensos a permanecer no longo prazo. Os clientes que compram uma segunda ou terceira vez no início provavelmente continuarão comprando por um período mais longo. Os assinantes que renovam após o primeiro mês ou ano geralmente permanecem por muitos meses ou anos subsequentes.

Nesta seção, falarei principalmente sobre as atividades de geração de receita dos clientes, mas essa análise também pode ser aplicada a situações em que clientes ou entidades incorrem em

custos, como por meio de devoluções de produtos, interações de suporte ou uso de serviços de saúde.

Com cálculos cumulativos, estamos menos preocupados se uma entidade realizou uma ação em uma determinada data e mais com o total de uma determinada data. Os cálculos cumulativos usados neste tipo de análise são mais frequentemente `counts` ou `sums`. Usaremos novamente o conceito de caixa de tempo para garantir comparações de maçãs com maçãs entre coortes. Vejamos o número de mandatos iniciados dentro de 10 anos da primeira `term_start`, coorte dos legisladores por século e tipo de 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 maior coorte é a dos representantes eleitos pela primeira vez no século XIX, mas a coorte com o maior número de mandatos iniciados em 10 anos é a dos representantes eleitos pela primeira vez no século XX. Esse tipo de cálculo pode ser útil para entender a contribuição geral de uma coorte para uma organização. O total de vendas ou o total de compras repetidas podem ser métricas valiosas. Normalmente, porém, queremos normalizar para entender a contribuição por entidade. Os cálculos que podemos querer fazer incluem ações médias por pessoa, valor médio do pedido (AOV), itens por pedido e pedidos por cliente. Para normalizar pelo tamanho da coorte, basta dividir pela coorte inicial, o que fizemos anteriormente com retenção, sobrevivência e devolução.

Aqui fazemos isso e também dinamizamos os resultados em forma de tabela para comparações mais fáceis:

```

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 termos cumulativos normalizados pelo tamanho da coorte, podemos agora confirmar que os representantes eleitos pela primeira vez no século XX tiveram o maior número médio de mandatos, enquanto aqueles que começaram no século 19 tiveram o menor número de mandatos em média. Os senadores têm menos mandatos, mas mais longos do que seus pares representativos, e novamente aqueles que começaram no século 20 tiveram o maior número de mandatos em média.

Os cálculos cumulativos são frequentemente usados nos cálculos do valor da vida útil do cliente. O LTV geralmente é calculado usando medidas monetárias, como o total de dólares gasto por um cliente ou a margem bruta (receita menos custos) gerada por um cliente ao longo de sua vida útil. Para facilitar comparações entre coortes, o “tempo de vida” geralmente é escolhido para refletir o tempo de vida médio do cliente ou períodos que são convenientes para analisar, como 3, 5 ou 10 anos. O conjunto de dados dos legisladores não contém métricas financeiras, mas a troca de valores em dólares em qualquer código SQL anterior seria simples. Felizmente, o SQL é uma linguagem flexível o suficiente para que possamos adaptar esses modelos para abordar uma ampla variedade de questões analíticas.

A análise de coorte inclui um conjunto de técnicas que podem ser usadas para responder perguntas relacionadas ao comportamento ao longo do tempo e como vários atributos podem contribuir para as diferenças entre os grupos. Sobrevida, retorno e cálculos cumulativos esclarecem essas questões. Com uma boa compreensão de como as coortes se comportam, muitas vezes temos que voltar nossa atenção para a composição ou mistura de coortes ao longo do tempo, entendendo como isso pode afetar a retenção total, sobrevida, retorno ou valores cumulativos de tal forma que essas medidas diferem surpreendentemente da coortes individuais.

## Análise de seção transversal, através de uma lente de coorte

Até agora, neste capítulo, vimos a análise de coorte. Acompanhamos o comportamento das coortes ao longo do tempo com análises de retenção, sobrevida, retorno e comportamento cumulativo. Um dos desafios dessas análises, no entanto, é que, mesmo que elas facilitem a identificação das alterações dentro das coortes, pode ser difícil identificar as alterações na composição geral de uma base de clientes ou usuários.

*mix*, que são mudanças na composição da base de clientes ou usuários ao longo do tempo, também podem ocorrer, tornando as coortes posteriores diferentes das anteriores. As mudanças de mix podem ser devido à expansão internacional, alternando entre estratégias de aquisição orgânica e paga, ou passando de um público entusiasta de nicho para um mercado de massa mais amplo. A criação de coortes ou segmentos adicionais ao longo de qualquer uma dessas linhas suspeitas pode ajudar a diagnosticar se uma mudança de mix está acontecendo.

A análise de coorte pode ser contrastada com a análise transversal, que compara indivíduos ou grupos em um único ponto no tempo. Estudos transversais podem correlacionar anos de estudo com renda atual, por exemplo. Do lado positivo, a coleta de conjuntos de dados para análise transversal geralmente é mais fácil, pois não é necessária nenhuma série temporal. A análise transversal pode ser perspicaz, gerando hipóteses para investigações posteriores. Do lado negativo, geralmente existe uma forma de viés de seleção chamado viés de sobrevida, que pode levar a conclusões falsas.

## Viés de sobrevivência

“Vamos olhar para 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. *viés de sobrevivência* é o erro lógico de focar nas pessoas ou coisas que passaram por algum processo de seleção, ignorando aquelas que não passaram. Comumente, isso ocorre porque as entidades não existem mais no conjunto de dados no momento da seleção, porque falharam, mudaram ou deixaram a população por algum outro motivo. Concentrar-se apenas na população restante pode levar a conclusões excessivamente otimistas, porque as falhas são ignoradas.

Muito já foi escrito sobre algumas pessoas que abandonaram a faculdade e iniciaram empresas de tecnologia de grande sucesso. Isso não significa que você deva deixar a faculdade imediatamente, já que a grande maioria das pessoas que desistem não se tornam CEOs bem-sucedidos. Essa parte da população não rende manchetes tão sensacionais, então é fácil esquecer essa realidade.

No contexto do cliente bem-sucedido, o viés de sobrevivência pode aparecer como uma observação de que os melhores clientes tendem a viver na Califórnia ou no Texas e tendem a ter de 18 a 30 anos. Esta é uma grande população para começar, e pode acontecer que essas características sejam compartilhadas por muitos clientes que desistiram antes da data da análise. Voltar para a população original pode revelar que outros dados demográficos, como pessoas de 41 a 50 anos em Vermont, na verdade permanecem e gastam mais ao longo do tempo, embora haja menos deles em termos absolutos. A análise de coorte ajuda a distinguir e reduzir o viés de sobrevivência.

A análise de coorte é uma maneira de superar o viés de sobrevivência, incluindo todos os membros de uma coorte inicial na análise. Podemos tirar uma série de seções transversais de uma análise de coorte para entender como a combinação de entidades pode ter mudado ao longo do tempo. Em qualquer data, usuários de várias coortes estão presentes. Podemos usar a análise transversal para examiná-los, como camadas de sedimentos, para revelar novos insights. No próximo exemplo, criaremos uma série temporal da parcela de legisladores de cada coorte para cada ano no conjunto de dados.

O primeiro passo é encontrar o número de legisladores em exercício a cada ano *date\_dim* a tabela `legislators` ao `date_dim`, *WHERE* a `date` do `date_dim` está entre as datas de início e término de cada mandato. Aqui usamos 31 de dezembro para cada ano para encontrar os legisladores em exercício no final 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, nós adicione os critérios de coorte juntando *JOIN*-se a uma subconsulta com o `first_term`:

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

Finalmente, calculamos a porcentagem do total de `legislators` em cada ano que a coorte do século representa. Isso pode ser feito de duas maneiras, dependendo da forma de saída desejada. A primeira maneira é manter uma linha para cada combinação de `date` e `century` e usar uma função de window `sum` no denominador do cálculo da porcentagem:

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

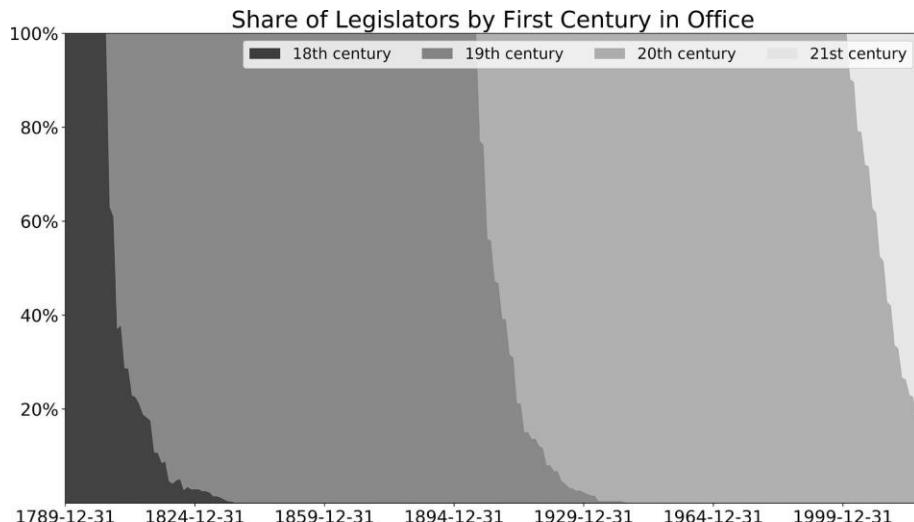
A segunda abordagem resulta em uma linha por ano, com uma coluna para cada século, um formato de tabela que pode ser mais fácil de verificar tendências:

```

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 graficamente a saída, como na [Figura 4-13](#), para ver como coortes mais recentes de os legisladores gradualmente ultrapassam as coortes mais antigas, até que eles próprios sejam substituídos por novas coortes.



*Figura 4-13. Porcentagem de legisladores a cada ano, por século eleitos pela primeira vez*

Em vez de coorte no `first_term`, podemos coorte na posse em vez disso. Encontrar a parcela de clientes que são relativamente novos, de médio prazo ou de longo prazo em vários momentos pode ser esclarecedor. Vamos dar uma olhada em como o mandato dos legisladores no Congresso mudou ao longo do tempo.

O primeiro passo é calcular, para cada ano, o número acumulado de anos de mandato de cada legislador. Como pode haver lacunas entre os mandatos quando os legisladores são eliminados ou deixam o cargo por outros motivos, primeiro encontraremos cada ano em que o legislador esteve no cargo no final do ano, na subconsulta. Em seguida, usaremos uma função de janela `count`, com a janela cobrindo as linhas `unbounded preceding`, ou todas as linhas anteriores para esse legislador, e `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
...	...	...

Em seguida, `count` 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 a porcentagem para cada mandato por ano e ajustar o formato de apresentação, podemos considerar agrupar os mandatos. Um rápido perfil de nossos resultados até agora revela que em alguns anos, quase 40 mandatos diferentes estão representados. Isso provavelmente será 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
...
...

```

Como resultado, podemos deseja agrupar os valores. Não existe uma única maneira correta de agrupar posses. Se houver definições organizacionais de grupos de posse, vá em frente e use-as. Caso contrário, geralmente tento dividir os mandatos em três a cinco grupos de tamanho aproximadamente igual. Aqui vamos agrupar os mandatos em quatro coortes, onde `cume_years` é menor ou igual a 4 anos, entre 5 e 10 anos, entre 11 e 20 anos e igual ou superior a 21 anos:

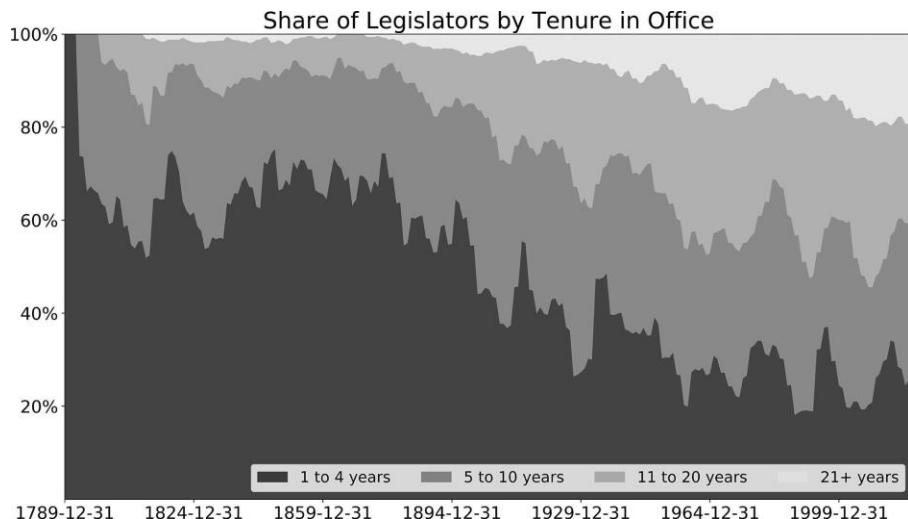
```

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 gráfica dos resultados na [Figura 4-14](#) mostra que nos primeiros anos do país, a maioria dos legisladores tinham muito pouco mandato. Nos anos mais recentes, a proporção de legisladores com 21 ou mais anos de mandato vem aumentando. Há também aumentos periódicos interessantes em legisladores de 1 a 4 anos de mandato que podem refletir mudanças nas tendências políticas.



*Figura 4-14. Percentual de legisladores por número de anos no cargo*

Uma seção transversal de uma população em qualquer momento é composta por membros de várias coortes. Criar uma série temporal dessas seções cruzadas é outra maneira interessante de analisar tendências. Combinar isso com insights de retenção pode fornecer uma imagem mais robusta das tendências em qualquer organização.

## Conclusão

A análise de coorte é uma maneira útil de investigar como os grupos mudam ao longo do tempo, seja da perspectiva de retenção, comportamento repetido ou ações cumulativas. A análise de coorte é retrospectiva, olhando para as populações usando atributos intrínsecos ou atributos derivados do comportamento. Correlações interessantes e úteis podem ser encontradas por meio desse tipo de análise. No entanto, como diz o ditado, correlação não implica causalidade. Para determinar a causalidade real, os experimentos aleatórios são o padrão-ouro. O Capítulo 7 aprofundará a análise de experimentos.

Antes de nos voltarmos para a experimentação, no entanto, temos alguns outros tipos de análise para cobrir. A seguir, abordaremos a análise de texto: os componentes da análise de texto geralmente aparecem em outras análises, e é uma faceta interessante da análise em si.

## CAPÍTULO 5

# Análise de texto

Nos dois últimos capítulos, exploramos as aplicações de datas e números com análise de séries temporais e análise de coorte. Mas os conjuntos de dados geralmente são mais do que apenas valores numéricos e carimbos de data/hora associados. De atributos qualitativos a texto livre, os campos de caracteres geralmente são carregados com informações potencialmente interessantes. Embora os bancos de dados sejam excelentes em cálculos numéricos, como contar, somar e calcular a média, eles também são muito bons em realizar operações em dados de texto.

Começarei este capítulo fornecendo uma visão geral dos tipos de tarefas de análise de texto para as quais o SQL é bom e daquelas para as quais outra linguagem de programação é uma escolha melhor. Em seguida, apresentarei nosso conjunto de dados de avistamentos de OVNIs. Em seguida, entraremos na codificação, cobrindo características de texto e criação de perfil, analisando dados com SQL, fazendo várias transformações, construindo novo texto a partir de partes e, finalmente, encontrando elementos em blocos maiores de texto, inclusive com expressões regulares.

## Por que Análise de Texto com SQL?

Entre os enormes volumes de dados gerados todos os dias, grande parte consiste em texto: palavras, frases, parágrafos e documentos ainda mais longos. Os dados de texto usados para análise podem vir de várias fontes, incluindo descritores preenchidos por humanos ou aplicativos de computador, arquivos de log, tíquetes de suporte, pesquisas de clientes, postagens de mídia social ou feeds de notícias. O texto em bancos de dados varia de *estruturado* (onde os dados estão em diferentes campos de tabela com significados distintos) a *semiestruturado* (onde os dados estão em colunas separadas, mas podem precisar de análise ou limpeza para serem úteis) ou principalmente *não estruturados* (onde campos longos VARCHAR ou BLOB são arbitrários strings de comprimento que requerem uma estruturação extensa antes da análise posterior). Felizmente, o SQL tem várias funções úteis que podem ser combinadas para realizar uma série de tarefas de estruturação e análise de texto.

## O que é análise de texto?

A análise de texto é o processo de derivar significado e insights de dados de texto. Existem duas grandes categorias de análise de texto, que podem ser distinguidas pelo fato de a saída ser qualitativa ou quantitativa. A *análise qualitativa*, que também pode ser chamada *análise textual*, busca compreender e sintetizar o significado de um único texto ou de um conjunto de textos, muitas vezes aplicando outros conhecimentos ou conclusões únicas. Esse trabalho geralmente é feito por jornalistas, historiadores e pesquisadores de experiência do usuário. A *análise quantitativa* de texto também busca sintetizar informações a partir de dados de texto, mas a saída é quantitativa. As tarefas incluem categorização e extração de dados, e a análise geralmente ocorre na forma de contagens ou frequências, geralmente com tendências ao longo do tempo. O SQL é muito mais adequado para a análise quantitativa, então é disso que o restante deste capítulo está preocupado. No entanto, se você tiver a oportunidade de trabalhar com uma contraparte especializada no primeiro tipo de análise de texto, aproveite sua experiência. Combinar o qualitativo com o quantitativo é uma ótima maneira de obter novos insights e persuadir colegas relutantes.

A análise de texto engloba vários objetivos ou estratégias. A primeira é a extração de texto, onde uma parte útil dos dados deve ser extraída do texto ao redor. Outra é a categorização, onde as informações são extraídas ou analisadas de dados de texto para atribuir tags ou categorias a linhas em um banco de dados. Outra estratégia é a análise de sentimentos, onde o objetivo é entender o humor ou intenção do escritor em uma escala de negativo a positivo.

Embora a análise de texto já exista há algum tempo, o interesse e a pesquisa nessa área decolaram com o advento do aprendizado de máquina e dos recursos de computação que muitas vezes são necessários para trabalhar com grandes volumes de dados de texto. O processamento de linguagem natural (NLP) fez grandes avanços no reconhecimento, classificação e até na geração de novos dados de texto. A linguagem humana é incrivelmente complexa, com diferentes idiomas e dialetos, gramáticas e gírias, sem falar nos milhares e milhares de palavras, algumas que têm significados sobrepostos ou modificam sutilmente o significado de outras palavras. Como veremos, o SQL é bom em algumas formas de análise de texto, mas para outras tarefas mais avançadas, existem linguagens e ferramentas mais adequadas.

## Por que o SQL é uma boa opção para análise de texto

Existem várias boas razões para usar o 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 de computação que pode ser aproveitado para tarefas de texto, além das outras tarefas que discutimos até agora. Mover dados para um arquivo simples para análise com outro idioma ou ferramenta é demorado, portanto, fazer o máximo de trabalho possível com SQL no banco de dados tem vantagens.

Se os dados ainda não estiverem em um banco de dados, para conjuntos de dados relativamente grandes, pode valer a pena mover os dados para um banco de dados. Bancos de dados são mais poderosos que planilhas para processar transformações em muitos registros.

O SQL é menos propenso a erros do que as planilhas, pois não é necessário copiar e colar e os dados originais permanecem intactos. Os dados podem ser alterados com um comando *UPDATE*, mas isso é difícil de fazer accidentalmente.

SQL também é uma boa escolha quando o objetivo final é algum tipo de quantificação. Contar quantos tickets de suporte contêm uma frase-chave e analisar categorias de texto maior que será usado para agrupar registros são bons exemplos de quando o SQL brilha. SQL é bom para limpar e estruturar campos de texto. A *limpeza* inclui a remoção de caracteres extras ou espaços em branco, correção de maiúsculas e padronização de ortografia. A *estruturação* envolve a criação de novas colunas a partir de elementos extraídos ou derivados de outros campos ou a construção de novos campos a partir de peças armazenadas em diferentes locais. As funções de string podem ser aninhadas ou aplicadas aos resultados de outras funções, permitindo praticamente qualquer manipulação que possa ser necessária.

O código SQL para análise de texto pode ser simples ou complexo, mas é sempre baseado em regras. Em um sistema baseado em regras, o computador segue um conjunto de regras ou instruções – nem mais, nem menos. Isso pode ser contrastado com o aprendizado de máquina, no qual o computador se adapta com base nos dados. As regras são boas porque são fáceis para os humanos entenderem. Eles são escritos em forma de código e podem ser verificados para garantir que produzam a saída desejada. A desvantagem das regras é que elas podem se tornar longas e complicadas, principalmente quando há muitos casos diferentes para lidar. Isso também pode torná-los difíceis de manter. Se a estrutura ou o tipo de dados inseridos na coluna forem alterados, o conjunto de regras precisará ser atualizado. Em mais de uma ocasião, comecei com o que parecia ser uma simples instrução CASE com 4 ou 5 linhas, apenas para crescer para 50 ou 100 linhas à medida que o aplicativo mudava. As regras ainda podem ser a abordagem correta, mas manter-se em sincronia com a equipe de desenvolvimento sobre as mudanças é uma boa ideia.

Finalmente, o SQL é uma boa escolha quando você sabe com antecedência o que está procurando. Existem várias funções poderosas, incluindo expressões regulares, que permitem pesquisar, extrair ou substituir informações específicas. “Quantos revisores mencionam ‘curta duração da bateria’ em seus comentários?” é uma pergunta que o SQL pode ajudá-lo a responder. Por outro lado, “Por que esses clientes estão com raiva?” não vai ser tão fácil.

## Quando o SQL não é uma boa escolha

SQL essencialmente permite que você aproveite o poder do banco de dados para aplicar um conjunto de regras, embora muitas vezes regras poderosas, a um conjunto de texto para torná-lo mais útil para análise. O SQL certamente não é a única opção para análise de texto, e há vários casos de uso para os quais não é a melhor escolha. É útil estar ciente disso.

A primeira categoria abrange casos de uso para os quais um humano é mais apropriado. Quando o conjunto de dados é muito pequeno ou muito novo, a rotulagem manual pode ser mais rápida e informativa. Além disso, se o objetivo é ler todos os registros e chegar a um resumo qualitativo dos principais temas, um humano é uma escolha melhor.

A segunda categoria é quando há necessidade de pesquisar e recuperar registros específicos que contenham strings de texto com baixa latência. Ferramentas como Elasticsearch ou Splunk foram desenvolvidas para indexar strings para esses casos de uso. O desempenho geralmente será um problema com SQL e bancos de dados; essa é uma das principais razões pelas quais geralmente tentamos estruturar os dados em colunas discretas que podem ser pesquisadas com mais facilidade pelo mecanismo de banco de dados.

A terceira categoria compreende tarefas na categoria mais ampla NLP, onde as abordagens de aprendizado de máquina e as linguagens que as executam, como Python, são uma escolha melhor. A análise de sentimentos, usada para analisar faixas de sentimentos positivos ou negativos em textos, pode ser tratada apenas de maneira simplista com SQL. Por exemplo, “amor” e “ódio” poderiam ser extraídos e usados para categorizar registros, mas dada a gama de palavras que podem expressar emoções positivas e negativas, bem como todas as maneiras de negar essas palavras, seria quase impossível crie um conjunto de regras com SQL para lidar com todos eles. A marcação de parte da fala, onde as palavras em um texto são rotuladas como substantivos, verbos e assim por diante, é melhor tratada com bibliotecas disponíveis em Python. A geração de idioma, ou a criação de um texto totalmente novo com base no aprendizado de textos de exemplo, é outro exemplo melhor tratado em outras ferramentas. Veremos como podemos criar um novo texto concatenando partes de dados, mas o SQL ainda está limitado por regras e não aprenderá e se adaptará automaticamente a novos exemplos no conjunto de dados.

Agora que discutimos os muitos bons motivos para usar SQL para análise de texto, bem como os tipos de casos de uso a serem evitados, vamos dar uma olhada no conjunto de dados que usaremos para os exemplos antes de iniciar o código SQL em si.

## O Conjunto de Dados de Avistamentos de OVNIs

Para os exemplos neste capítulo, usaremos um conjunto de dados de avistamentos de OVNIs compilado pelo [National UFO Reporting Center](#). O conjunto de dados consiste em aproximadamente 95.000 relatórios publicados entre 2006 e 2020. Os relatórios são provenientes de indivíduos que podem inserir informações por meio de um formulário online.

A tabela com a qual trabalharemos é `ufo`, e possui apenas duas colunas. A primeira é uma coluna composta chamada `sighting_report` que contém informações sobre quando o avistamento ocorreu, quando foi relatado e quando foi publicado. Ele também contém metadados sobre a localização, forma e duração do evento de observação. A segunda coluna é um campo de texto chamado `description` que contém a descrição completa do evento. [Figura 5-1](#) mostra uma amostra dos dados.

	sighting_report	description
1	Occurred : 6/2/1980 14:00 [Entered as : 06/24/80 14:00]Reported: 4/6/2006 8:45:00 PM 02:45Posted: 5/15/2006Location: Mount Washington, Ohio... Missing Time: Two PeopleMy mother and I have a long history of UFO sightings/Invovement	
2	Occurred : 4/8/2006 05:05 [Entered as : 04/06/06 05:05]Reported: 4/6/2006 8:01:21 PM 18:09Posted: 5/15/2006Location: Ottoville, Ohio... Bright lights near Ottoville, Ohio-looking westward on SR 189 out of Ft. Jennings, Ohio at th	
3	Occurred : 8/11/2001 08:00 [Entered as : 9/1/01 08:00]Reported: 4/6/2006 8:04:07 PM 16:04Posted: 5/15/2006Location: Erie, PA... 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 8:26:52 PM 24:26Posted: 5/15/2006Location: Leeds, UK... two bright lights in sky - Brightened then dimmed - definitely not a planettwo lights in the sky	
5	Occurred : 4/4/2006 22:00 [Entered as : 4/4/06 22:00]Reported: 4/6/2006 12:16:18 PM 12:16Posted: 5/15/2006Location: Franklin Park, IL... The Light of Life is shining as 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 11:07:07 AM 11:40Posted: 5/15/2006Location: Oklahoma... Two hovering orange circles drop third orange circle over oklahoma.My girlfriend and i were	
7	Occurred : 1/11/2006 15:00 [Entered as : 11/15-06 15:00]Reported: 4/6/2006 9:53:25 AM 09:53Posted: 5/15/2006Location: Utah (USA)... bright changing light hover and landed behind a mountain. lat 37°49'15.6" N lon 115°42'22.1	
8	Occurred : 4/5/2006 22:15 [Entered as : 04/05/06 22:15]Reported: 4/5/2006 11:28:51 PM 23:24Posted: 5/15/2006Location: Graham, North Carolina... At first glance, it looked like a plane, we realized that it wasn't moves, but dancing in place	
9	Occurred : 10/1/1994 14:00 [Entered as : 10/15/94 14:00]Reported: 4/6/2006 8:32:00 AM 08:32Posted: 5/15/2006Location: Circle, New Mexico... Silver disc saucer hoverd above town, then disappeared was driving my car with three friend	
10	Occurred : 4/15/1973 21:30 [Entered as : 04/15/73 21:30]Reported: 4/5/2006 8:24:26 PM 20:59Posted: 5/15/2006Location: Cold Lake, Alberta, Canada... Strange Lights Indeed Upon returning home on a beautiful spring evening, in the approximatly	
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, PA... Three disk like objects flew over Erie, PA at around 3:15, there were lights and not any radio	
12	Occurred : 10/8/2005 06:15 [Entered as : 10/08/05 06:15]Reported: 4/6/2006 8:24:23 AM 06:24Posted: 5/15/2006Location: Orlando, Florida... 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: Torrance, California... square shaped object over Southern CaliforniaDriving home from work, from the corner of 7th and	
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, Arizona... 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 : 01/01/06 04:45]Reported: 4/6/2006 12:22:32 AM 00:22Posted: 5/15/2006Location: Claremore, Oklahoma... Bright geynchronous light that appears every morning.On 01/01/06 at approx. 0445, my pal	
16	Occurred : 4/5/2006 08:31 [Entered as : 4:5-06 8:31]Reported: 4/5/2006 10:42:31 PM 22:40Posted: 5/15/2006Location: Shingletown, Pennsylvania... scary! saw amassive blue object floating in the sky for about 60 seconds the n split 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 (USA)... noisy rectangular object falling from the sky and crash landed 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 10:42:31 PM 20:15Posted: 5/15/2006Location: Centralia, Washington... Sighted over Cooks Hill Road and Joppisch Road just west of I-5. Dark clear sky. Large fire	
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 Marcos... Flying triangle in sky over San Marcos looked up thinking that I was seeing a plane, but it w	
20	Occurred : 4/5/2006 20:30 [Entered as : 04/05/06 20:30]Reported: 4/5/2006 8:49:44 PM 20:49Posted: 5/15/2006Location: Des Moines, Iowa... Weird fireball that just disappeared without fanfare and followed up by 4 fighters.Say it strel	
21	Occurred : 9/14/1986 20:30 [Entered as : 09/14/86 20:30]Reported: 4/5/2006 8:20:41 PM 20:20Posted: 5/15/2006Location: Lublin, 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... while square object moving very fastThis thing was unlike anything I have ever seen. Instead	

Figura 5-1. Amostra da tabela ufo

Por meio dos exemplos e da discussão neste capítulo, mostrarei como analisar a primeira coluna em datas estruturadas e descritores. Também mostrarei como realizar várias análises em campo **description**. Se eu estivesse trabalhando com esses dados continuamente, poderia considerar a criação de um pipeline ETL, um trabalho que processa os dados da mesma maneira regularmente e armazenando os dados estruturados resultantes em uma nova tabela. Para os exemplos deste capítulo, no entanto, ficaremos com a tabela bruta.

Vamos entrar no código, começando com SQL para explorar e caracterizar o texto dos avistamentos.

## Características do texto

O tipo de dado mais flexível em um banco de dados é VARCHAR, porque quase todos os dados podem ser colocados em campos desse tipo. Como resultado, os dados de texto em bancos de dados vêm em uma variedade de formas e tamanhos. Assim como em outros conjuntos de dados, criar perfis e caracterizar os dados é uma das primeiras coisas que fazemos. A partir daí, podemos desenvolver um plano de jogo para os tipos de limpeza e análise que podem ser necessários para a análise.

Uma forma de conhecer os dados do texto é encontrar o número de caracteres em cada valor, o que pode ser feito com a função **length** (ou **len** em alguns bancos de dados). Esta função recebe o campo de string ou caractere como argumento e é semelhante a funções encontradas em outras linguagens e programas de planilhas:

```
SELECT length('Sample string');
```

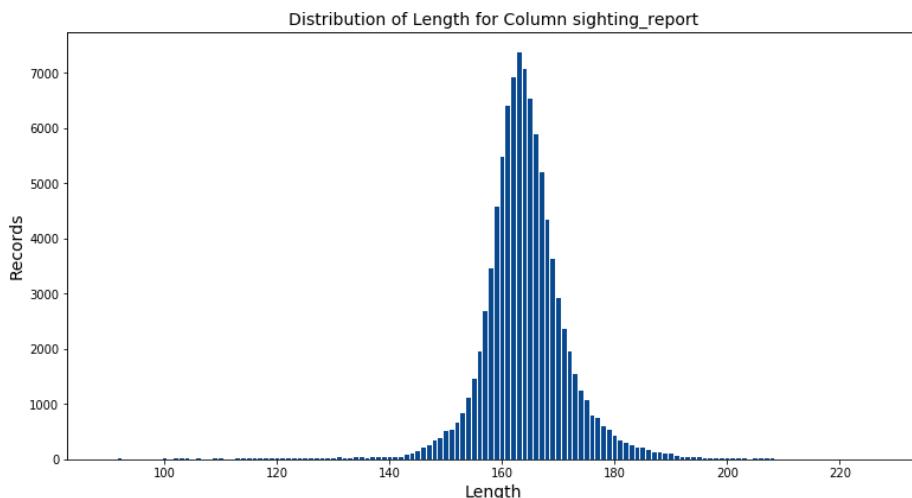
```
length
-----
13
```

Podemos criar uma distribuição de comprimentos de campo para ter uma noção do comprimento típico e se há algum outlier extremo que pode precisar ser tratado 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 150 e 180 caracteres, e muito poucos têm menos de 140 ou mais de 200 caracteres. Os comprimentos do campo **description** variam de 5 a 64.921 caracteres. Podemos supor que há muito mais variedade neste campo, mesmo antes de fazer qualquer perfil adicional.



*Figura 5-2. Distribuição de comprimentos de campo na primeira coluna da tabela ufo*

Vamos dar uma olhada em algumas linhas de amostra da coluna **sighting\_report**. Em uma ferramenta de consulta, posso rolar por cerca de cem linhas para me familiarizar com o conteúdo, mas estes são representativos dos valores na 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
```

Estes dados é o que eu chamaria de semiestruturado, ou overstuffed. Ele não pode ser usado em uma análise como está, mas há informações claramente distintas armazenadas aqui, e o padrão é semelhante entre as linhas. Por exemplo, cada linha tem a palavra "Ocorreu" seguida pelo que parece ser um carimbo de data/hora, "Local" seguido por um local e "Duração" seguido por uma quantidade de tempo.



Os dados podem acabar em campos sobrecarregados por vários motivos, mas vejo dois comuns. Uma é quando não há campos suficientes disponíveis no sistema ou aplicativo de origem para armazenar todos os atributos necessários, de modo que vários atributos são inseridos no mesmo campo. Outra é quando os dados são armazenados em um blob JSON em um aplicativo para acomodar atributos esparsos ou adições frequentes de novos atributos. Embora ambos os cenários sejam menos do que ideais do ponto de vista de análise, desde que haja uma estrutura consistente, geralmente podemos lidar com isso com SQL.

Nossa próxima etapa é tornar esse campo mais utilizável analisando-o em vários novos campos, cada um contendo uma única informação. As etapas deste processo são:

- Planejar o(s) campo(s) desejado(s) como saída
- Aplicar funções de análise
- Aplicar transformações, incluindo conversões de tipo de dados
- Verificar os resultados quando aplicados a todo o conjunto de dados, pois geralmente haverá alguns registros que não estão de acordo com o padrão
- Repita essas etapas até que os dados estejam nas colunas e formatos desejados

As novas colunas que analisaremos `sighting_report` são `occurred`, `entered_as`, `reported`, `posted`, `location`, `shape`, e `duration`. Em seguida, aprenderemos sobre funções de análise e trabalharemos na estruturação do conjunto de dados `ufo`.

## Análise de texto

A análise de dados com SQL é o processo de extração de partes de um valor de texto para torná-las mais úteis para análise. A análise divide os dados na parte que queremos e “todo o resto”, embora normalmente nosso código retorne apenas a parte que queremos.

As funções de análise mais simples retornam um número fixo de caracteres do início ou do fim de uma string. A função `left` retorna caracteres do lado esquerdo ou do início da string, enquanto a função `right` retorna caracteres do lado direito ou do final da string. Caso contrário, eles funcionam da mesma maneira, tomando o valor a ser analisado como o primeiro argumento e o número de caracteres como o segundo. Qualquer argumento pode ser um campo de banco de dados ou cálculo, permitindo 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 analisar a primeira palavra, “Ocorreu”, 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 que todos os registros começam com essa palavra, o que é uma boa notícia porque significa que pelo menos essa parte do padrão é consistente. No entanto, o que realmente queremos são os valores para o que ocorreu, não a palavra em si, então vamos tentar novamente. No primeiro registro de exemplo, o final do timestamp ocorreu no caractere 25. Para remover “Occurred” e reter apenas o timestamp real, podemos retornar os 14 caracteres mais à 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 isso retorne o resultado correto para o primeiro registro, infelizmente ele não pode lidar com os registros que têm valores de mês ou dia de dois dígitos. Poderíamos aumentar o número de caracteres retornados pelas funções `left` e `right`, mas o resultado incluiria muitos caracteres para o primeiro registro.

As funções `left` e `right` são úteis para extrair partes de comprimento 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 por trás dessa função é dividir uma string em partes com base em um delimitador e permitir que você selecione uma parte específica. Um *delimitador* é um ou mais caracteres usados para especificar o limite entre regiões de texto ou outros dados. O delimitador de vírgula e o delimitador de tabulação são provavelmente os mais comuns, pois 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 análise. A forma da função é:

```
split_part(string or field name, delimiter, index)
```

O índice é a posição do texto a ser retornado, em relação ao delimitador. Então `index = 1` retorna todo o texto à esquerda da primeira instância do delimitador, `index = 2` retorna o texto entre a primeira e a segunda instância do delimitador (ou todo o texto à direita do delimitador se o delimitador aparece apenas uma vez), e assim por diante. Não há índice 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
```



MySQL tem uma função `substring_index` ao invés de `split_part`. O SQL Server não tem uma função `split_part` os

Observe que os espaços no texto serão mantidos, a menos que especificados como parte do delimitador. Vamos dar uma olhada em como podemos analisar os elementos da coluna `sighting_report`. Como lembrete, um valor de amostra tem esta aparência:

```
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 entre “Occurred :” e “(Entered). Ou seja, queremos a string “6/3/2014 23:00”. Verificando o texto de exemplo, “Occurred :” e “(Entered” aparecem apenas uma vez. Dois pontos (:) aparecem várias vezes, tanto para separar o rótulo do valor quanto no meio dos carimbos de data/hora. Isso pode dificultar a análise usando os dois pontos. O caractere de parêntese aberto aparece apenas uma vez. Temos algumas opções sobre o que especificar como delimitador, escolhendo strings mais longas ou apenas o menor número de caracteres necessários para dividir a string com precisão. Eu costumo ser um pouco mais detalhado para garantir que eu obtenha exatamente a peça que eu quero, mas isso realmente depende da situação.

Primeiro, divida `sighting_report` em “Occurred :” e verifique 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 extra. Vamos verificar o resultado quando dividimos em “(Entered”:

```
SELECT split_part(sighting_report,' (Entered',1) as split_2
FROM ufo
;

split_2
-----
Occurred : 6/3/2014 23:00
```

Isso está mais próximo, mas ainda tem o rótulo no resultado. Felizmente, as funções de aninhamento `split_part` retornarão 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

Agora o resultado inclui os valores desejados. A revisão de algumas linhas adicionais mostra que os valores de dia e mês de dois dígitos são tratados adequadamente, assim como as datas que não têm um valor de hora. Acontece que alguns registros omitem o valor “Inserido como”, então uma divisão adicional é necessária para lidar com registros onde o rótulo “Reportado” marca o final 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 `occurred` analisados com o código SQL são representados graficamente na Figura 5-3.

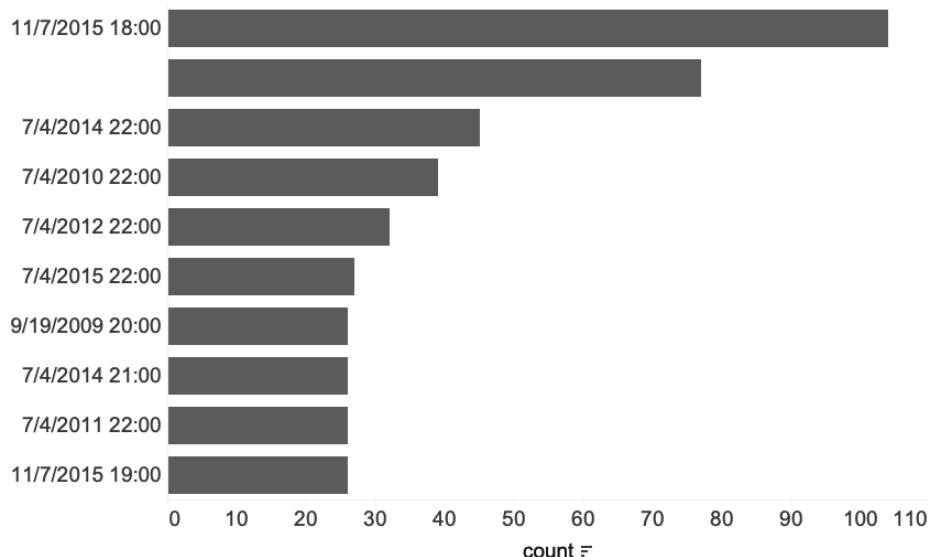


Figura 5-3. Os 10 valores mais comuns `occurred` para avistamentos de OVNIs



Encontrar um conjunto de funções que funcione para todos os valores no conjunto de dados é uma das partes mais difíceis da análise de texto. Muitas vezes, são necessárias várias rodadas de tentativa e erro e o perfil dos resultados ao longo do caminho para acertar.

A próxima etapa é aplicar regras de análise semelhantes para extrair os outros campos desejados, usando delimitadores iniciais e finais para isolar apenas a parte relevante da string. A consulta final usa várias vezes `split_part`, com argumentos diferentes 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 essa análise SQL, os dados agora estão em um formato muito mais estruturado e utilizável. Antes de terminarmos, no entanto, existem algumas transformações que limpam os dados um pouco mais. Vamos dar uma olhada nessas funções de transformação de strings a seguir.

## Transformações de Texto

As transformações alteram os valores da string de alguma forma. Vimos várias funções de transformação de data e hora no [Capítulo 3](#). Existe um conjunto de funções em SQL que trabalham especificamente em valores de string. Eles são úteis para trabalhar com dados analisados, mas também para quaisquer dados de texto em um banco de dados que precisem ser ajustados ou limpos para análise.

Entre as transformações mais comuns estão as que alteram a capitalização. A função `upper` converte todas as letras em sua forma maiúscula, enquanto a função `lower` converte todas as letras em sua forma minúscula. Por exemplo:

```
SELECT upper('Some sample text');
```

```
upper
-----
SOME SAMPLE TEXT
```

```
SELECT lower('Some sample text');
```

```
lower
-----
some sample text
```

Estes são úteis para padronizar valores que podem ter sido inseridos de maneiras diferentes. Por exemplo, qualquer humano reconhecerá que “California”, “caLiforNia” e “CALIFORNIA” se referem ao mesmo estado, mas um banco de dados os tratará como valores distintos. Se fôssemos contar os avistamentos de OVNIs por estados com esses valores, teríamos três registros para a Califórnia, resultando em conclusões de análise incorretas. Convertê-los para todas as letras maiúsculas ou todas as letras minúsculas resolveria esse problema. Alguns bancos de dados, incluindo o Postgres, têm uma função `initcap` que coloca em maiúscula a primeira letra de cada palavra em uma string. Isso é útil para nomes próprios, como nomes de estado:

```
SELECT initcap('caLiforNia'), initcap('golden gate bridge');
```

```
initcap      initcap
-----      -----
California  Golden Gate Bridge
```

O campo `shape` no conjunto de dados que analisamos contém um valor que está em todos maiúsculas, “TRIANGULAR”. Para limpar isso e padronizá-lo com os demais valores, todos com apenas a primeira letra maiúscula, 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 para cada forma é mostrado na Figura 5-4. A luz é de longe a forma mais comum, seguida por círculo e triângulo. Alguns avistamentos não relatam uma forma, portanto, uma contagem para valor nulo também aparece no gráfico.

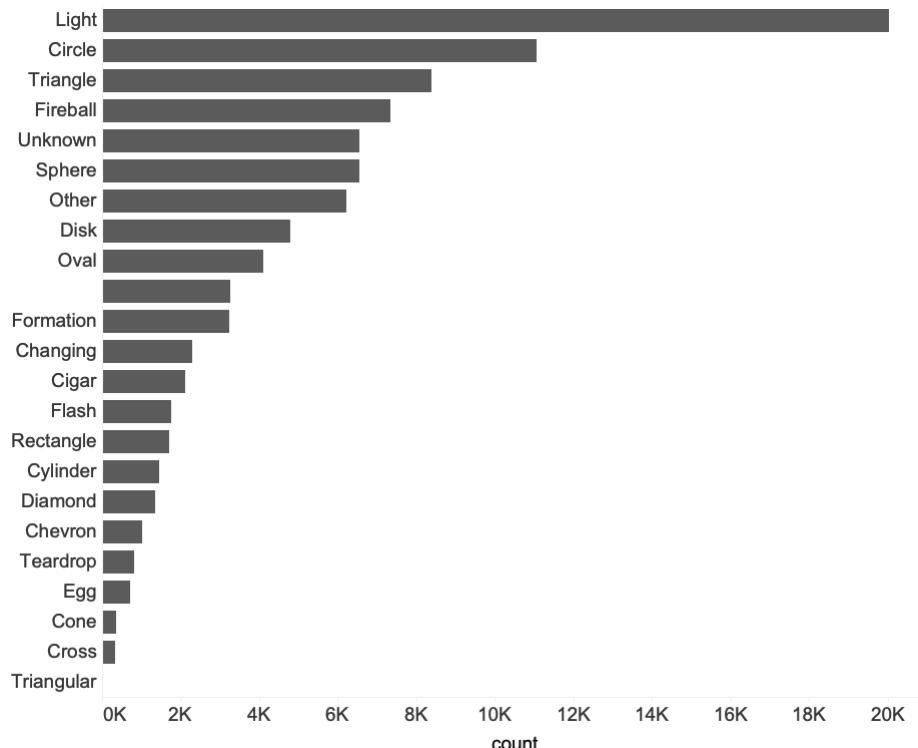


Figura 5-4. Frequência de formas em avistamentos de OVNIs

Outra função de transformação útil é uma chamada `trim` que remove espaços em branco no início e no final de uma string. Caracteres de espaço em branco extras são um problema comum ao analisar valores de strings mais longas ou quando os dados são criados por entrada humana ou copiando dados de um aplicativo para outro. Como exemplo, podemos remover os espaços iniciais antes de “California” na seguinte string usando a função `trim`:

```
SELECT trim(' California ');
trim
-----
California
```

A função `trim` tem alguns parâmetros opcionais que o tornam flexível para uma variedade de desafios de limpeza de dados. Primeiro, ele pode remover caracteres do início de uma string ou do final de uma string, ou ambos. Aparar de ambas as extremidades é o padrão, mas as outras opções podem ser especificadas com `leading` ou `trailing`. Além disso, `trim` pode remover qualquer caractere, não apenas o espaço em branco. Assim, por exemplo, se um aplicativo colocasse um cifrão (\$) no início de cada nome de estado por algum motivo, poderíamos removê-lo com `trim`:

```
SELECT trim(leading '$' from '$California');
```

Alguns dos valores no campo `duration` têm espaços à esquerda, então 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 para as durações mais comuns estão representados graficamente na [Figura 5-5](#). Avistamentos com duração entre 1 e 10 minutos são comuns. Alguns avistamentos não informam uma duração, portanto, uma contagem para valor nulo aparece no gráfico.

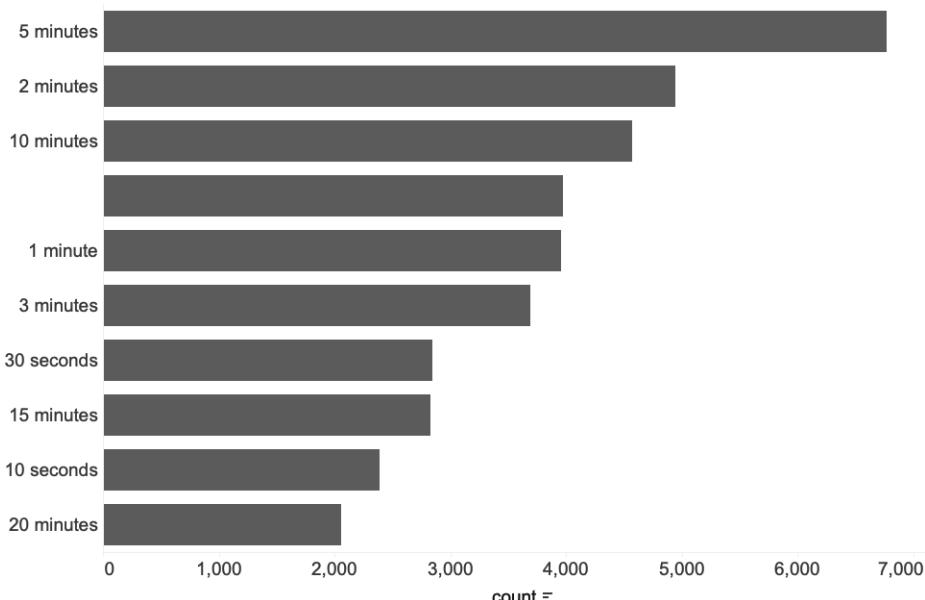


Figura 5-5. As 10 durações mais comuns de avistamentos de OVNIs

O próximo tipo de transformação é uma conversão de tipo de dados. Esse tipo de transformação, discutido no [Capítulo 2](#), será útil para garantir que os resultados de nossa análise tenham o tipo de dados pretendido. No nosso caso, existem dois campos que devem ser tratados como timestamps — as colunas `occurred` e `reported` — e a coluna `posted` deve ser do tipo data. Os tipos de dados podem ser alterados com conversão, usando o operador dois-pontos duplos (::) ou a sintaxe `CAST field as type`. vamos deixar os valores `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(
                    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)
            as reported
        ,location
        ,shape
        ,duration
    ) as entered_as
)
```

```

        , , ' 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 é convertida para os novos formatos. Observe que o banco de dados adiciona os segundos ao carimbo de data/hora, embora haja não havia segundos no valor original e reconhece corretamente as datas que estavam no formato mês/dia/ano (mm/dd/aaaa) é um problema ao aplicar essas transformações a todo o conjunto de dados, no entanto. Alguns registros não têm nenhum valor, aparecendo como uma string vazia, e alguns têm o valor de hora, mas nenhuma data associada a eles. Embora uma string vazia e null pareçam conter as mesmas informações – nada – os bancos de dados os tratam de maneira diferente. Uma string vazia ainda é uma string e não pode ser convertida em outro tipo de dados. Definir todos os registros não conformes como nulos com uma instrução CASE permite que a conversão de tipo funcione corretamente. Como sabemos que as datas devem conter pelo menos oito caracteres (quatro dígitos para ano, um ou dois dígitos para mês e dia e dois caracteres “-” ou “/”), uma maneira de fazer isso é definir qualquer registro com LENGTH menor que 8 igual a nulo com 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
(

```

---

<sup>1</sup> Como o conjunto de dados foi criado nos Estados Unidos, ele está no formato mm/dd/aaaa. Muitos outros países do mundo usam o formato dd/mm/aaaa. Sempre vale a pena verificar sua fonte e ajustar seu código conforme necessário.

```

SELECT
    split_part(
        split_part(
            split_part(sighting_report,'(Entered',1)
            , 'Occurred : ',2)
            , 'Reported',1) as occurred
        ,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 transformação final que discutirei nesta seção é a função `replace`. Às vezes, há uma palavra, frase ou outra string dentro de um campo que gostaríamos de alterar para outra string ou remover completamente. A função `replace` é útil para esta tarefa. São necessários três argumentos — o texto original, a string para localizar e a string para substituir em seu lugar:

```
replace(string or field, string to find, string to substitute)
```

Então, por exemplo, se quisermos alterar as referências de “ objetos voadores não identificados” para “OVNIs”, podemos usar a função `replace`:

```
SELECT replace('Some unidentified flying objects were noticed
above...', 'unidentified flying objects', 'UFOs');
```

```
replace
-----
Some UFOs were noticed above...
```

Esta função irá encontrar e substituir todas as instâncias do string no segundo argumento, independentemente de onde ele aparece. Uma string vazia pode ser usada como terceiro argumento, que é uma boa maneira de remover partes de uma string que não são desejadas. Como outras funções de string, a `replace` pode ser aninhada, com a saída de uma `replace` se tornando a entrada de outra.

No conjunto de dados de avistamento de OVNIs com o qual estamos trabalhando, alguns dos valores `location` incluem qualificadores que indicam que o avistamento ocorreu “perto”, “perto de” ou “fora de” uma cidade ou vila. Podemos usar `replace` para padronizá-los para “próximo”:

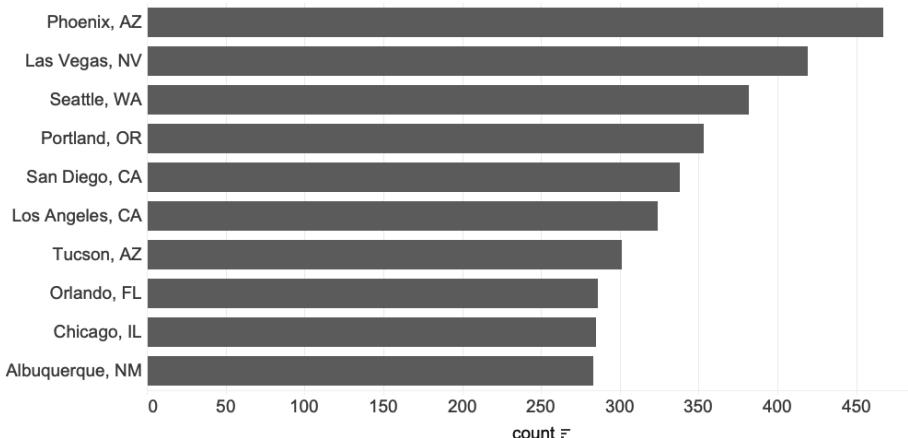
```

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 observação são representados graficamente na [Figura 5-6](#).



*Figura 5-6. Locais mais comuns de avistamentos de OVNIs*

Agora nós analisamos e limpamos todos os elementos do campo `sighting_report` em colunas distintas e apropriadamente tipadas. O código final se parece com isso:

```

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	duration
-----	-----	-----	-----	-----	-----	-----
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
...	...	...	...	...	...	...

Este pedaço de código SQL pode ser reutilizado em outras consultas, ou pode ser usado para copiar o UFO bruto dados em uma nova tabela limpa.

Alternativamente, ele pode ser transformado em uma exibição ou colocado em uma expressão de tabela comum para reutilização. O Capítulo 8 discutirá essas estratégias com mais detalhes.

Vimos como aplicar funções de análise e transformação para limpar e melhorar o valor da análise de dados de texto que possuem alguma estrutura. Em seguida, veremos o outro campo no conjunto de dados de avistamentos de OVNIs, o campo de texto livre `description`, e aprenda como usar funções SQL para procurar elementos específicos.

## Encontrando elementos em blocos maiores de texto

A análise e as transformações são operações comuns aplicadas a dados de texto para prepará-los para análise. Outra operação comum com dados de texto é encontrar strings em blocos de texto maiores. Isso pode ser feito para filtrar resultados, categorizar registros ou substituir as strings pesquisadas por valores alternativos.

### Correspondências de caracteres curinga: LIKE, ILIKE

SQL tem várias funções para combinar padrões dentro de strings. O operador LIKE corresponde ao padrão especificado na string. Para permitir que ele corresponda a um padrão e não apenas encontre uma correspondência exata, símbolos curinga podem ser adicionados antes, depois ou no meio do padrão. O curinga “%” corresponde a zero ou mais caracteres, enquanto o curinga “\_” corresponde a exatamente um caractere. Se o objetivo é corresponder ao “%” ou “\_” em si, coloque o símbolo de escape de 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 na instrução SQL. Ele pode ser usado para filtrar registros na cláusula `WHERE`. Por exemplo, alguns repórteres mencionam que estavam com um cônjuge na época e, portanto, podemos querer descobrir quantos relatórios mencionam a palavra “esposa”. Como queremos encontrar a string em qualquer lugar no texto da descrição, colocaremos 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, isso retornará apenas correspondências na string minúscula. E se alguns repórteres mencionarem “Wife” ou deixarem Caps Lock ativado e digitarem “WIFE”? Existem duas opções para tornar a pesquisa insensível a maiúsculas e minúsculas. Uma opção é transformar o campo a ser pesquisado usando a função `upper` ou `lower` discutida na seção anterior, o que tem o efeito de tornar a pesquisa insensível, pois os caracteres são todos maiúsculos ou minúsculos:

```
SELECT count(*)
FROM ufo
WHERE lower(description) like '%wife%'
;

count
-----
6439
```

Outra maneira de fazer isso é com o operador ILIKE, que é efetivamente um operador LIKE que não diferencia maiúsculas de minúsculas. A desvantagem é que não está disponível em todos os bancos de dados; notavelmente, MySQL e SQL Server não o suportam. No entanto, é uma opção de sintaxe compacta e agradável se você estiver trabalhando em um banco de dados que a suporta:

```
SELECT count(*)
FROM ufo
WHERE description ilike '%wife%'
;

count
-----
6439
```

Qualquer uma dessas variações de LIKE e ILIKE pode ser negada com NOT. Assim, por exemplo, para encontrar os registros que não mencionam “esposa”, podemos usar NOT LIKE:

```
SELECT count(*)
FROM ufo
WHERE lower(description) not like '%wife%'
;

count
-----
89024
```

A filtragem em várias strings é possível com os operadores AND e OR:

```
SELECT count(*)
FROM ufo
WHERE lower(description) like '%wife%'
```

```

or lower(description) like '%husband%'
;
count
-----
10571

```

Tenha cuidado ao usar parênteses para controlar a ordem das operações ao usar OR em conjunto com os operadores AND, ou você poderá obter resultados inesperados. Por exemplo, essas cláusulas *WHERE* não retornam o mesmo resultado, pois 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 filtrar nas cláusulas *WHERE* ou *JOIN...ON*, LIKE pode ser usado na cláusula *SELECT* para categorizar ou agregar determinados registros. Vamos começar com a categorização. O operador LIKE pode ser usado em uma instrução CASE para rotular e agrupar registros. Algumas das descrições mencionam uma atividade que o observador estava fazendo durante 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, enquanto muitas pessoas não relatam avistamentos enquanto nadam ou andam de bicicleta. Isso talvez não seja surpreendente, uma vez que essas atividades são simplesmente menos comuns do que dirigir.



Embora os valores derivados por meio de funções de transformação de análise de texto possam ser usados em critérios JOIN, o desempenho do banco de dados geralmente é um problema. Considere analisar e/ou transformar em uma subconsulta e, em seguida, unir o resultado com uma correspondência exata na cláusula JOIN.

Observe que esta instrução CASE rotula cada descrição com apenas uma das atividades e avalia se cada registro corresponde ao padrão na ordem em que a instrução é escrita. Uma descrição que contenha "condução" e "caminhada" será rotulada como "condução". Isso é apropriado em muitos casos, mas principalmente ao analisar textos mais longos, como de revisões, comentários de pesquisas ou tiquetes de suporte, a capacidade de rotular registros com várias categorias é importante. Para este tipo de caso de uso, uma série de colunas binárias ou sinalizadoras BOOLEAN é chamada.

Vimos anteriormente que LIKE pode ser usado para gerar uma resposta BOOLEAN de TRUE ou FALSE, e podemos usar isso para rotular linhas. No conjunto de dados, várias descrições mencionam a direção em que o objeto foi detectado, como norte ou sul, e algumas mencionam mais de uma direção. Podemos querer 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 encontrar a frequência de várias combinações de direções ou para descobrir quando uma direção é usada sem nenhuma outra direção na mesma descrição.

Todas as combinações são úteis em alguns contextos, principalmente na construção de conjuntos de dados que serão usados por outros para explorar os dados, ou em uma ferramenta de BI ou visualização. No entanto, às vezes é mais útil resumir ainda mais os dados e realizar uma agregação nos registros que contêm um padrão de string. Aqui vamos contar os registros, mas outras agregações como `sum` e `average` podem ser usadas se o conjunto de dados contiver outros campos numéricos, como números de vendas:

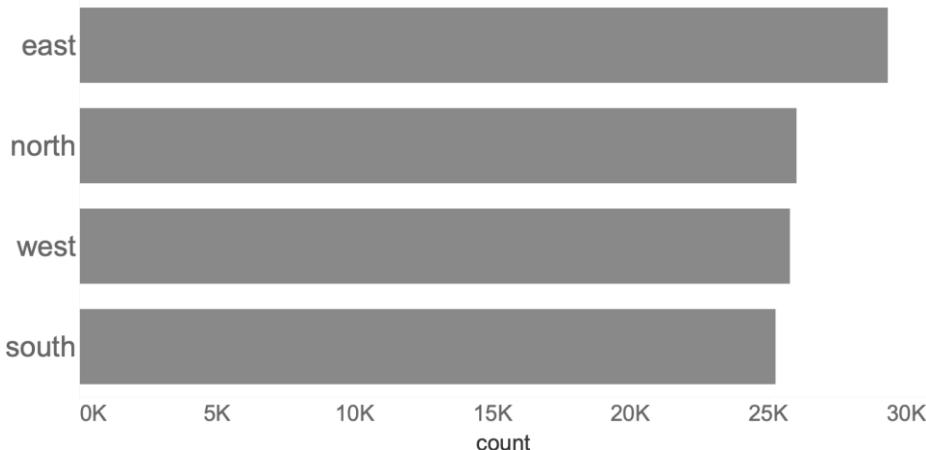
```

SELECT
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 dos termos de direção no campo de descrição e podemos ver que “leste” é mencionado com mais frequência do que outras direções. Os resultados estão representados graficamente na Figura 5-7.



*Figura 5-7. Frequência das direções da bússola mencionadas nos relatórios de avistamento de OVNIs*

Na consulta anterior, ainda permitimos que um registro que contenha mais de uma direção seja contado mais de uma vez. No entanto, não há mais visibilidade sobre quais combinações específicas existem. A complexidade pode ser adicionada à consulta conforme necessário para lidar com esses casos, com uma declaração como:

```
count(case when description ilike '%east%'  
and description ilike '%north%' then 1 end) as east
```

Correspondências de padrões com LIKE, NOT LIKE e ILIKE são flexíveis e podem ser usados em vários lugares em uma consulta SQL para filtrar, categorizar e agrregar dados para uma variedade de necessidades de saída. Esses operadores podem ser usados em combinação com as funções de análise e transformação de texto que discutimos anteriormente para obter ainda mais flexibilidade. Em seguida, discutirei como lidar com vários elementos quando as correspondências são exatas antes de retornar a mais padrões em uma discussão sobre expressões regulares.

## Correspondências Exatas: IN, NOT IN

Antes de passarmos para a correspondência de padrões mais complexa com expressões regulares, vale a pena examinar alguns operadores adicionais que são úteis na análise de texto. Embora não sejam estritamente sobre correspondência de padrões, eles geralmente são úteis em combinação com LIKE e seus parentes para criar um conjunto de regras que inclua exatamente o conjunto correto de resultados. Os operadores são IN e sua negação, NOT IN. Eles permitem que você especifique uma lista de correspondências, resultando em um código mais compacto.

Vamos imaginar que estamos interessados em categorizar os avistamentos com base na primeira palavra da `description`. Podemos encontrar a primeira palavra usando a função `split_part`, com um caractere de espaço como delimitador. Muitos relatórios começam com uma cor como primeira palavra.

Podemos querer filtrar os registros para dar uma olhada nos relatórios que começam por nomear uma cor. Isso pode ser feito listando cada cor com uma construção 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...
...
...

```

O uso de uma lista IN é mais compacto e geralmente menos propenso a erros, principalmente quando há outros elementos na cláusula *WHERE*. IN leva uma lista de itens separados por vírgulas para corresponder. O tipo de dados dos elementos deve corresponder ao tipo de dados da coluna. Se o tipo de dados for numérico, os elementos devem ser números; se o tipo de dados for texto, os elementos devem ser citados como texto (mesmo que o elemento seja 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 são idênticas em seus resultados, e as freqüências são mostradas na [Figura 5-8](#).

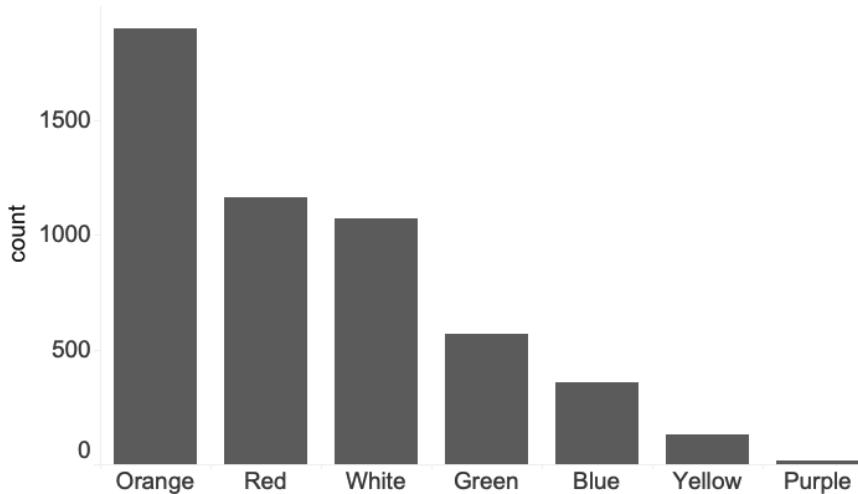


Figura 5-8. Frequência das cores selecionadas usadas como a primeira palavra nas descrições de avistamentos de OVNIs

O principal benefício de IN e NOT IN é que eles tornam o código mais compacto e legível. Isso pode ser útil ao criar categorizações mais complexas na cláusula *SELECT*. Por exemplo, imagine que queremos categorizar e contar os registros pela primeira palavra em cores, formas, movimentos ou outras palavras possíveis. Podemos chegar a algo como o seguinte que combina elementos de análise sintática, 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 categorizar com precisão os relatórios pela primeira palavra. O SQL permite que você crie uma variedade de expressões complexas e diferenciadas para lidar com dados de texto. A seguir, veremos algumas maneiras ainda mais sofisticadas de trabalhar com dados de texto em SQL, usando expressões regulares.

## Expressões Regulares

Existem várias maneiras de corresponder padrões em SQL. Um dos métodos mais poderosos, embora também confuso, é o uso de expressões regulares (regex). Admito que considero as expressões regulares intimidantes e evitei usá-las por muito tempo em minha carreira de análise de dados. Em uma pitada, tive a sorte de ter colegas dispostos a compartilhar trechos de código e desembaraçar meu trabalho. Foi só quando acabei com um grande projeto de análise de texto que decidi que finalmente era hora de aprender sobre eles.

*Expressões regulares* são sequências de caracteres, muitos com significados especiais, que definem padrões de pesquisa. O principal desafio no aprendizado de regex e no uso e manutenção do código que o contém é que a sintaxe não é particularmente intuitiva. Os trechos de código não lêem nada como uma linguagem humana, ou mesmo como linguagens de computador como SQL ou Python. Com um conhecimento prático dos caracteres especiais, no entanto, o código pode ser escrito e decifrado. Assim como o código para todas as nossas consultas, é uma boa ideia começar com simplicidade, desenvolver a complexidade conforme necessário e verificar os resultados à medida que avança. E deixe comentários liberalmente, tanto para outros analistas quanto para você no futuro.

Regex é uma linguagem, mas é usada apenas em outras linguagens. Por exemplo, expressões regulares podem ser chamadas em 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. A sintaxe nem sempre é exatamente a mesma, mas como em outras funções, uma vez que você tenha uma noção das possibilidades, deve ser possível ajustar a sintaxe ao seu ambiente.

Uma explicação completa e toda a sintaxe e maneiras de usar regex estão além do escopo deste livro, mas mostrarei o suficiente para você começar e realizar várias tarefas comuns em SQL. Para uma introdução mais completa, *Learning Regular Expressions* por Ben Forta (O'Reilly) é uma boa escolha. Aqui, começarei apresentando as maneiras de indicar ao banco de dados que você está usando um regex e, em seguida, apresentarei a sintaxe, antes de passar para alguns exemplos de como o regex pode ser útil na análise de relatórios de avistamento de OVNIs.

Regex pode ser usado em instruções SQL de duas maneiras. O primeiro é com comparadores POSIX e o segundo é com funções regex. POSIX significa Portable Operating System Interface e refere-se a um conjunto de padrões IEEE, mas você não precisa saber mais do que isso 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 BOOLEAN, TRUE ou FALSE. Observe que, embora não contenha nenhuma sintaxe especial, “data” é uma regex. As expressões regulares também podem conter cadeias de texto normais. Este exemplo é semelhante ao que poderia ser feito com um operador LIKE. O comparador ~ diferencia maiúsculas de minúsculas. Para não diferenciar maiúsculas de minúsculas, semelhante a ILIKE, use ~\* (o til seguido por um asterisco):

```
SELECT 'The data is about UFOs' ~* 'DATA' as comparison;

comparison
-----
true
```

Para negar o comparador, coloque um ! (ponto de exclamação) antes da combinação til ou til-asterisco:

```
SELECT 'The data is about UFOs' !~ 'alligators' as comparison;

comparison
-----
true
```

**Tabela 5-1** resume os quatro comparadores POSIX.

*Tabela 5-1. Comparadores POSIX*

Sintaxe	O que faz	Case sensitive?
~	Compara duas declarações e retorna TRUE se uma estiver contida na outra	Sim
~*	Compara duas declarações e retorna TRUE se uma estiver contida na outra	Não
!~	Compara duas declarações 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 regex em nosso SQL, vamos nos familiarizar com algumas das sintaxes especiais de correspondência de padrões que ela oferece. O primeiro caractere especial a ser conhecido é o símbolo . (ponto), um curinga que é usado para corresponder a qualquer caractere único:

```

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

```

Vamos detalhar isso para entender o que está acontecendo e desenvolver nossa intuição sobre como funciona a regex. Na primeira comparação, o padrão tenta corresponder a qualquer caractere, indicado pelo ponto, um espaço e, em seguida, a palavra “data”. Esse padrão corresponde à string “e data” na frase de exemplo, então TRUE é retornado. Se isso parecer contra-intuitivo, já que existem caracteres adicionais antes da letra “e” e depois da palavra “data”, lembre-se de que o comparador está apenas procurando por esse padrão em algum lugar da string, semelhante a um operador LIKE. Na segunda comparação, o padrão tenta corresponder a qualquer caractere seguido por “The”. Como na frase de exemplo “The” é o início da string e não há caracteres antes dela, o valor FALSE é retornado.

Para corresponder a vários caracteres, use o símbolo \* (asterisco). Isso corresponderá a zero ou mais caracteres, semelhante ao uso do símbolo % (porcentagem) em uma instrução LIKE. Esse uso do asterisco é diferente de colocá-lo imediatamente após o til (~\*), o que torna a correspondência insensível. Observe, no entanto, que neste caso “%” não é um curinga e é tratado como um caractere literal a ser correspondido:

```

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 a serem conhecidos são [ e ] (colchetes esquerdo e direito). Eles são usados para incluir um conjunto de caracteres, qualquer um dos quais deve corresponder. Os colchetes correspondem a um único caractere, embora vários caracteres possam estar entre eles, embora veremos em breve como fazer a correspondência mais de uma vez. Um uso para os colchetes é tornar parte de um padrão insensível a maiúsculas e minúsculas, colocando as letras maiúsculas e minúsculas entre colchetes (não use uma vírgula, pois isso corresponderia ao próprio caractere de vírgula):

```

SELECT 'The data is about UFOs' ~ '[Tt]he' as comparison;

comparison
-----
true

```

Neste exemplo, o padrão corresponderá a “the” ou “The”; como essa string inicia a frase de exemplo, a instrução retorna o valor TRUE. Isso não é exatamente a mesma coisa que a correspondência que não diferencia maiúsculas de minúsculas ~\*, porque nesse caso variações como “tHe” e “THE” não correspondem ao 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 correspondência entre colchetes é combinar um padrão que inclui um número, permitindo qualquer número. Por exemplo, imagine que queremos corresponder a qualquer descrição que mencione “7 minutos”, “8 minutos” ou “9 minutos”. Isso pode ser feito com uma instrução CASE com vários operadores LIKE, mas com regex a sintaxe do padrão é mais compacta:

```
SELECT 'sighting lasted 8 minutes' ~ '[789] minutes' as comparison;

comparison
-----
true
```

Para corresponder a qualquer número, podemos colocar todos os dígitos entre colchetes:

```
[0123456789]
```

No entanto, regex permite que um intervalo de caracteres seja inserido com um separador - (traço). Todos os números podem ser indicados por [0-9]. Qualquer intervalo menor de números também pode ser usado, como [0-3] ou [4-9]. Este padrão, com um intervalo, é equivalente ao último exemplo que listou cada número:

```
SELECT 'sighting lasted 8 minutes' ~ '[7-9] minutes' as comparison;

comparison
-----
true
```

Intervalos de letras podem ser combinados de maneira semelhante. A Tabela 5-2 resume os padrões de intervalo que são mais úteis na análise SQL. Valores não numéricos e não letras também podem ser colocados entre colchetes, como em [%@].

Tabela 5-2. Padrões de intervalo Regex

Padrão de intervalo	Objetivo
[0-9]	Corresponder a qualquer número
[az]	Corresponder a qualquer letra minúscula
[AZ]	Corresponder a qualquer letra maiúscula
[A-Za-z0-9]	Corresponder a qualquer letra minúscula ou maiúscula, ou qualquer número
[Az ]	Corresponde a qualquer caractere ASCII; geralmente não é usado porque corresponde a tudo, incluindo símbolos

Se a correspondência de padrão desejada contiver mais de uma instância de um determinado valor ou tipo de valor, uma opção é incluir quantos intervalos forem necessários, um após o outro. Por exemplo, podemos combinar um número de três dígitos repetindo a notação do intervalo de números três vezes:

```
SELECT 'driving on 495 south' ~ 'on [0-9][0-9][0-9]' as comparison;
comparison
-----
true
```

Outra opção é usar uma das sintaxes especiais opcionais para repetir um padrão várias vezes. Isso pode ser útil quando você não sabe exatamente quantas vezes o padrão se repetirá, mas tenha cuidado ao verificar os resultados para garantir que não retorne acidentalmente mais correspondências do que o pretendido. Para corresponder uma ou mais vezes, coloque o símbolo + (mais) após o 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 indicar o número de vezes para repetir um padrão.

Tabela 5-3. Padrões Regex para combinar um conjunto de caracteres várias vezes; em cada caso, o símbolo ou símbolos são colocados imediatamente após a expressão definida

Símbolo	Objetivo
+	Corresponder ao conjunto de caracteres uma ou mais vezes
*	Corresponder ao conjunto de caracteres zero ou mais vezes
?	Corresponde ao conjunto de caracteres zero ou uma vez
{ }	Corresponde ao conjunto de caracteres o número de vezes especificado entre as chaves; por exemplo, {3} corresponde exatamente três vezes
{, }	Corresponde ao conjunto de caracteres qualquer número de vezes em um intervalo especificado pelos números separados por vírgula entre chaves; por exemplo, {3,5} corresponde entre três e cinco vezes

Às vezes, em vez de corresponder a um padrão, queremos encontrar itens que *não* correspondem a um padrão. Isso pode ser feito colocando o símbolo ^ (caret) antes do padrão, que serve para negar o padrão:

```
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
```

Podemos querer corresponder a um padrão que inclui um dos caracteres especiais, portanto, precisamos de uma maneira de informar ao banco de dados para verificar esse caractere literal e não tratá-lo como especial. Para fazer isso, precisamos de um caractere de escape, que é o símbolo \ (barra invertida) na regex:

```
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, omitindo a barra invertida antes que o ponto de interrogação faça com que o banco de dados retorne um erro “expressão regular inválida” (o texto exato do erro pode ser diferente dependendo do tipo de banco de dados). Na segunda linha, mesmo que ^ seja seguido por um ou mais dígitos ([0-9]+), o banco de dados interpreta o ^ na comparação '^[0-9]+' como uma negação e avaliará se o string não inclui os dígitos especificados. A terceira linha escapa do acento circunflexo com uma barra invertida, e o banco de dados agora interpreta isso como o caractere literal ^.

Os dados de texto geralmente incluem caracteres de espaço em branco. Eles vão desde o espaço, que nossos olhos percebem, até a aba sutil e às vezes não impressa e os caracteres de nova linha. Veremos mais tarde como substituí-los por regex, mas por enquanto vamos nos ater a como combiná-los em um regex. As guias são combinadas com \t. As novas linhas são combinadas com \r para um retorno ou \n para um avanço de linha e, dependendo do sistema operacional, às vezes ambos são necessários: \r\n. Faça experiências com seu ambiente executando algumas consultas simples para ver o que retorna o resultado desejado. Para corresponder a qualquer caractere de espaço em branco, use \s, mas observe que isso também corresponde ao caractere de espaço:

```
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
```



Ferramentas de consulta SQL ou analisadores de consulta SQL podem ter problemas para interpretar novas linhas digitadas diretamente nelas e, portanto, podem retornar um erro. Se este for o caso, tente copiar e colar o texto da fonte em vez de digitá-lo. No entanto, todas as ferramentas de consulta SQL devem ser capazes de trabalhar com novas linhas que existem em uma tabela de banco de dados.

Semelhante às expressões matemáticas, os parênteses podem ser usados para incluir expressões que devem ser tratadas juntas. Por exemplo, podemos querer combinar um padrão um tanto complexo que se repete 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 regex, '([0-9]{2}[a-z])\{3\}', para correspondência. O padrão dentro dos parênteses, [0-9]{2}[a-z], procura por dois dígitos seguidos por uma letra minúscula.

Fora dos parênteses, {3} indica que todo o padrão deve ser repetido três vezes. A primeira linha segue esse padrão, pois contém a string 12a34b56c. A segunda linha não corresponde ao padrão; ele tem dois dígitos seguidos por uma letra minúscula (23a) e depois mais dois dígitos (23a45), mas essa segunda repetição é seguida por um terceiro dígito em vez de outra letra minúscula (23a456), portanto não há correspondência. A terceira linha tem um padrão correspondente, 99x66y33z.

Como acabamos de ver, regex pode ser usado em qualquer número de combinações com outras expressões, tanto regex quanto texto normal, para criar código de correspondência de padrões. Além de especificar *o* que corresponder, o regex pode ser usado para especificar *onde* corresponder. Use o caractere especial \y para corresponder a um padrão que começa no início ou no final de uma palavra (em alguns bancos de dados, pode ser \b). Como exemplo, imagine que estávamos interessados em encontrar a palavra “carro” nos relatórios de avistamento de OVNIs. Poderíamos escrever uma expressão como esta:

```
SELECT
'I was in my car going south toward my home' ~ 'car' as comparison;

comparison
-----
true
```

Localiza “car” na string e retorna TRUE conforme esperado. No entanto, vamos ver 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 combinam com o padrão “car” também, embora “scares”, “carpenter” e “boxcar” não sejam exatamente o que se pretendia quando procuramos menções a carros. Para corrigir isso, podemos adicionar \y ao início e ao final 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, neste exemplo simples, poderíamos simplesmente adicionar espaços antes e depois da palavra “car” com o mesmo resultado. A vantagem do padrão é que ele também pegará casos em que o padrão está no início de uma string e, portanto, não possui um espaço à esquerda:

```

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 correspondência que não diferencia maiúsculas de minúsculas quando "Car" é a primeira palavra, mas o padrão não 'car'. Para combinar o início de uma string inteira, use o caractere especial \A, e para combinar o final de uma 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 corresponde a "Car" no início da string. A segunda linha começa com "I", para que o padrão não corresponda. Na terceira linha, o padrão está procurando por "carro" no final da string e corresponde a ele. Finalmente, na quarta linha, a última palavra é "casa", então o padrão não corresponde.

Se esta é a primeira vez que você trabalha com expressões regulares, pode levar algumas leituras e alguns experimentos em seu editor SQL para pegar o jeito.

Não há nada como trabalhar com exemplos reais para ajudar a solidificar o aprendizado, então, a seguir, passarei por algumas aplicações para nossa análise de avistamentos de OVNIs e também apresentarei algumas funções específicas de regex SQL.



As implementações de expressões regulares variam muito de acordo com o fornecedor do banco de dados. Os operadores POSIX nesta seção funcionam no Postgres e em bancos de dados derivados do Postgres, como Amazon Redshift, mas não necessariamente em outros.

Uma alternativa ao operador ~ é a função `rlike` ou `regexp_like` (dependendo do banco de dados). Eles têm o seguinte formato:

```
regexp_like(string, pattern, optional_parameters)
```

O primeiro exemplo nesta seção seria escrito como:

```
SELECT regexp_like('The data is about UFOs','data')
    as comparison;
```

Os parâmetros opcionais controlam o tipo de correspondência, como se a correspondência não diferencia maiúsculas de minúsculas.

Muitos desses bancos de dados têm funções adicionais não abordadas aqui, como `regexp_substr` para localizar substrings correspondentes e `regexp_count` para encontrar o número de vezes que um padrão é correspondido. O Postgres suporta POSIX mas infelizmente não suporta essas outras funções. As organizações que esperam fazer muita análise de texto farão bem em escolher um tipo de banco de dados com um conjunto robusto de funções de expressão regular.

## Localizando e substituindo com regex

Na seção anterior, discutimos expressões regulares e como construir padrões com regex para corresponder a partes de strings em nossos conjuntos de dados. Vamos aplicar esta técnica ao conjunto de dados de avistamentos de OVNIs para ver como funciona na prática. Ao longo do caminho, também apresentarei algumas funções regex SQL adicionais.

Os relatórios de avistamento contêm uma variedade de detalhes, como o que o repórter estava fazendo no momento do avistamento e quando e onde o estava fazendo. Outro detalhe comumente mencionado é ver um certo número de luzes. Como primeiro exemplo, vamos encontrar as descrições que contêm um número e a palavra “light” ou “lights”. Por uma questão de exibição neste livro, vou apenas verificar os primeiros 100 caracteres, mas este código também pode funcionar em todo o campo de descrição:

```
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 de expressão regular corresponde a qualquer número de dígitos ([0-9]+), seguido por um espaço, depois a string “light” e, finalmente, uma letra “s”, um espaço, uma vírgula ou um ponto. Além de encontrar os registros relevantes, podemos querer dividir apenas a parte que se refere ao número e a palavra “luzes”. Para fazer isso, usaremos a função regex `regexp_matches`.



O suporte à função Regex varia muito de acordo com o fornecedor do banco de dados e, às vezes, com a versão do software do banco de dados. O SQL Server não suporta as funções, enquanto o MySQL tem suporte mínimo para elas. Bancos de dados analíticos como Redshift, Snowflake e Vertica oferecem suporte a uma variedade de funções úteis. O Postgres tem apenas funções match e replace. Explore a documentação do seu banco de dados para disponibilidade de funções específicas.

A função `regexp_matches` recebe dois argumentos: uma string para pesquisar e um padrão de correspondência de regex. Ele retorna uma matriz da(s) string(s) que correspondeu ao padrão. Se não houver correspondências, um valor nulo será retornado. Como o valor de retorno é um array, usaremos um índice de [1] para retornar apenas um único valor como VARCHAR, o que permitirá a manipulação de string adicional conforme necessário. Se você estiver trabalhando em outro tipo de banco de dados, a função `regexp_substr` é semelhante a `regexp_matches`, mas retorna um valor VARCHAR, portanto 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 são colocados entre {} (chaves), e essa é uma boa maneira de identificar que algo no banco de dados não é um dos tipos de dados regulares com os quais trabalhamos até agora. As matrizes têm algumas vantagens ao armazenar e recuperar dados, mas não são tão fáceis de trabalhar em SQL, pois exigem uma sintaxe especial. Os elementos em uma matriz são acessados usando a notação [ ] (colchetes). Para nossos propósitos aqui, basta saber que o primeiro elemento é encontrado com [1], o segundo com [2], e assim por diante.

Com base em nosso exemplo, podemos analisar o valor desejado, o número e a palavra “light(s)” do campo de descrição e, em seguida, GROUP BY esse valor e as variações mais comuns:

```
SELECT (regexp_matches(description,'[0-9]+ light[s ,.]'))[1]
, count(*)
FROM ufo
WHERE description ~ '[0-9]+ light[s ,.]'
```

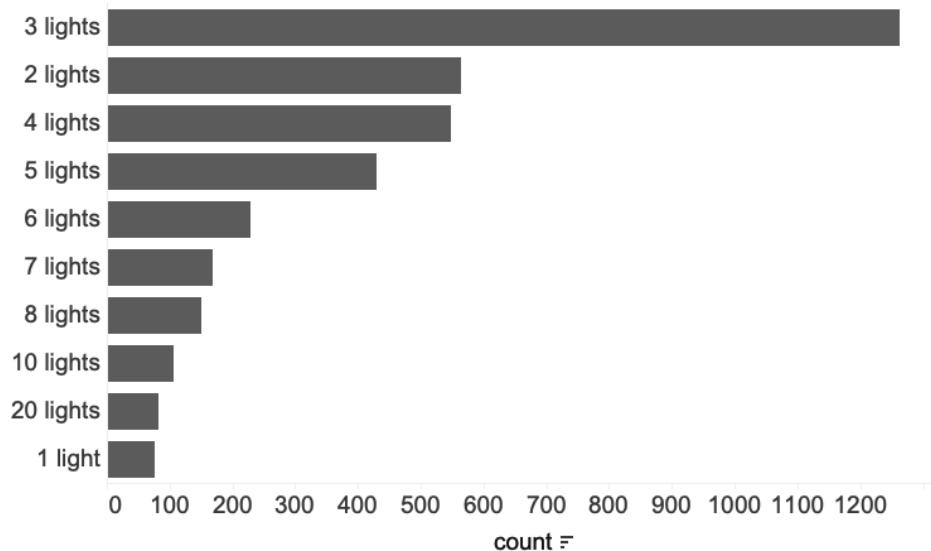
```

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 graficamente na [Figura 5-9](#).



*Figura 5-9. Número de luzes mencionadas no início das descrições de avistamentos de OVNIs*

Relatos que mencionam três luzes são mais que duas vezes mais comuns que o segundo número de luzes mais frequentemente mencionado, e de duas a seis luzes são mais comumente vistas. Para encontrar o intervalo completo do número de luzes, podemos analisar o texto correspondente e, em seguida, 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 relatório menciona duas mil luzes, e o valor mínimo de zero luzes também é mencionado. Podemos querer revisar esses relatórios mais a fundo para ver se há algo mais interessante ou incomum sobre esses valores extremos.

Além de encontrar correspondências, talvez queiramos substituir o texto correspondente por algum texto alternativo. Isso é particularmente útil ao tentar limpar um conjunto de dados de texto que tenha várias grafias para a mesma coisa subjacente. A função `regexp_replace` pode fazer isso. É semelhante à `replace` discutida anteriormente neste capítulo, mas pode receber um argumento de expressão regular como padrão para corresponder. A sintaxe é semelhante à função `replace`:

```
regexp_replace(field or string, pattern, replacement value)
```

Vamos colocar isso em prática para tentar limpar o campo `duration` que analisamos da coluna anterior `sighting_report`. 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 desta amostra, as durações de “10 minutos,” “10 min” e “10 min” representam a mesma quantidade de tempo, mas o banco de dados não sabe combiná-los porque as grafias são ligeiramente diferentes. Poderíamos usar uma série de funções aninhadas `replace` para converter todas essas grafias diferentes. No entanto, também teríamos que levar em conta outras variações, como as capitalizações. Regex é útil nesta situação, permitindo-nos criar um código mais compacto. O primeiro passo é desenvolver um padrão que corresponda à string desejada, o que podemos fazer com a função `regexp_matches`.

É uma boa ideia revisar esta etapa intermediária para verificar se estamos correspondendo ao texto correto:

```

SELECT duration
,(regexp_matches(duration
,'[m|M][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
...	...

Vamos quebrar isso. Na subconsulta, o valor `duration` é separada do campo `sighting_report`. Em seguida, a função `regexp_matches` procura por strings que correspondam ao padrão:

```
'[m|M][Ii][Nn][A-Za-z]*y'
```

Este padrão começa no início de uma palavra (`\m`) e depois procura qualquer sequência das letras “m”, “i” e “n”, independentemente da capitalização ([`Mm`] e assim por diante). Em seguida, ele procura zero ou mais instâncias de qualquer outra letra minúscula ou maiúscula (`[A-Za-z]*`) e, finalmente, verifica o final de uma palavra (`\y`) para que apenas a palavra que inclui o a variação de “minutos” está incluída e não o resto da string. Observe que os caracteres “+” e “?” não são correspondentes. Com esse padrão, agora podemos substituir todas essas variações pelo valor padrão “min”:

```

SELECT duration
,(regexp_matches(duration
,'[m|M][Ii][Nn][A-Za-z]*y')
 )[1] as matched_minutes
,regexp_replace(duration
,'[m|M][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 na coluna `replace_text` estão muito mais padronizados agora. Os caracteres de ponto, mais e ponto de interrogação também podem ser substituídos aprimorando o regex. Do ponto de vista analítico, no entanto, podemos querer considerar como representar a incerteza que o sinal de mais e o ponto de interrogação representam. As funções `regexp_replace` podem ser aninhadas para conseguir a substituição de diferentes partes ou tipos de strings. Por exemplo, podemos padronizar os minutos e 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 para horas é semelhante à de minutos, procurando correspondências de “hour” sem distinção entre maiúsculas e minúsculas no início de uma palavra, seguidas por zero ou mais caracteres de outras letras antes do final da palavra. As correspondências intermediárias de hora e minutos podem não ser necessárias no resultado final, mas acho que são úteis para revisar enquanto estou desenvolvendo meu código SQL para evitar erros mais tarde. Uma limpeza completa da coluna `duration` provavelmente envolveria muito mais linhas de código, e é muito fácil perder o controle e introduzir um erro de digitação.

A função `regexp_replace` pode ser aninhada quantas vezes quiser ou pode ser combinada com a função básica `replace`. Outro uso para `regexp_replace` está nas instruções CASE, para substituição direcionada quando as condições na instrução forem atendidas. Regex é uma ferramenta poderosa e flexível dentro do SQL que, como vimos, pode ser usada de várias maneiras em uma consulta SQL geral.

Nesta seção, apresentei várias maneiras de pesquisar, localizar e substituir elementos específicos em textos mais longos, desde correspondências curinga com listas LIKE a IN e correspondências de padrões mais complexas com regex. Tudo isso, junto com as funções de análise e transformação de texto apresentadas anteriormente, nos permitem criar conjuntos de regras personalizados com a complexidade necessária para lidar com os conjuntos de dados em mãos. No entanto, vale a pena manter em mente o equilíbrio entre complexidade e carga de manutenção. Para uma análise única de um conjunto de dados, pode valer a pena criar conjuntos de regras complexos que limpem perfeitamente os dados. Para relatórios e monitoramento contínuos, geralmente vale a pena explorar opções para receber dados mais limpos de fontes de dados. A seguir, veremos várias maneiras de construir novas strings de texto com SQL: usando constantes, strings existentes e strings analisadas.

## Construindo e Remodelando Texto

Vimos como analisar, transformar, localizar e substituir elementos de strings para realizar uma variedade de tarefas de limpeza e análise com SQL. Além disso, o SQL pode ser usado para gerar novas combinações de texto. Nesta seção, discutirei primeiro *concatenação*, que permite que diferentes campos e tipos de dados sejam consolidados em um único campo. Em seguida, discutirei a alteração da forma do texto com funções que combinam várias colunas em uma única linha, bem como o oposto: dividir uma única string em várias linhas.

### Concatenação

Novo texto pode ser criado com SQL com concatenação. Qualquer combinação de texto constante ou codificado, campos de banco de dados e cálculos nesses campos podem ser unidos.

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** (concatenate with separator), que recebe um valor de separador como o primeiro argumento, seguido pela lista de valores a serem concatenados. Isso é útil quando há vários valores que você deseja juntar, usando uma vírgula, um traço ou um elemento semelhante para separá-los:

```
concat_ws(separator, value1, value2...)
```

Finalmente, **||** (duplo pipe) pode ser usado em muitos bancos de dados para concatenar strings (o SQL Server usa **+** em vez disso):

```
value1 || value2
```



Se algum dos valores em 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, imagine que queremos rotular as formas como tal e adicionar a palavra “relatórios” à contagem de relatórios para cada forma. A subconsulta analisa o nome da forma do campo **sighting\_report** e **counts** o número de registros. A consulta externa concatena as formas com a string '**(shape)**' e os **reports** 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, podemos unir os valores de forma e localização em um único 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 graficamente na Figura 5-10.

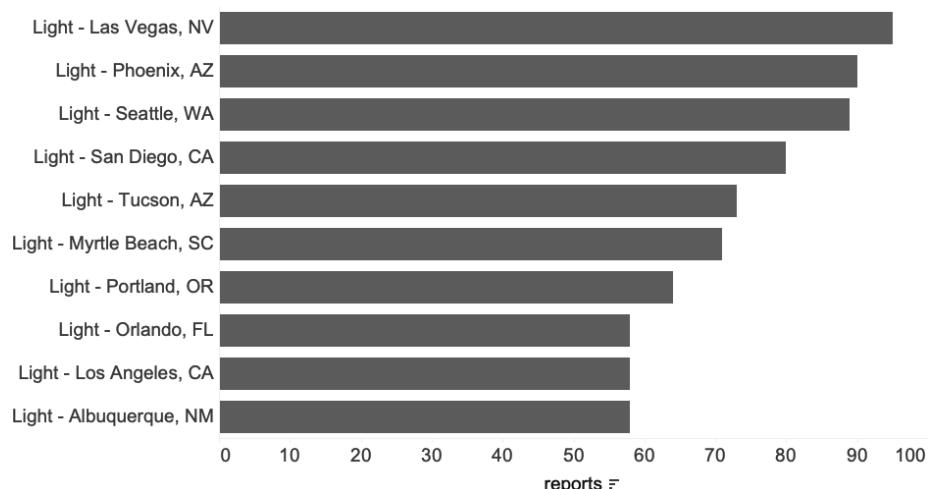


Figura 5-10. Principais combinações de forma e localização em avistamentos de OVNIs

Vimos anteriormente que “luz” é a forma mais comum, então não é surpreendente que ela apareça em cada um dos principais resultados. Phoenix é o local mais comum, enquanto Las Vegas é o segundo mais comum em geral.

Nesse caso, como tivemos tanto trabalho para analisar os diferentes campos, pode não fazer muito sentido concatená-los novamente. No entanto, pode ser útil reorganizar o texto ou combinar valores em um único campo para exibição em outra ferramenta. Ao combinar vários campos e texto, também podemos gerar frases que podem funcionar como resumos dos dados, para uso em e-mails ou relatórios automatizados. Neste exemplo, a subconsulta `a` analisa os campos `occurred` e `shape`, como vimos anteriormente, e `counts` os registros. Então na subconsulta `aa`, o `min` e `max` do `occurred` são calculados, juntamente com o número total de `reports`, e os resultados são *GROUPed BY shape*. Linhas com campos `occurred` menos de oito caracteres são excluídos, para remover aqueles que não possuem datas devidamente formadas e evitar erros nos cálculos `min` e `max`. Por fim, na consulta externa, o texto final é montado com a função `concat`. O formato das datas é alterado para datas longas (9 de abril de 1957) para as datas mais antigas e mais recentes:

```

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 ao formatar o número de relatórios ou adicionar `coalesce` ou `CASE` para lidar com nomes de formas em branco, por exemplo. Embora essas frases sejam repetitivas e, portanto, não sejam páreo para escritores humanos (ou IA), elas serão dinâmicas se a fonte de dados for atualizada com frequência e, portanto, podem ser úteis em aplicativos de relatórios.

Junto com funções e operadores para criar novo texto com concatenação, o SQL tem algumas funções especiais para remodelar o texto, que veremos a seguir.

## Reformulando o texto

Como vimos no [Capítulo 2](#), alterar a forma dos dados — seja girando de linhas para colunas ou vice-versa, alterando os dados de colunas para linhas — às vezes é útil. Vimos como fazer isso com `GROUP BY` e agregações, ou com declarações `UNION`. No SQL existem algumas funções especiais para remodelar o texto, no entanto.

Um caso de uso para remodelar o texto é quando há várias linhas com diferentes valores de texto para uma entidade e gostaríamos de combiná-las em um único valor. A combinação de valores pode torná-los mais difíceis de analisar, é claro, mas às vezes o caso de uso requer um único registro por entidade na saída. Combinar os valores individuais em um único campo nos 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 agrupa apenas valores que não são nulos, e a ordem pode ser controlada com uma cláusula `ORDER BY` dentro da função conforme 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
...
...

```

Como `string_agg` é uma função agregada, ela requer uma `GROUP BY` no outros campos na consulta. No MySQL, uma função equivalente é `group_concat`, e bancos de dados analíticos como Redshift e Snowflake têm uma função semelhante chamada `listagg`.

Outro caso de uso é fazer exatamente o oposto de `string_agg` e, em vez disso, dividir um único campo em várias linhas. Há muita inconsistência em como isso é implementado em diferentes bancos de dados e até mesmo se existe uma função para isso. Postgres tem uma função chamada `regexp_split_to_table`, enquanto alguns outros bancos de dados têm uma função `split_to_table` que opera de forma semelhante (verifique a documentação para disponibilidade e 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 uma vírgula ou um caractere de espaço. A função então divide os valores em linhas:

```

SELECT
regexp_split_to_table('Red, Orange, Yellow, Green, Blue, Purple'
    , ', ');
-----+
Red
Orange
Yellow

```

Green  
Blue  
Purple

A string a ser dividida pode incluir qualquer coisa e não precisa necessariamente ser uma lista. Podemos usar a função para dividir qualquer string, incluindo frases. Podemos então usar isso para encontrar as palavras mais comuns usadas em campos de texto, uma ferramenta potencialmente útil para o trabalho de análise de texto. Vamos dar uma olhada nas palavras mais comuns usadas nas descrições dos relatórios de avistamento 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
```

A subconsulta primeiro transforma a `description` em minúscula, pois variações de maiúsculas não são interessantes para este exemplo. Em seguida, a string é dividida usando o regex '`\s+`', que divide em qualquer um ou mais caracteres de espaço em branco.

As palavras mais usadas não são surpreendentes; no entanto, eles não são particularmente úteis, pois são apenas palavras comumente usadas em geral. Para encontrar uma lista mais significativa, podemos remover as chamadas *palavras de parada*. Estas são simplesmente as palavras mais usadas em um idioma. Alguns bancos de dados têm listas embutidas nos chamados dicionários, mas as implementações não são padrão. Também não há uma lista correta de palavras de parada acordada e é comum ajustar a lista específica para a aplicação desejada; no entanto, existem várias listas de palavras de parada comuns na internet. Para este exemplo, carreguei uma lista de 421 palavras comuns em uma tabela chamada `stop_words`, disponível no [site GitHub](#). As palavras de parada são removidas do conjunto de resultados com um *LEFT JOIN* para a tabela `stop_words`, filtradas para resultados que não estão 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
```

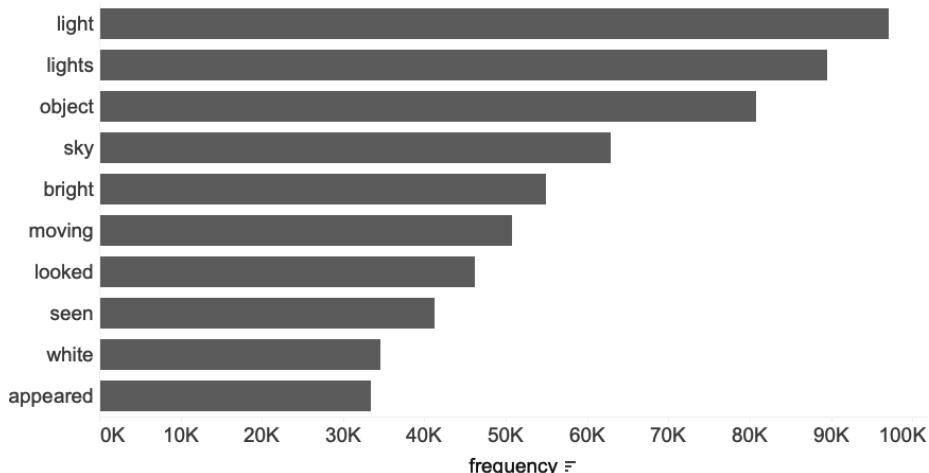
```

GROUP BY 1
ORDER BY 2 desc
;

word      frequency
-----
light     97071
lights    89537
object    80785
...
...

```

As 10 palavras mais comuns estão representadas graficamente na [Figura 5-11](#).



*Figura 5-11. Palavras mais comuns em descrições de avistamentos de OVNIs, excluindo palavras de parada*

Poderíamos continuar a ficar mais sofisticados adicionando palavras comuns adicionais à tabela `stop_words` ou por *JOINing* os resultados com as descrições para marcá-las com as palavras interessantes que elas contêm. Observe que `regexp_split_to_table` e funções semelhantes em outros bancos de dados podem ser lentas, dependendo do tamanho e do número de registros analisados.

Construir e remodelar texto com SQL pode ser feito de maneira simples ou complexa, conforme necessário. As funções de concatenação, agregação de strings e divisão de strings podem ser usadas sozinhas, em combinação entre si e com outras funções e operadores SQL para obter a saída de dados desejada.

## Conclusão

Embora o SQL nem sempre seja a primeira ferramenta mencionada quando se trata de análise de texto, ele possui muitas funções e operadores poderosos para realizar uma variedade de tarefas. Desde análise e transformações, até localizar e substituir, construir e remodelar texto, o SQL pode ser usado para limpar e preparar dados de texto, bem como realizar análises.

No próximo capítulo, vamos nos voltar para o uso do SQL para detecção de anomalias, outro tópico no qual o SQL nem sempre é a primeira ferramenta mencionada, mas para o qual possui recursos surpreendentes.

## CAPÍTULO 6

# Detecção de Anomalias

Uma *anomalia* é algo diferente de outros membros do mesmo grupo. Nos dados, uma anomalia é um registro, uma observação ou um valor que difere dos pontos de dados restantes de uma forma que levanta preocupações ou suspeitas. As anomalias têm vários nomes diferentes, incluindo *outliers*, *novidades*, *ruidos*, *desvios* e *exceções*, para citar alguns. Usarei os termos *anomalia* e *outlier* forma intercambiável ao longo deste capítulo, e você também poderá ver os outros termos usados nas discussões deste tópico. A detecção de anomalias pode ser o objetivo final de uma análise ou uma etapa de um projeto de análise mais amplo.

As anomalias normalmente têm uma de duas fontes: eventos reais que são extremos ou incomuns, ou erros introduzidos durante a coleta ou processamento de dados. Embora muitas das etapas usadas para detectar discrepâncias sejam as mesmas, independentemente da origem, a forma como escolhemos lidar com uma anomalia específica depende da causa raiz. Como resultado, entender a causa raiz e distinguir entre os dois tipos de causas é importante para o processo de análise.

Eventos reais podem gerar discrepâncias por vários motivos. Dados anômalos podem sinalizar fraude, invasão de rede, defeitos estruturais em um produto, brechas nas políticas ou uso do produto que não foi planejado ou previsto pelos desenvolvedores. A detecção de anomalias é amplamente utilizada para erradicar fraudes financeiras, e a segurança cibernética também faz uso desse tipo de análise. Às vezes, os dados anômalos ocorrem não porque um agente mal-intencionado está tentando explorar um sistema, mas porque um cliente está usando um produto de maneira inesperada. Por exemplo, eu conhecia alguém que usava um aplicativo de rastreamento de condicionamento físico, destinado a corrida, ciclismo, caminhada e atividades semelhantes, para registrar dados de suas saídas na pista de corrida de automóveis. Ele não havia encontrado uma opção melhor e não estava pensando em quão anômalos os valores de velocidade e distância de um carro em uma pista são comparados aos registrados para passeios de bicicleta ou corrida. Quando as anomalias podem ser rastreadas a um processo real, decidir o que fazer com elas requer um bom entendimento da análise a ser

feita, bem como conhecimento de domínio, termos de uso e, às vezes, o sistema legal que rege o produto.

Os dados também podem conter anomalias devido a erros na coleta ou processamento. Os dados inseridos manualmente são notórios por erros de digitação e dados incorretos. Alterações em formulários, campos ou regras de validação podem introduzir valores inesperados, incluindo nulos. O rastreamento de comportamento de aplicativos da web e móveis é comum; no entanto, qualquer alteração em como e quando esse registro é feito pode introduzir anomalias. Passei horas suficientes diagnosticando alterações nas métricas e aprendi a perguntar antecipadamente se algum registro foi alterado recentemente. O processamento de dados pode introduzir valores discrepantes quando alguns valores são filtrados erroneamente, as etapas de processamento não são concluídas ou os dados são carregados várias vezes, criando duplicatas. Quando as anomalias resultam do processamento de dados, geralmente podemos estar mais confiantes em corrigir ou descartar esses valores. Obviamente, corrigir a entrada ou o processamento de dados upstream é sempre uma boa ideia, se possível, para evitar futuros problemas de qualidade.

Neste capítulo, discutirei primeiro alguns dos motivos para usar o SQL para esse tipo de análise e os locais em que ele falha. Em seguida, apresentarei o conjunto de dados de terremotos que será usado nos exemplos do restante do capítulo. Depois disso, apresentarei as ferramentas básicas que temos à nossa disposição no SQL para detectar outliers. Em seguida, discutirei as várias formas de valores discrepantes que podemos aplicar as ferramentas para encontrar. Uma vez que detectamos e entendemos as anomalias, o próximo passo é decidir o que fazer com elas. As anomalias nem sempre precisam ser problemáticas, pois são na detecção de fraudes, detecção de ataques cibernéticos e monitoramento do sistema de saúde. As técnicas deste capítulo também podem ser usadas para detectar clientes ou campanhas de marketing excepcionalmente bons ou mudanças positivas no comportamento do cliente. Às vezes, o objetivo da detecção de anomalias é passar as anomalias para outros humanos ou máquinas para lidar com elas, mas geralmente esse é um passo em uma análise mais ampla, então vou encerrar com várias opções para corrigir anomalias.

## Capacidades e limites do SQL para detecção de anomalias

SQL é uma linguagem versátil e poderosa para muitas tarefas de análise de dados, embora não possa fazer tudo. Ao realizar a detecção de anomalias, o SQL tem vários pontos fortes, bem como algumas desvantagens que tornam outras linguagens ou ferramentas melhores escolhas para algumas tarefas.

Vale a pena considerar o SQL quando o conjunto de dados já está em um banco de dados, como vimos anteriormente com séries temporais e análise de texto nos Capítulos 3 e 5, respectivamente. O SQL aproveita o poder computacional do banco de dados para realizar cálculos em muitos registros rapidamente. Particularmente com grandes tabelas de dados, a transferência de um banco de dados para outra ferramenta é demorada. Trabalhar em um banco de dados faz ainda mais sentido quando a detecção de anomalias é uma etapa de uma análise maior que será feita em SQL.

Código escrito em SQL pode ser examinado para entender por que determinados registros foram sinalizados como discrepantes, e o SQL permanecerá consistente ao longo do tempo, mesmo que os dados que fluam para um banco de dados mudem.

Do lado negativo, o SQL 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 cálculos estatísticos adicionais mais complexos podem ser muito lentos ou intensos para alguns bancos de dados. Para casos de uso que exigem resposta muito rápida, como detecção de fraude ou intrusão, a análise de dados em um banco de dados pode simplesmente não ser apropriada, pois geralmente há atraso no carregamento de dados, principalmente em bancos de dados analíticos. Um fluxo de trabalho comum é usar o SQL para fazer a análise inicial e determinar os valores mínimos, máximos e médios típicos e, em seguida, desenvolver mais monitoramento em tempo real usando um serviço de streaming ou armazenamentos de dados especiais em tempo real. No entanto, detectar tipos de padrões discrepantes e implementá-los em serviços de streaming ou armazenamentos de dados especiais em tempo real pode ser uma opção. Finalmente, o código SQL é baseado em regras, como vimos no Capítulo 5. É muito bom para lidar com um conjunto conhecido de condições ou critérios, mas o SQL não se ajustará automaticamente aos tipos de padrões de mudança vistos com adversários que mudam rapidamente. As abordagens de aprendizado de máquina e as linguagens associadas a elas geralmente são a melhor escolha para esses aplicativos.

Agora que discutimos as vantagens do SQL e quando usá-lo em vez de outra linguagem ou ferramenta, vamos dar uma olhada nos dados que usaremos como exemplos neste capítulo antes de passar para o código em si.

## O conjunto de dados

Os dados para os exemplos neste capítulo são um conjunto de registros para todos os terremotos registrados pelo US Geological Survey (USGS) de 2010 a 2020. O USGS fornece os dados em vários formatos, incluindo 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 evento de terremoto e inclui informações como carimbo de data/hora, localização, magnitude, profundidade e fonte das informações. Uma amostra dos dados é mostrada na Figura 6-1. Um dicionário de dados (<https://earthquake.usgs.gov/data/comcat/data-eventterms.php>) está disponível no site USGS.

*	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 `earthquakes`

Os terremotos são causados por deslizamentos súbitos ao longo de falhas nas placas tectônicas que existem na superfície externa da Terra. Locais nas bordas dessas placas experimentam muito mais terremotos e mais dramáticos do que outros lugares. O chamado Anel de Fogo é uma região ao longo da borda do Oceano Pacífico em que ocorrem muitos terremotos. Vários locais nessa região, incluindo Califórnia, Alasca, Japão e Indonésia, aparecerão com frequência em nossa análise.

*Magnitude* é uma medida do tamanho de um terremoto em sua origem, medido por suas ondas sísmicas. A magnitude é registrada em uma escala logarítmica, o que significa que a amplitude de um terremoto de magnitude 5 é 10 vezes maior que a de um terremoto de magnitude 4. A medição real de terremotos é fascinante, mas está além do escopo deste livro. O site do [USGS](#) é um bom lugar para começar se você quiser aprender mais.

## Detectando Outliers

Embora a ideia de uma anomalia ou outlier — um ponto de dados muito diferente do resto — pareça simples, na verdade, encontrar um em qualquer conjunto de dados específico apresenta alguns desafios. O primeiro desafio tem a ver com saber quando um valor ou ponto de dados é comum ou raro, e o segundo é definir um limite para marcar valores em ambos os lados dessa linha divisória. À medida que analisamos os dados de `earthquakes`, traçaremos o perfil das profundidades e magnitudes para desenvolver uma compreensão de quais valores são normais e quais são incomuns.

Geralmente, quanto maior ou mais completo o conjunto de dados, mais fácil é fazer um julgamento sobre o que é realmente anômalo. Em alguns casos, rotulamos valores ou “verdade básica” aos quais podemos nos referir. Um rótulo geralmente é uma coluna no conjunto de dados que indica se o registro é normal ou um outlier.

A verdade do terreno pode ser obtida da indústria ou de fontes científicas ou de análises anteriores e pode nos dizer, por exemplo, que qualquer terremoto maior que a magnitude 7 é uma anomalia. Em outros casos, devemos olhar para os próprios dados e aplicar um julgamento razoável. Para o restante do capítulo, vamos supor que temos um conjunto de dados grande o suficiente para fazer exatamente isso, embora, é claro, existam referências externas que poderíamos consultar sobre magnitudes típicas e extremas de terremotos.

Nossas ferramentas para detectar discrepâncias usando o próprio conjunto de dados se enquadram em algumas categorias. Primeiro, podemos classificar ou *ORDER BY* os valores nos dados. Isso pode ser combinado opcionalmente com várias cláusulas *GROUP BY* para encontrar valores discrepantes por frequência. Segundo, podemos usar as funções estatísticas do SQL para encontrar valores extremos em cada extremidade de um intervalo de valores. Finalmente, podemos representar graficamente os dados e inspecioná-los visualmente.

## Ordenando para Encontrar Anomalias

Uma das ferramentas básicas que temos para encontrar outliers é a ordenação dos dados, realizada com a cláusula *ORDER BY*. O comportamento padrão de *ORDER BY* é classificar em ordem crescente (*ASC*). Para classificar em ordem decrescente, adicione *DESC* após a coluna. Uma cláusula *ORDER BY* pode incluir uma ou mais colunas, e cada coluna pode ser classificada em ordem crescente ou decrescente, independentemente das outras. A classificação começa com a primeira coluna especificada. Se uma segunda coluna for especificada, os resultados da primeira classificação serão classificados pela segunda coluna (retendo a primeira classificação) e assim por diante por todas as colunas da cláusula.



Como a ordenação ocorre depois que o banco de dados calculou o restante da consulta, muitos bancos de dados permitem que você faça referência às colunas de consulta por número em vez de nome. O SQL Server é uma exceção; requer o nome completo. Prefiro a sintaxe de numeração porque resulta em um código mais compacto, principalmente quando as colunas de consulta incluem cálculos longos ou sintaxe de função.

Por exemplo, podemos classificar a tabela `earthquakes` por `mag`, a magnitude:

```
SELECT mag
FROM earthquakes
ORDER BY 1 desc
;
```

```
mag
-----
(null)
(null)
(null))
...
...
```

Isso retorna um número de linhas de nulos. Vamos observar que o conjunto de dados pode conter valores nulos para magnitude — um possível outlier em si. 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 que 9, e há apenas dois valores adicionais maiores que 8,5. Em muitos contextos, estes não parecem ser valores particularmente grandes. No entanto, com um pouco de conhecimento de domínio sobre terremotos, podemos reconhecer que esses valores são de fato muito grandes e incomuns. O USGS fornece uma lista dos [20 maiores terremotos do mundo](#) (<https://www.usgs.gov/programs/earthquake-hazards/science/20-largest-earthquakes-world>). Todos eles são de magnitude 8,4 ou maior, enquanto apenas cinco são de magnitude 9,0 ou maior, e três ocorreram entre 2010 e 2020, período coberto pelo nosso conjunto de dados.

Outra maneira de considerar se os valores são anomalias dentro de um conjunto de dados é calcular sua frequência. Podemos `count` o campo `id` e `GROUP BY` o `mag` para encontrar o número de terremotos por magnitude. O número de terremotos por magnitude é então dividido pelo número total de terremotos, que pode ser encontrado usando uma função `sum`. Todas as funções de janela requerem uma cláusula `OVER` com uma cláusula `PARTITION BY` e/ou `ORDER BY`. Como o denominador deve contar todos os registros, adicionei uma `PARTITION BY 1`, que é uma maneira de forçar o banco de dados a torná-lo uma função de janela, mas ainda ler a tabela inteira. Finalmente, o conjunto de resultados é `ORDERED BY` a 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
...   ...

```

Há apenas um terremoto de magnitude superior a 8,5, mas há dois que registraram 8,3. Pelo valor 6,9, há dois dígitos de terremotos, mas esses ainda representam uma porcentagem muito pequena dos dados. Em nossa investigação, também devemos verificar a outra extremidade da classificação, os menores valores, classificando em ordem crescente em vez de decrescente:

```

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 inferior dos valores, -9,99 e -9 ocorrem com mais frequência do que poderíamos esperar. Embora não possamos obter o logaritmo de zero ou um número negativo, um logaritmo pode ser negativo quando o argumento é maior que zero e menor que um. Por exemplo,  $\log(0,5)$  é igual a aproximadamente -0,301. Os valores -9,99 e -9 representam magnitudes de terremotos extremamente pequenas, e podemos questionar se esses pequenos terremotos podem realmente ser detectados. Dada a frequência desses valores, suspeito que eles representem um valor desconhecido em vez de um terremoto verdadeiramente minúsculo e, portanto, podemos considerá-los anomalias.

Além de classificar os dados gerais, pode ser útil *GROUP BY* um ou mais campos de atributo para localizar anomalias em subconjuntos dos dados. Por exemplo, podemos querer verificar as magnitudes mais altas e mais baixas registradas para geografias 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
```

“Norte da Califórnia” é o mais comum `place` no conjunto de dados e, inspecionando apenas o subconjunto, podemos ver que os valores alto e baixo não são próximos tão extremos quanto aqueles para o conjunto de dados como um todo. Terremotos de magnitude superior a 5,0 não são incomuns em geral, mas são discrepantes para o “Northern California”.

## Calcular percentis e desvios padrão para localizar anomalias

Classificar e, opcionalmente, agrupar dados e, em seguida, revisar os resultados visualmente é uma abordagem útil para detectar anomalias, principalmente quando os dados têm valores muito extremos. Sem conhecimento de domínio, no entanto, pode não ser óbvio que um terremoto de magnitude 9,0 seja uma anomalia. Quantificar a extremaidade dos pontos de dados adiciona outra camada de rigor à análise. Existem duas maneiras de fazer isso: com percentis ou com desvios padrão.

Os percentis representam a proporção de valores em uma distribuição que são menores 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. A mediana é tão comumente usada que possui sua própria função SQL, `median`, em muitos, mas não em todos os bancos de dados. Outros percentis também podem ser calculados. Por exemplo, podemos encontrar o percentil 25, onde 25% dos valores são mais baixos e 75% são mais altos, ou o percentil 89, onde 89% dos valores são mais baixos e 11% são mais altos. Os percentis são frequentemente encontrados em contextos acadêmicos, como testes padronizados, mas podem ser aplicados a qualquer domínio.

SQL tem uma função, `percent_rank`, que retorna o percentil para cada linha dentro de uma partição. Como em todas as funções de janela, a direção de classificação é controlada com uma instrução `ORDER BY`. Semelhante à função `rank`, `percent_rank` não recebe nenhum argumento; ele 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 ordenação é sempre uma boa idéia. Para encontrar o percentil das magnitudes de cada terremoto para cada local, podemos primeiro calcular o `percent_rank` para cada linha na subconsulta e, em seguida, contar as ocorrências de cada magnitude na consulta externa.

Observe que é importante calcular o primeiro `percent_rank`, antes de fazer qualquer agregação, para que os 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 um percentil de 1, ou 100%, indicando que todos os outros valores são menores que este. O terremoto de magnitude -1,6 tem um percentil de 0, indicando que nenhum outro ponto de dados é menor.

Além de encontrar o percentil exato de cada linha, o SQL pode dividir 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, podemos querer esculpir os dados configurados 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
```

Observando os resultados para “Alasca Central”, vemos que os três terremotos maiores que 5 estão no percentil 100, 1,5 se enquadra no 79º percentil, e os menores valores de –0,5 se enquadram no primeiro percentil. Depois de calcular esses valores, podemos encontrar os limites de cada ntile, usando `max` e `min`. Para este exemplo, usaremos quatro ntiles para manter a exibição mais simples, mas qualquer número inteiro positivo é permitido no argumento `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 ntil mais alto, 4, que representa os percentis 75 a 100, tem a faixa mais ampla, abrangendo de 1,4 a 5,4. Por outro lado, os 50% médios dos valores, que incluem os títulos 2 e 3, variam apenas de 0,8 a 1,4.

Além de encontrar o percentil ou ntile para cada linha, podemos calcular percentis específicos em todo o conjunto de resultados de uma consulta. Para fazer isso, podemos usar a função `percentile_cont` ou a função `percentile_disc`. Ambas são funções de janela, mas com uma sintaxe ligeiramente diferente de outras funções de janela discutidas anteriormente porque requerem uma cláusula `WITHIN GROUP`. A forma das funções é:

```
percentile_cont(numeric) within group (order by field_name) over (partition by field_name)
```

O 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 retornar o percentil, bem como a ordenação.

*ASC* ou *DESC* podem ser adicionados opcionalmente, sendo *ASC* o padrão, como em todas as cláusulas *ORDER BY* em SQL. A cláusula *OVER (PARTITION BY...)* é opcional (e confusamente, alguns bancos de dados não a suportam, então verifique sua documentação se encontrar erros).

A função `percentile_cont` retornará um valor interpolado (calculado) que corresponde ao percentil exato, mas que pode não existir no conjunto de dados. A `percentile_disc` (percentil descontínuo), por outro lado, retorna o valor no conjunto de dados mais próximo do percentil solicitado. Para grandes conjuntos de dados, ou para aqueles com valores razoavelmente contínuos, geralmente há pouca diferença prática entre a saída das duas funções, mas vale a pena considerar qual é mais apropriado para sua análise. Vamos dar uma olhada em um exemplo para ver como isso se parece na prática. Calcularemos as magnitudes de percentil 25, 50 (ou mediana) e 75 para todas as magnitudes não nulas no Alasca Central:

```
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 no conjunto de dados. Observe que os valores correspondem aos valores máximos para ntile 1, 2 e 3 calculados no exemplo anterior. Percentis para campos diferentes podem ser calculados dentro da mesma consulta alterando o 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
```

Ao contrário de outras funções, `percentile_cont` e `percentile_disc` requerem uma cláusula *GROUP BY* no nível da consulta quando outros campos estão presentes na consulta. Por exemplo, se quisermos considerar duas áreas dentro do Alasca, e assim incluir o campo `place`, a consulta também deverá incluí-lo no *GROUP BY*, e os percentis são calculados por `place`:

```

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 estas funções, podemos encontrar qualquer percentil necessário para análise. Como o valor da mediana é tão comumente calculado, vários bancos de dados implementaram uma função `median` que possui apenas um argumento, o campo para o qual calcular a mediana. Esta é uma sintaxe útil e certamente muito mais simples, mas observe que o mesmo pode ser feito com `percentile_cont` se uma função `median` não estiver disponível.



As funções `percentile` e `median` podem ser lentas e computacionalmente intensivas em grandes conjuntos de dados. Isso ocorre porque o banco de dados deve classificar e classificar todos os registros, geralmente na memória. Alguns fornecedores de banco de dados implementaram versões aproximadas das funções, como `approximant_percentile`, que são muito mais rápidas e retornam resultados muito próximos da função que calcula todo o conjunto de dados.

Encontrar os percentis ou tis de um conjunto de dados nos permite adicionar alguma quantificação às anomalias. Veremos mais adiante neste capítulo como esses valores também nos fornecem algumas ferramentas para lidar com anomalias em conjuntos de dados. Como os percentis são sempre dimensionados entre 0 e 100, no entanto, eles não dão uma ideia de quão incomuns certos valores são. Para isso, podemos recorrer a funções estatísticas adicionais suportadas pelo SQL.

Para medir quão extremos são os valores em um conjunto de dados, podemos usar o *desvio padrão*. O desvio padrão é uma medida da variação em um conjunto de valores. Um valor mais baixo significa menos variação, enquanto um número mais alto significa mais variação. Quando os dados são normalmente distribuídos em torno da média, cerca de 68% dos valores estão dentro de  $+/-$  um desvio padrão da média e cerca de 95% estão dentro de dois desvios padrão. O desvio padrão é calculado como a raiz quadrada da soma das diferenças da média, dividida pelo número de observações:

$$\sqrt{\sum (x_i - \mu)^2 / N}$$

Nesta fórmula,  $x_i$  é uma observação,  $\mu$  é a média de todas as observações,  $\Sigma$  indica que todos os valores devem ser somados, e  $N$  é o número de observações. Consulte qualquer bom texto estatístico ou recurso on-line<sup>1</sup> 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 toda a população, como costuma ser o caso de um conjunto de dados do cliente, use o `stddev_pop`. O `stddev_samp` encontra o desvio padrão de uma amostra e difere da fórmula acima dividindo 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 de toda a população é usada. A função `stddev` disponível em muitos bancos de dados é idêntica à função `stddev_samp` e pode ser usada simplesmente porque é mais curta. Se você estiver trabalhando com dados que são uma amostra, como de uma pesquisa ou estudo de uma população maior, use o `stddev_samp` ou `stddev`. Na prática, quando você está trabalhando com grandes conjuntos de dados, geralmente há pouca diferença entre os resultados `stddev_pop` e `stddev_samp`. Por exemplo, nos 1,5 milhão de registros na tabela `earthquakes`, os valores divergem somente 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 pequenas o suficiente para na maioria das aplicações práticas, não importa qual função de desvio padrão você usa.

Com esta função, agora podemos calcular o número de desvios padrão da média para cada valor no conjunto de dados. Esse valor é conhecido como *z-score* e é uma forma de padronizar os dados. Os valores que estão acima da média têm um escore z positivo e aqueles abaixo da média têm um escore z negativo. [Figura 6-2](#) mostra como os escores z e os desvios padrão se relacionam com a distribuição normal.

---

<sup>1</sup> <https://www.mathsisfun.com/data/standard-deviation-formulas.html> tem uma boa explicação.

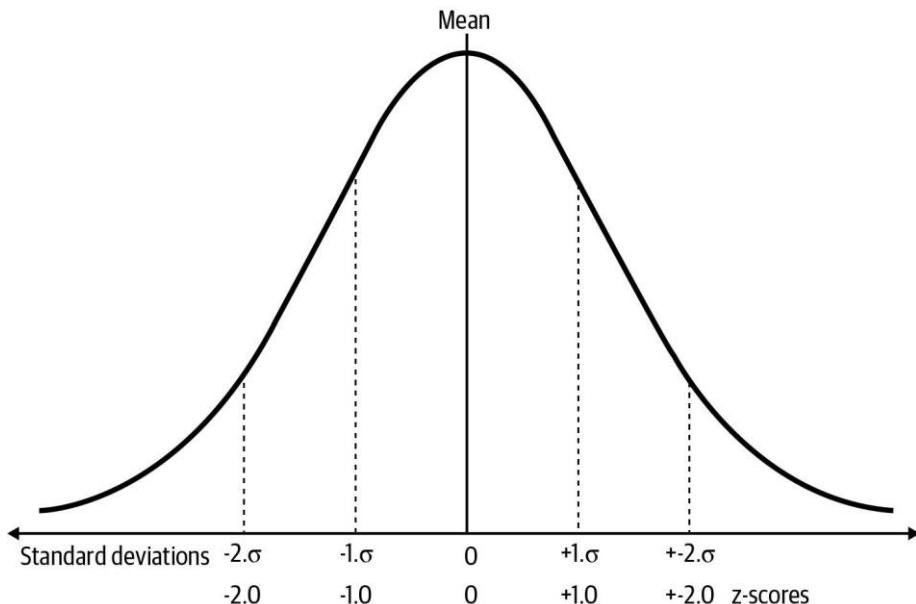


Figura 6-2. Desvios padrão e escores z para uma distribuição normal

Para encontrar os escores z para os terremotos, primeiro calcule a média e o desvio padrão para todo o conjunto de dados em uma subconsulta. Então *JOIN* cartesiano *JOIN*, de modo que os valores médios e de desvio padrão sejam *JOINED* para cada linha de terremoto. Isso é feito com a sintaxe `1 = 1`, pois a maioria dos bancos de dados exige que algum *JOIN* seja especificada.

Na consulta externa, subtraia a magnitude média de cada magnitude individual e depois 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_score
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	-3.2389
Nevada		-2.6	1.6251	1.2736	-3.3174
Nevada		-2.6	1.6251	1.2736	-3.3174

Os maiores terremotos têm um z-score de quase 6, enquanto os menores (excluindo os terremotos -9 e -9,99 que parecem ser anomalias de entrada de dados) têm z-scores próximos de 3. Podemos concluir que os maiores terremotos são mais extremos do que os de baixo valor.

## Gráficos para encontrar anomalias visualmente

Além de classificar os dados e calcular percentis e desvios padrão para encontrar anomalias, visualizar os dados em um dos vários formatos de gráfico também pode ajudar a encontrar anomalias. Como vimos nos capítulos anteriores, um ponto forte dos gráficos é sua capacidade de resumir e apresentar muitos pontos de dados de forma compacta. Ao inspecionar gráficos, muitas vezes podemos identificar padrões e discrepâncias que, de outra forma, poderíamos perder se considerarmos apenas a saída bruta. Finalmente, os gráficos auxiliam na tarefa de descrever os dados, e quaisquer problemas potenciais com os dados relacionados a anomalias, para outras pessoas.

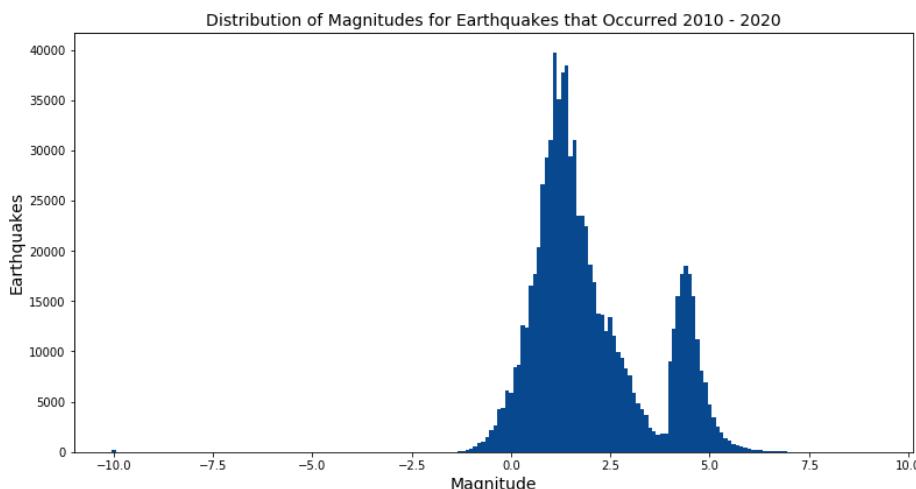
Nesta seção, apresentarei três tipos de gráficos que são úteis para detecção de anomalias: gráficos de barras, gráficos de dispersão e gráficos de caixa. O SQL necessário para gerar a saída para esses gráficos é simples, embora você precise listar estratégias de pivoteamento discutidas nos capítulos anteriores, dependendo dos recursos e limitações do software usado para criar os gráficos. Qualquer grande ferramenta de BI ou software de planilha, ou linguagens como Python ou R, serão capazes de produzir esses tipos de gráficos. Os gráficos nesta seção foram criados usando Python com Matplotlib.

O *gráfico de barras* é usado para plotar um histograma ou distribuição dos valores em um campo e é útil tanto para caracterizar os dados quanto para identificar discrepâncias. A extensão total dos valores é plotada ao longo de um eixo e o número de ocorrências de cada valor é plotado no outro eixo. Os valores extremos altos e baixos são interessantes, assim como a forma do gráfico. Podemos determinar rapidamente se a distribuição é aproximadamente normal (simétrica em torno de um valor de pico ou médio), tem outro tipo de distribuição ou tem picos em valores específicos.

Para representar graficamente um histograma para as magnitudes dos terremotos, primeiro crie um conjunto de dados que agrupe as magnitudes e conte os terremotos. Em seguida, plote 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 considerando nossa exploração anterior dos dados. Ele tem um pico e é aproximadamente simétrico em torno de um valor na faixa de  $1,1$  a  $1,4$  com quase 40.000 terremotos de cada magnitude, mas tem um segundo pico de quase 20.000 terremotos em torno do valor  $4,4$ . Exploraremos o motivo desse segundo pico na próxima seção sobre formas de anomalias. No entanto, os valores extremos são difíceis de detectar neste gráfico, portanto, podemos querer ampliar uma subseção do gráfico, como na [Figura 6-4](#).

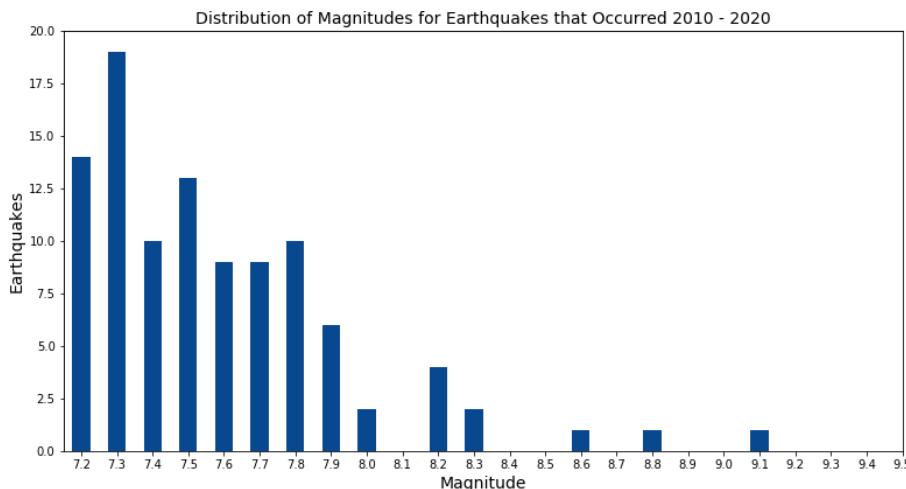


Figura 6-4. Uma visão ampliada da distribuição de magnitudes de terremotos, focada nas magnitudes mais altas

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 apenas 1 à medida que o valor passa de os baixos 7s para mais de 8. Felizmente esses tremores são extremamente raros.

Um segundo tipo de gráfico que pode ser usado para caracterizar dados e identificar outliers é o *dispersão*. Um gráfico 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 representado graficamente para cada par de valores xey no conjunto de dados. Por exemplo, podemos representar graficamente a magnitude em relação à profundidade dos terremotos no conjunto de dados. Primeiro, consulte os dados para criar um conjunto de dados de cada par de valores. Em seguida, faça um gráfico da 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
...	...	...

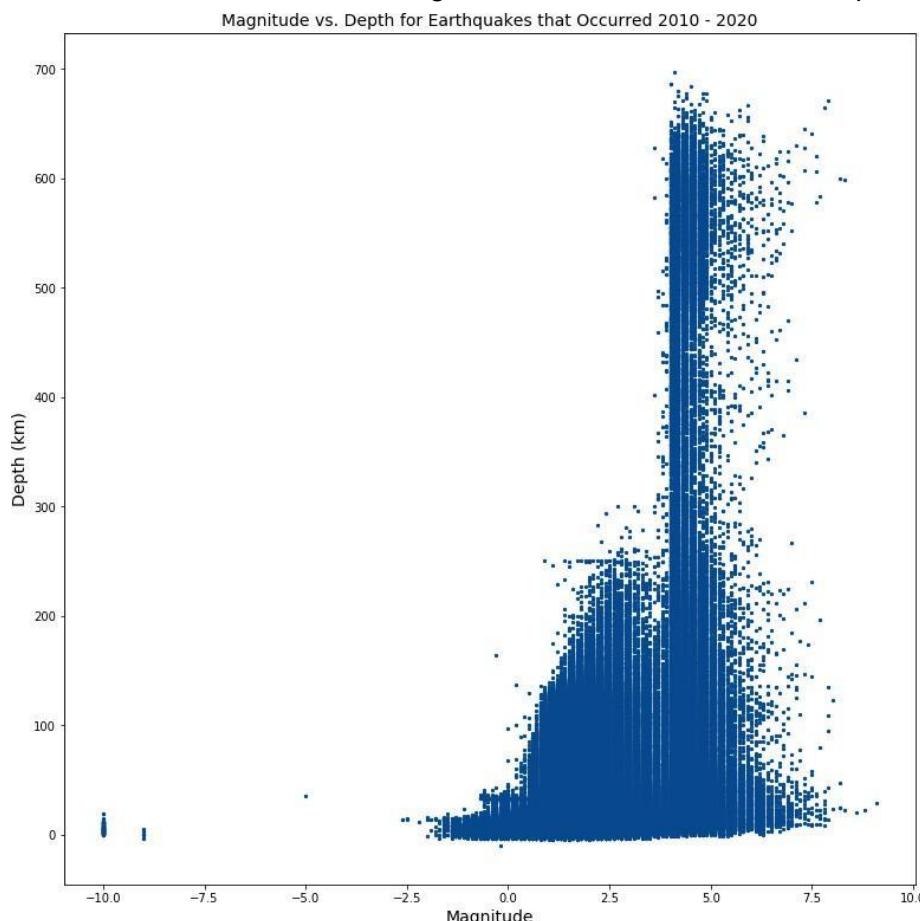
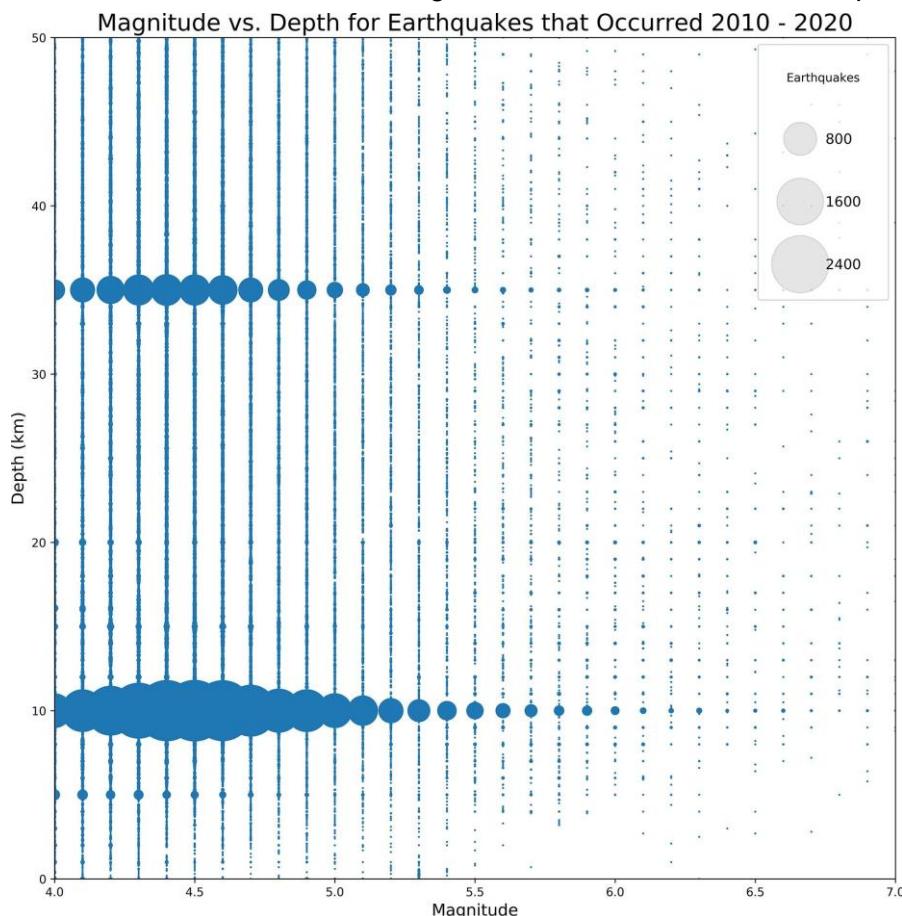


Figura 6 -5.de dispersão da magnitude e profundidade dos terremotos

Neste gráfico, podemos ver a mesma faixa de magnitudes, agora plotadas em relação às profundidades, que variam de pouco abaixo de zero a cerca de 700 quilômetros. Curiosamente, os altos valores de profundidade, acima de 300, correspondem a magnitudes que são aproximadamente 4 e superiores. Talvez esses terremotos profundos só possam ser detectados depois de atingirem uma magnitude mínima. Observe que, devido ao volume de dados, peguei um atalho e agrupei os valores por combinação de magnitude e profundidade, em vez de plotar todos os 1,5 milhão de pontos de dados. A contagem de terremotos pode ser usada para dimensionar cada círculo na dispersão, como na [Figura 6-6](#), que é ampliada para a faixa de magnitudes de 4,0 a 7,0 e profundidades de 0 a 50 km.



*Figura 6-6. Gráfico de dispersão da magnitude e profundidade dos terremotos, ampliado e com círculos dimensionados pelo número de terremotos*

Um terceiro tipo de gráfico útil para encontrar e analisar valores discrepantes é o *box plot*, também conhecido como *box-and-whisker plot*. Esses gráficos resumem os dados no meio do intervalo de valores, mantendo os valores discrepantes. O tipo de gráfico é nomeado para a caixa, ou retângulo, no meio. A linha que forma a parte inferior do retângulo está localizada no valor do percentil 25, a linha que forma o topo está localizada no percentil 75 e a linha no meio está localizada no valor do percentil 50, ou mediana. Os percentis devem ser familiares de nossa discussão na seção anterior. Os “bigodes” do box plot são linhas que se estendem para fora da caixa, normalmente até 1,5 vezes o *intervalo interquartil*.

O intervalo interquartil é simplesmente a diferença entre o valor do percentil 75 e o valor do percentil 25. Quaisquer valores além dos bigodes são plotados no gráfico como valores discrepantes.



Seja qual for o software ou linguagem de programação que você usar para gráficos de caixa, cuidará dos cálculos dos percentis e intervalo interquartil. Muitos também oferecem opções para traçar os bigodes com base em desvios padrão da média ou em percentis mais amplos, como o 10º e o 90º. O cálculo será sempre simétrico em torno do ponto médio (como um desvio padrão acima e abaixo da média), mas o comprimento dos bigodes superiores e inferiores pode diferir com base nos dados.

Normalmente, todos os valores são plotados em um gráfico de caixa. Como o conjunto de dados é muito grande, para este exemplo veremos o subconjunto de 16.036 terremotos que incluem “Japan” no campo `place`. Primeiro, crie o conjunto de dados com SQL, que é um simples *SELECT* de todos os valores `mag` que atendem aos critérios do filtro:

```
SELECT mag
FROM earthquakes
WHERE place like '%Japan%'
ORDER BY 1
;

mag
---
2.7
3.1
3.2
...
```

Em seguida, crie um gráfico de caixa em nosso software gráfico de escolha, conforme mostrado na [Figura 6-7](#).

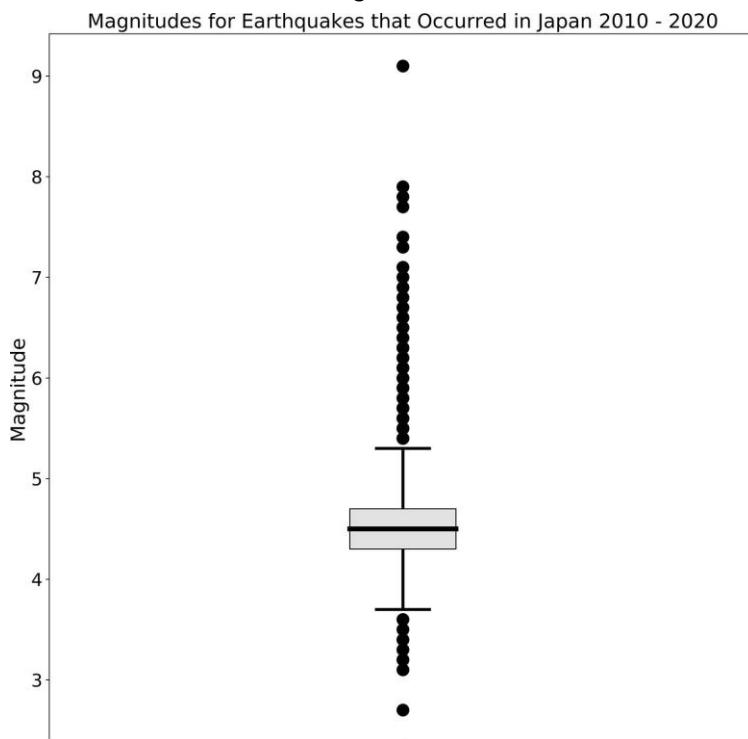


Figura 6-7. Gráfico de caixa mostrando a distribuição de magnitude de terremotos no Japão

Embora o software gráfico geralmente forneça essas informações, também podemos encontrar os valores-chave para o gráfico 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
;

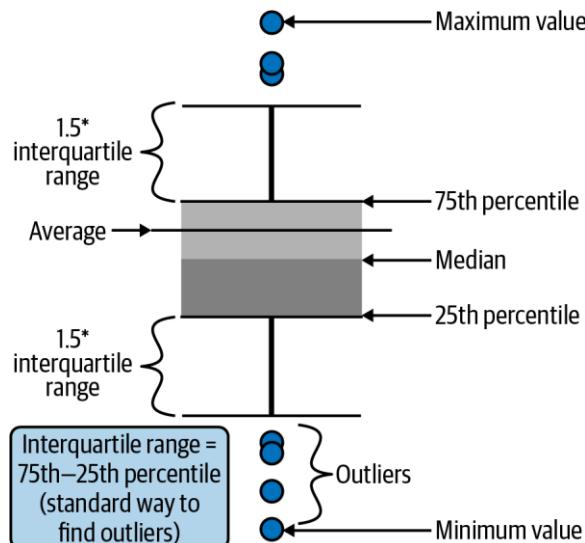
```

ntile_25	median	ntile_75	iqr	lower_whisker	upper_whisker
4.3	4.5	4.7	0.60	3.70	5.30

O terremoto japonês mediano teve uma magnitude de 4,5, e os bigodes se estendem de 3,7 a 5,3. Os círculos plotados representam terremotos atípicos, pequenos e grandes. O Grande Terremoto de Tohoku de 2011, em 9,1, é uma exceção óbvia, mesmo entre os terremotos maiores que o Japão experimentou.



Na minha experiência, os diagramas de caixa são uma das visualizações mais difíceis de explicar para aqueles que não têm experiência em estatística, ou que não passam o dia todo fazendo e analisando visualizações. O intervalo interquartil é um conceito particularmente confuso, embora a noção de outliers pareça fazer sentido para a maioria das pessoas. Se você não tiver certeza absoluta de que seu público sabe como interpretar um box plot, reserve um tempo para explicá-lo em termos claros, mas não excessivamente técnicos. Eu mantengo um desenho como [Figura 6-8](#) que explica as partes de um box plot e o envio junto com meu trabalho “caso” meu público precise de uma atualização.



*Figura 6-8. Diagrama de partes de um gráfico de caixa*

Os gráficos de caixa também podem ser usados para comparar grupos de dados para identificar e diagnosticar melhor onde ocorrem discrepâncias. Por exemplo, podemos comparar terremotos no Japão em anos diferentes. Primeiro adicione o ano do campo `time` na saída SQL e, em seguida, faça o gráfico, 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
...	...

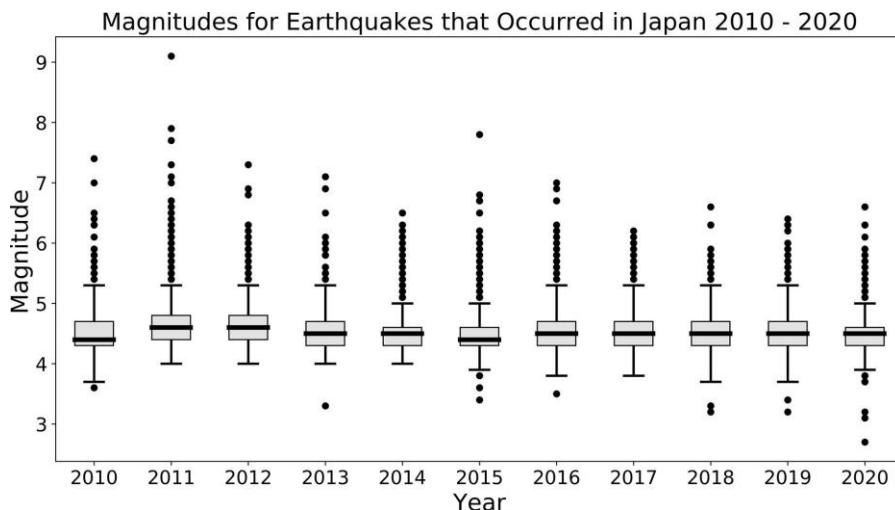


Figura 6-9. Gráfico de caixa de magnitudes de terremotos no Japão, por ano

Embora a mediana e o alcance das caixas variem um pouco de ano para ano, eles estão consistentemente entre 4 e 5. O Japão experimentou grandes terremotos atípicos todos os anos, com pelo menos um maior que 6.0, e em seis dos anos experimentou pelo menos um terremoto igual ou maior que 7.0. O Japão é, sem dúvida, uma região muito sismicamente ativa.

Gráficos de barras, gráficos de dispersão e gráficos de caixa são comumente usados para detectar e caracterizar valores discrepantes em conjuntos de dados. Eles nos permitem absorver rapidamente a complexidade de grandes quantidades de dados e começar a contar a história por trás deles. Juntamente 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 várias formas que as anomalias podem assumir, além das que vimos até agora.

# Formas de Anomalias

As anomalias podem ter todas as formas e tamanhos. Nesta seção, discutirei três categorias gerais de anomalias: valores, contagens ou frequências e presença ou ausência. Esses são pontos de partida para investigar qualquer conjunto de dados, seja como um exercício de criação de perfil ou porque há suspeita de anomalias. Valores atípicos e outros valores incomuns geralmente são específicos de um domínio específico, portanto, em geral, quanto mais você souber sobre como e por que os dados foram gerados, melhor. No entanto, esses padrões e técnicas para detectar anomalias são bons pontos de partida para investigação.

## Valores Anômalos

Talvez o tipo mais comum de anomalia, e a primeira coisa que vem à mente neste tópico, é quando valores únicos são valores discrepantes extremamente altos ou baixos, ou quando valores no meio da distribuição são incomuns.

Na última seção, analisamos várias maneiras de encontrar discrepâncias, por meio de classificação, percentis e desvios padrão e gráficos. Descobrimos que o conjunto de dados de terremotos tem valores incomumente grandes para a magnitude e alguns valores que parecem incomumente pequenos. As magnitudes também contêm números variados de *dígitos significativos*, ou dígitos à direita do ponto decimal. Por exemplo, podemos olhar para um subconjunto de valores em torno de 1 e encontrar um padrão que se repete 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 há um valor com 8 algarismos significativos. Muitos valores têm dois dígitos significativos, mas ter apenas um único dígito significativo é mais comum. Isso provavelmente se deve a diferentes níveis de precisão nos instrumentos que coletam os dados de magnitude. Além disso, o banco de dados não exibe um segundo dígito significativo quando esse dígito é zero, então “1,10” aparece simplesmente como “1,1”. No entanto, o grande número de registros em “1,1” indica que isso não é apenas um problema de exibição.

Dependendo do objetivo da análise, podemos ou não querer ajustar os valores para que todos tenham o mesmo número de dígitos significativos por arredondamento.

Muitas vezes, além de encontrar valores anômalos, é útil entender por que eles aconteceram ou outros atributos correlacionados com anomalias. É aqui que a criatividade e o trabalho de detetive de dados entram em jogo. Por exemplo, 1.215 registros no conjunto de dados têm valores de profundidade muito altos de mais de 600 quilômetros. Podemos querer saber onde esses valores discrepantes ocorreram ou como eles foram coletados. Vamos dar uma olhada na fonte, que podemos encontrar no campo `net` (for network):

```
SELECT net, count(*)
FROM earthquakes
WHERE depth > 600
GROUP BY 1
;

net  count
---  ---
us   1215
```

O site do USGS indica que esta fonte é o [USGS National Earthquake Information Center, PDE](#). Isso não é muito informativo, no entanto, vamos verificar os valores `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 muito profundos acontecem 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 summarização. Podemos aplicar alguma análise de texto para focar no próprio local para obter melhores insights. Para lugares que contêm alguns valores e depois “of” e mais alguns valores, divida 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 confiança que a maioria dos valores muito profundos foram registrados para terremotos em algum lugar em Fiji, com uma concentração particular em torno da pequena ilha vulcânica de Ndoi. A análise pode continuar a ficar mais complexa, por exemplo, analisando o texto para agrupar todos os terremotos registrados na região, o que revelaria que depois de Fiji, outros terremotos muito profundos foram registrados em Vanuatu e nas Filipinas.

As anomalias podem vir na forma de erros ortográficos, variações de capitalização ou outros erros de texto. A facilidade de encontrá-los depende do número de valores distintos, ou *cardinalidade*, do campo. As diferenças na capitalização podem ser detectadas contando os valores distintos e os valores distintos 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

```

Existem 24 valores distintos do campo `type`, mas 25 formas diferentes. Para encontrar os tipos específicos, podemos usar um cálculo para sinalizar os valores cuja forma minúscula não corresponde ao valor real. Incluir a contagem de registros para cada formulário ajudará a contextualizar para que possamos decidir mais tarde como lidar com os valores:

```

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 detectar, pois é o único valor para o qual o cálculo do sinalizador retorna `false`. Como há apenas um registro com esse valor, comparado a 10.136 com a forma minúscula, podemos supor que ele pode ser agrupado com os demais registros. Outras funções de texto podem ser aplicadas, como `trim` se suspeitarmos que os valores contêm espaços extras à esquerda ou à direita, ou `replace` se suspeitarmos que certas grafias têm várias formas, como o número "2" e a palavra "dois".

Erros de ortografia podem ser mais difíceis de descobrir do que outras variações. Se existir um conjunto conhecido de valores e grafias corretos, ele pode ser usado para validar os dados por meio de um *OUTER JOIN* para uma tabela contendo os valores ou com uma instrução CASE combinada com uma lista IN. Em ambos os casos, o objetivo é sinalizar valores inesperados ou inválidos. Sem esse conjunto de valores corretos, nossas opções geralmente são aplicar o conhecimento do domínio ou fazer suposições educadas. Na tabela `earthquakes`, podemos ver os valores `type` com apenas alguns registros e então tentar determinar se existe 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
```

Nós analisamos "Ice Quake" anteriormente e decidimos que provavelmente era o mesmo que "ice quake". Há apenas um registro para "deslizamento de rochas", embora possamos considerar isso próximo o suficiente de outro dos valores, "deslizamento de terra", que tem 15 registros. "Colapso" é mais ambíguo, uma vez que o conjunto de dados inclui tanto "colapso da mina" quanto "colapso do edifício". O que fazemos com eles, ou se fazemos alguma coisa, depende do objetivo da análise, como discutirei mais adiante em "[Como lidar com anomalias](#)" na página 260.

## Contagens ou Frequências Anômalas

Às vezes, as anomalias não vêm na forma de valores individuais, mas na forma de padrões ou agrupamentos de atividades nos dados. Por exemplo, um cliente gastando US\$ 100 em um site de comércio eletrônico pode não ser incomum, mas esse mesmo cliente gastando US\$ 100 a cada hora ao longo de 48 horas quase certamente seria uma anomalia.

Existem várias dimensões nas quais os agrupamentos de atividades podem indicar anomalias, muitas delas dependentes do contexto dos dados. A hora e o local são comuns em muitos conjuntos de dados e são recursos do conjunto de dados `earthquakes`, então vou usá-los para ilustrar as técnicas nesta seção. Tenha em mente que essas técnicas geralmente podem ser aplicadas a outros atributos também.

Eventos que acontecem com frequência incomum em um curto período de tempo podem indicar atividade anômala. Isso pode ser bom, como quando uma celebridade promove inesperadamente um produto, levando a uma explosão de vendas deste produto. Eles também podem ser ruins, como quando picos incomuns indicam uso fraudulento de cartão de crédito ou tentativas de derrubar um site com uma enxurrada de tráfego. Para entender esses tipos de anomalias e se há desvios da tendência normal, primeiro aplicamos as agregações apropriadas e, em seguida, usamos as técnicas apresentadas anteriormente neste capítulo, juntamente com as técnicas de análise de séries temporais discutidas no [Capítulo 3](#).

Nos exemplos a seguir, passarei por uma série de etapas e consultas que nos ajudarão a entender os padrões normais e procurar os incomuns. Este é um processo iterativo que usa perfis de dados, conhecimento de domínio e insights de resultados de consultas anteriores para orientar cada etapa. Começaremos nossa jornada verificando as contagens de terremotos por ano, o que podemos fazer truncando o campo `time` para o nível do ano e contando os registros. Para bancos de dados que não suportam `date_trunc`, considere `extract` ou `trunc` em vez disso:

```
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 isso 2012 teve um baixo número de terremotos em comparação com outros anos. Houve também um aumento acentuado nos registros em 2018 que se manteve até 2019 e 2020. Isso parece incomum, e podemos supor que a Terra se tornou mais ativa sismicamente de repente, que há um erro nos dados, como duplicação de registros ou que algo mudou no processo de coleta de dados. Vamos detalhar o nível do mês para ver se essa tendência persiste em um nível de tempo mais granular:

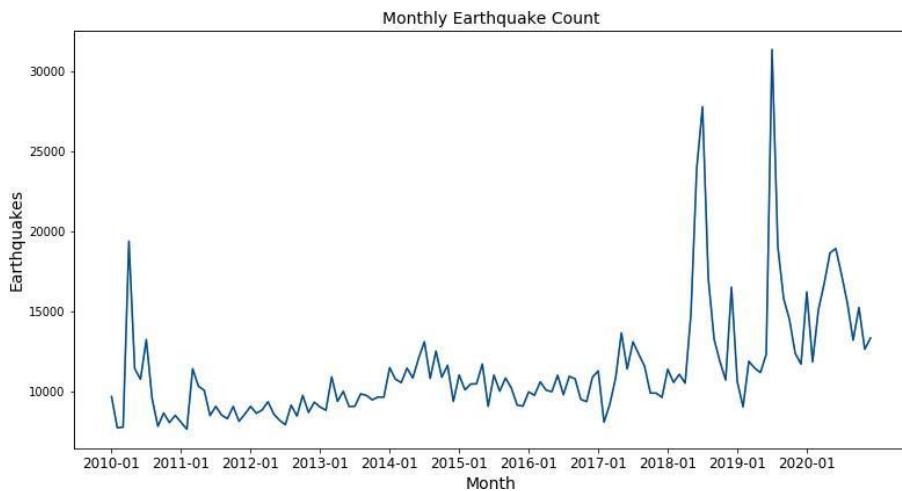
```

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 é exibida na [Figura 6-10](#). Podemos ver que, embora o número de terremotos varie de mês para mês, parece haver um aumento geral a partir de 2017. Também podemos ver que existem três meses atípicos, em abril de 2010, julho de 2018 e julho de 2019.



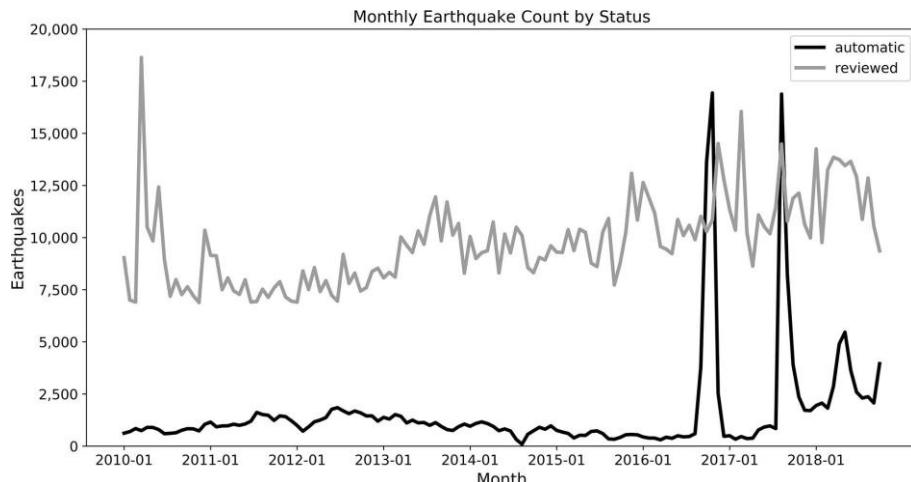
*Figura 6-10. Número de terremotos por mês*

A partir daqui, podemos continuar verificando os dados em períodos de tempo mais granulares, talvez filtrando opcionalmente o conjunto de resultados por um intervalo de datas para focar nesses períodos anômalos de tempo. Depois de restringir os dias específicos ou até mesmo as horas do dia para identificar quando os picos ocorreram, talvez queiramos dividir ainda mais os dados por outros atributos no conjunto de dados. Isso pode ajudar a explicar as anomalias ou pelo menos diminuir as condições em que elas ocorreram. Por exemplo, verifica-se que o aumento dos terremotos a partir de 2017 pode ser explicado pelo menos parcialmente pelo campo `status`. O status indica se o evento foi revisado por um humano (“revisado”) ou foi postado diretamente por um sistema sem revisão (“automático”):

```
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 do status “automático” e “revisado” são plotadas na Figura 6-11.



*Figura 6-11. Número de terremotos por mês, dividido por status*

No gráfico, podemos ver que as contagens discrepantes em julho de 2018 e julho de 2019 se devem a grandes aumentos no número de terremotos de status “automático”, enquanto o pico em abril de 2010 foi em terremotos com status “revisado”.

Um novo tipo de equipamento de gravação automática pode ter sido adicionado ao conjunto de dados em 2017, ou talvez ainda não tenha havido tempo suficiente para revisar todas as gravações.

Analisar a localização em conjuntos de dados que possuem essas informações pode ser outra maneira poderosa de encontrar e entender anomalias. A tabela `earthquakes` contém informações sobre muitos milhares de terremotos muito pequenos, potencialmente obscurecendo nossa visão dos terremotos muito grandes e muito notáveis. Vamos olhar para os locais dos maiores terremotos, aqueles de magnitude 6 ou maior, e ver onde eles se agrupam 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
```

Em contraste com o tempo, onde consultamos em níveis progressivamente mais granulares, os valores `place` já são tão granulares que é um pouco difícil compreender o quadro completo, embora a região de Honshu, no Japão, se destaque claramente. Podemos aplicar algumas das técnicas de análise de texto do [Capítulo 5](#) para analisar e agrupar as informações geográficas. Nesse caso, usaremos `split_part` para remover o texto de 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, sofreu 89 terremotos, tornando-se não apenas o local do maior terremoto no conjunto de dados, mas também uma exceção no número de terremotos muito grandes registrados. Poderíamos continuar analisando, limpando e agrupando os valores `place` para obter uma imagem mais refinada de onde ocorrem os principais terremotos no mundo.

Encontrar contagens, somas ou frequências anômalas nos dados geralmente é um exercício que envolve várias rodadas de consulta sucessiva de diferentes níveis de granularidade. É comum começar de forma ampla, depois ir mais granular, diminuir o zoom novamente para comparar com as tendências da linha de base e aumentar o zoom novamente em divisões ou dimensões específicas dos dados. Felizmente, o SQL é uma ótima ferramenta para esse tipo de iteração rápida. A combinação de técnicas, especialmente de análise de séries temporais, discutida no [Capítulo 3](#), e análise de texto, discutida no [Capítulo 5](#), trará ainda mais riqueza à análise.

## Anomalias da Ausência de Dados

Vimos como frequências excepcionalmente altas de eventos podem sinalizar anomalias. Tenha em mente que a ausência de registros também pode sinalizar anomalias. Por exemplo, o batimento cardíaco de um paciente submetido a uma cirurgia é monitorado. A ausência de batimentos cardíacos a qualquer momento gera um alerta, assim como irregularidades nos batimentos cardíacos. Em muitos contextos, no entanto, detectar a ausência de dados é difícil se você não estiver procurando especificamente por eles. Os clientes nem sempre anunciam que estão prestes a sair. Eles simplesmente param de usar o produto ou serviço e saem silenciosamente do conjunto de dados.

Uma maneira de garantir que as ausências nos dados sejam percebidas é usar técnicas de análise de coorte, discutidas no [Capítulo 4](#). Em particular, um `JOIN` para uma série de datas ou dimensão de dados, para garantir que exista um registro para cada entidade, esteja ou não presente naquele período de tempo, facilita a detecção de ausências.

Outra maneira de detectar ausência é consultar lacunas ou tempo desde a última visualização. Algumas regiões são mais propensas a grandes terremotos devido à maneira como as placas tectônicas estão dispostas ao redor do globo. Também detectamos um pouco disso nos dados em nossos exemplos anteriores. Terremotos são notoriamente difíceis de prever, mesmo quando temos uma noção de onde eles provavelmente ocorrerão. Isso não impede que algumas pessoas especulem sobre o próximo “grande” simplesmente devido à quantidade de tempo que se passou desde o último. Podemos usar o SQL para encontrar as lacunas entre grandes terremotos e o tempo 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` é analisado e limpo, retornando regiões ou países maiores, juntamente com a hora de cada terremoto, para todos os terremotos de magnitude 5 ou maior. A segunda subconsulta usa uma função `lead` para encontrar o `time` do próximo terremoto, se houver, para cada lugar e hora, e o `gap` entre cada terremoto e o seguinte. A função `max` retorna o terremoto mais recente para cada lugar. A consulta externa calcula os dias desde o último terremoto de 5+ no conjunto de dados, usando a função `extract` para retornar apenas os dias do intervalo que é retornado quando duas datas são subtraídas. Como o conjunto de dados inclui registros apenas até o final de 2020, o carimbo de data/hora “2020-12-31 23:59:59” é usado, embora `current_timestamp` ou uma expressão equivalente seria apropriada se os dados fossem atualizados continuamente. Os dias são extraídos de forma semelhante da média e máxima do valor `gap`.

O tempo desde o último grande terremoto em um local pode ter pouco poder preditivo na prática, mas em muitos domínios, as lacunas e o tempo desde a última vez que as métricas vistas têm aplicações práticas. Compreender as lacunas típicas entre as ações estabelece uma linha de base com a qual a lacuna atual pode ser comparada. Quando o gap atual está dentro do intervalo de valores históricos, podemos julgar que um cliente é retido, mas quando o gap atual é muito maior, o risco de churn aumenta.

O conjunto de resultados de uma consulta que retorna lacunas históricas pode se tornar objeto de uma análise de detecção de anomalias, respondendo a perguntas como o maior tempo que um cliente esteve ausente antes de retornar posteriormente.

## Lidando com anomalias

As anomalias podem aparecer em conjuntos de dados por vários motivos e podem assumir várias formas, como acabamos de ver. Depois de detectar anomalias, o próximo passo é tratá-las de alguma forma. Como isso é feito depende tanto da fonte da anomalia – processo subjacente ou problema de qualidade de dados – quanto do objetivo final do conjunto de dados ou análise. As opções incluem investigação sem alterações, remoção, substituição, redimensionamento e correção upstream.

### Investigação

Encontrar, ou tentar encontrar, a causa de uma anomalia é geralmente o primeiro passo para decidir o que fazer com ela. Essa parte do processo pode ser divertida e frustrante - divertida no sentido de que rastrear e resolver um mistério envolve nossas habilidades e criatividade, mas frustrante no sentido de que muitas vezes estamos trabalhando sob pressão de tempo e rastreando anomalias pode parecer descendo uma série interminável de buracos de coelho, levando-nos a pensar se toda uma análise é falha.

Quando estou investigando anomalias, meu processo geralmente envolve uma série de consultas que alternam entre a busca de padrões e a análise de exemplos específicos. Um verdadeiro valor discrepante é fácil de detectar. Nesses casos, geralmente consultarei a linha inteira que contém o valor discrepante para obter pistas sobre o tempo, a origem e quaisquer outros atributos disponíveis. Em seguida, verificar os registros que compartilham esses atributos para ver se eles têm valores que parecem incomuns. Por exemplo, posso verificar se outros registros no mesmo dia têm valores normais ou incomuns. O tráfego de um determinado site ou as compras de um determinado produto podem revelar outras anomalias.

Depois de investigar a origem e os atributos das anomalias ao trabalhar com dados produzidos internamente em minha organização, entro em contato com as partes interessadas ou proprietários do produto. Às vezes, há um bug ou falha conhecida, mas muitas vezes há um problema real em um processo ou sistema que precisa ser resolvido, e as informações de contexto são úteis. Para conjuntos de dados externos ou públicos, pode não haver uma oportunidade de encontrar a causa raiz. Nesses casos, meu objetivo é reunir informações suficientes para decidir qual das opções discutidas a seguir é apropriada.

### Remoção

Uma opção para lidar com anomalias de dados é simplesmente removê-las do conjunto de dados. Se houver motivos para suspeitar que houve um erro na coleta de dados que possa afetar todo o registro, a remoção é apropriada.

A remoção também é uma boa opção quando o conjunto de dados é grande o suficiente para que a eliminação de alguns registros não afete as conclusões. Outra boa razão para usar a remoção é quando os valores discrepantes são tão extremos que distorcem os resultados o suficiente para que conclusões totalmente inadequadas sejam tiradas.

Vimos anteriormente que o conjunto de dados de `earthquakes` contém vários registros com magnitude de -9,99 e alguns com -9. Como os terremotos aos quais esses valores correspondem são extremamente pequenos, podemos suspeitar que sejam valores errôneos ou simplesmente inseridos quando a magnitude real era desconhecida. A remoção de registros com esses valores é direta 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
```

Antes de remover os registros, no entanto, podemos querer determinar se incluir os valores discrepantes realmente faz diferença na saída. Por exemplo, podemos querer saber se a remoção dos valores discrepantes afeta a magnitude média, pois as médias podem ser facilmente distorcidas pelos valores discrepantes. Podemos fazer isso calculando a média em todo o conjunto de dados, bem como a média excluindo os valores extremamente baixos, usando uma instrução CASE para excluir-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 são diferentes apenas no terceiro dígito significativo (1,625 versus 1,627), que é uma diferença bastante pequena. No entanto, se filtrarmos apenas para o Parque Nacional de Yellowstone, onde ocorrem muitos dos valores -9,99, a diferença é mais dramática:

```
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 estes ainda são valores pequenos, a diferença entre uma média de 0,46 e 0,92 é grande o suficiente para que provavelmente escolhamos remover os valores discrepantes.

Observe que há duas opções para isso: na cláusula *WHERE*, que remove os valores discrepantes de todos os resultados, ou na instrução *CASE*, que os remove apenas de cálculos específicos. A opção escolhida depende do contexto da análise, bem como se é importante preservar as linhas para reter contagens totais ou valores úteis em outros campos.

## Substituição por valores alternativos

Os valores anômalos geralmente podem ser tratados substituindo-os por outros valores, em vez de remover registros inteiros. Um valor alternativo pode ser um padrão, um valor substituto, o valor numérico mais próximo dentro de um intervalo ou uma estatística de resumo, como a média ou a mediana.

Vimos anteriormente que valores nulos podem ser substituídos por um padrão usando a função `coalesce`. Quando os valores não são necessariamente nulos, mas são problemáticos por algum outro motivo, uma instrução *CASE* pode ser usada para substituir um valor padrão. Por exemplo, em vez de relatar todos os vários eventos sísmicos, podemos querer agrupar os tipos que *não* terremotos em um único valor “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

```

Isso reduz a quantidade de detalhes nos dados, é claro, mas também pode ser uma maneira de resumir um conjunto de dados que têm vários valores discrepantes para `type`, como vimos anteriormente. Quando você sabe que os valores discrepantes estão incorretos e sabe o valor correto, substituí-los por uma instrução *CASE* também é uma solução que preserva a linha no conjunto de dados geral. Por exemplo, um 0 extra pode ter sido adicionado ao final de um registro ou um valor pode ter sido registrado em polegadas em vez de milhas.

Outra opção para lidar com valores discrepantes numéricos é 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 médias enganosas que podem resultar de valores discrepantes extremos. *Winsorization* é uma técnica específica para isso, onde os valores discrepantes são definidos para um percentual específico dos dados. Por exemplo, valores acima do percentil 95 são definidos para o valor do percentil 95, enquanto os valores abaixo do percentil 5 são definidos para o valor do percentil 5. Para calcular isso em SQL, primeiro calculamos os valores do 5º e 95º percentil:

```
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 colocar este cálculo em uma subconsulta e então usar uma instrução CASE para manipular valores de configuração para valores discrepantes abaixo do percentil 5 e acima do 95.cartesiano JOIN que nos permite comparar os valores percentuais 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_winsorize
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 percentil 5 é 0,12, enquanto o percentil 95 é 4,5. Os valores abaixo e acima desses limites são alterados para o limite no campo `mag_winsorize`. Os valores entre esses limites permanecem os mesmos. Não há limite de percentil definido para winsorizing. Os percentis 1 e 99 ou mesmo os percentis 0,01 e 99,9 podem ser usados dependendo dos requisitos para a análise e quão prevalentes e extremos são os outliers.

## Redimensionamento

Em vez de filtrar registros ou alterar os valores de valores discrepantes, o redimensionamento de valores fornece um caminho que retém todos os valores, mas facilita a análise e a representação gráfica.

Discutimos o z-score anteriormente, mas vale ressaltar que isso pode ser usado como uma forma de redimensionar valores. O z-score é útil porque pode ser usado com valores positivos e negativos.

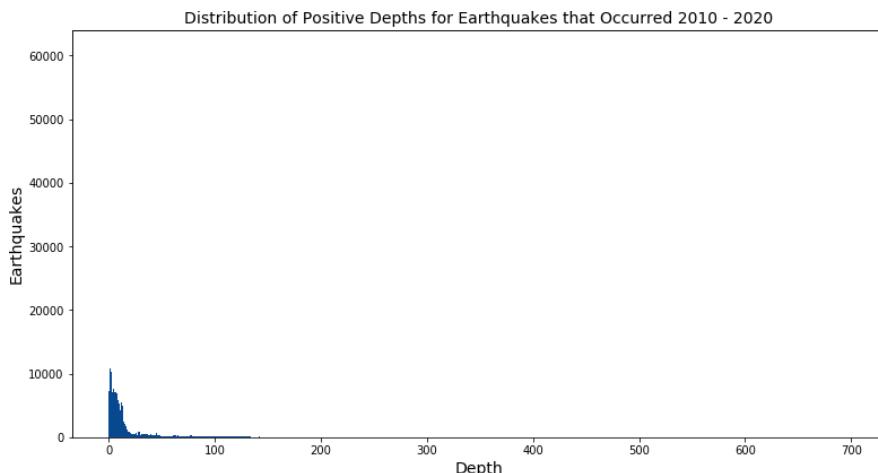
Outra transformação comum é a conversão para escala logarítmica (log). O benefício de transformar valores em escala logarítmica é que eles mantêm a mesma ordenação, mas números pequenos se espalham mais. As transformações de log também podem ser transformadas de volta à escala original, facilitando a interpretação. Uma desvantagem é que a transformação de log não pode ser usada em números negativos. No conjunto de dados de `terremotos`, aprendemos que a magnitude já é expressa em escala logarítmica. A magnitude 9,1 do Grande Terremoto de Tohoku é extrema, mas o valor pareceria ainda mais extremo se não fosse expresso em escala logarítmica!

O campo `depth` é medido em quilômetros. Aqui vamos consultar a profundidade e a profundidade com a função `log` aplicado e, em seguida, representar graficamente a saída nas Figuras 6-12 e 6-13 para demonstrar a diferença. A função `log` usa a base 10 como padrão. Para reduzir o conjunto de resultados para gráficos mais fáceis, a profundidade também é arredondada para um dígito significativo usando a função `round`. A tabela é filtrada para excluir valores menores que 0,05, pois eles seriam arredondados para zero ou menores que 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     -1.000000000000000 6994
0.2     -0.6989700043360188 6876
0.3     -0.5228787452803376 7269
...      ...            ...

```





Outros tipos de transformações de escala, embora não sejam necessariamente apropriados para remover valores discrepantes, podem ser realizados com SQL. Alguns comuns incluem:

- Raiz quadrada: use a função `sqrt`
- Raiz cúbica: use a função `cbrt`
- Transformação recíproca: `1 / field_name`

Altere as unidades, como polegadas para pés ou libras para quilogramas: multiplique ou divida pelo fator de conversão apropriado com `*` ou `/`.

O reescalonamento pode ser feito em código SQL ou, muitas vezes, alternativamente no software ou na linguagem de codificação usada para gráficos. A transformação de log é particularmente útil quando há uma grande dispersão de valores positivos e os padrões que são importantes para detectar existem nos valores mais baixos.

Como em todas as análises, decidir como lidar com anomalias depende da finalidade e da quantidade de conhecimento de contexto ou domínio que você tem sobre o conjunto de dados. A remoção de valores discrepantes é o método mais simples, mas para reter todos os registros, técnicas como winsorizing e redimensionamento funcionam bem.

## Conclusão

A detecção de anomalias é uma prática comum em análise. O objetivo pode ser detectar os outliers ou manipulá-los para preparar um conjunto de dados para análise posterior. Em ambos os casos, as ferramentas básicas de classificação, cálculo de percentis e representação gráfica da saída de consultas SQL podem ajudá-lo a encontrá-los com eficiência. As anomalias vêm em muitas variedades, com valores distantes, explosões incomuns de atividade e ausências incomuns sendo as mais comuns. O conhecimento do domínio é quase sempre útil à medida que você passa pelo processo de localização e coleta de informações sobre as causas das anomalias. As opções para lidar com anomalias incluem investigação, remoção, substituição por valores alternativos e redimensionamento dos dados. A escolha depende muito do objetivo, mas qualquer um desses caminhos pode ser realizado com SQL. No próximo capítulo, voltaremos nossa atenção para a experimentação, onde o objetivo é descobrir se todo um grupo de sujeitos difere da norma do grupo de controle.

## CAPÍTULO 7

# Análise de experimentos

A *experimentação*, também conhecida como *teste A/B* ou *divisão*, é considerada o padrão-ouro para estabelecer causalidade. Muito trabalho de análise de dados envolve o estabelecimento de correlações: uma coisa é mais provável de acontecer quando outra coisa também acontece, seja uma ação, um atributo ou um padrão sazonal. Você provavelmente já ouviu o ditado “correlação não implica causalidade”, no entanto, e é exatamente esse problema na 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 mudança pode ser em uma interface de usuário, um novo fluxo de integração de usuário, um algoritmo que alimenta recomendações, mensagens de marketing ou tempo, ou qualquer outra área. Se a organização o construiu ou tem controle sobre ele, pode ser experimentado, pelo menos em teoria. As hipóteses são muitas vezes impulsionadas por outros trabalhos de análise de dados. Por exemplo, podemos descobrir que uma alta porcentagem de pessoas abandona o fluxo de checkout e podemos supor que mais pessoas podem concluir o processo de checkout se o número de etapas for reduzido.

O segundo elemento necessário para qualquer experimento é uma *métrica de sucesso*. A mudança comportamental que hipotetizamos pode estar relacionada ao preenchimento do formulário, conversão de compra, cliques, retenção, 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 sensível o suficiente para detectar uma mudança. O clique, a conclusão do checkout e o tempo para concluir um processo geralmente são boas métricas de sucesso. Retenção e satisfação do cliente geralmente são métricas de sucesso menos adequadas, apesar de serem muito importantes, porque são frequentemente influenciadas por muitos fatores além do que está sendo testado em qualquer experimento individual e, portanto, são menos sensíveis às mudanças que gostaríamos de testar. As boas métricas de sucesso geralmente são aquelas que você já acompanha como parte da compreensão da saúde da empresa ou organizacional.



Você pode se perguntar se um experimento pode ter várias métricas de sucesso. Certamente com SQL, geralmente é possível gerar muitos cálculos e métricas diferentes. No entanto, você deve estar ciente do problema de comparações múltiplas. Não vou entrar em uma explicação completa aqui, mas a essência é que quanto mais lugares você procurar por uma mudança significativa, maior a probabilidade de encontrar uma. Verifique uma métrica e você poderá ou não encontrar uma alteração significativa em uma das variantes do experimento. Verifique 20 métricas, no entanto, e há uma boa chance de que pelo menos uma mostre significância, independentemente de o experimento ter ou não algo a ver com essa métrica em primeiro lugar. Como regra geral, deve haver uma ou talvez duas métricas principais de sucesso. Uma a cinco métricas adicionais podem ser usadas para proteção contra desvantagens. Às vezes, elas são chamadas de *métricas*. Por exemplo, você pode querer garantir que uma experiência não prejudique o tempo de carregamento da página, mesmo que não seja o objetivo da experiência melhorá-la.

O terceiro elemento da experimentação é um sistema que atribui aleatoriamente entidades a um grupo de variantes de controle ou experimento e altera a experiência de acordo. Este tipo de sistema também é às vezes chamado de *sistema de coorte*. Vários fornecedores de software oferecem ferramentas de coorte de experimentos, embora algumas organizações optem por construí-las internamente para obter mais flexibilidade. De qualquer forma, para realizar a análise de experimentos com SQL, os dados de atribuição em nível de entidade devem fluir para uma tabela no banco de dados que também contém dados comportamentais.



As discussões de experimentos neste capítulo referem-se especificamente a experimentos online, nos quais a atribuição de variantes ocorre por meio de um sistema de computador e o comportamento é rastreado digitalmente. Certamente existem muitos tipos de experimentos realizados nas disciplinas de ciências e ciências sociais. Uma diferença fundamental é que as métricas de sucesso e os comportamentos examinados em experimentos online geralmente já são rastreados para outros fins, enquanto em muitos estudos científicos, o comportamento resultante é rastreado especificamente para o experimento e apenas durante o período do experimento. Com experimentos on-line, às vezes precisamos ser criativos para encontrar métricas que sejam boas proxies para um impacto quando uma medição direta não é possível.

Com uma hipótese, uma métrica de sucesso e um sistema de coorte de variantes em vigor, você pode executar experimentos, coletar os dados e analisar os resultados usando SQL.

# Pontos fortes e limites da análise de experimentos com SQL

SQL é útil para analisar experimentos. Em muitos casos com análise de experimentos, os dados de coorte de experimentos e os dados comportamentais já estão fluindo para um banco de dados, tornando o SQL uma escolha natural. As métricas de sucesso geralmente já fazem parte do vocabulário de relatórios e análises de uma organização, com consultas SQL já desenvolvidas. A junção de dados de atribuição de variantes à lógica de consulta existente geralmente é relativamente simples.

O SQL é uma boa opção para automatizar a geração de relatórios de resultados de experimentos. A mesma consulta pode ser executada para cada experimento, substituindo o nome ou identificador do experimento na cláusula *WHERE*. Muitas organizações com grandes 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 de experimentos, ele apresenta uma grande falha: o SQL não é capaz de calcular a significância estatística. Muitos bancos de dados permitem que os desenvolvedores estendam a funcionalidade SQL com funções *definidas pelo usuário* (UDFs). UDFs podem alavancar testes estatísticos de linguagens como Python, mas estão além do escopo deste livro. Uma boa opção é calcular estatísticas resumidas em SQL e depois usar uma calculadora online como a fornecida em [Evanmiller.org](http://Evanmiller.org) para determinar se o resultado do experimento é estatisticamente significativo.

## Por que a correlação não é causação: como os valores podem se relacionar

É mais fácil provar que dois valores são correlacionados (eles aumentam ou diminuem juntos, ou um existe principalmente na presença do outro) do que provar que uma *causa* o outro. Por que este é o caso? Embora nossos cérebros estejam programados para detectar causalidade, na verdade existem cinco maneiras pelas quais dois valores, X e Y, podem se relacionar:

1. *X causa Y*: É claro que isso é o que todos estamos tentando encontrar. Por algum mecanismo, Y é o resultado de X.
2. *Y causa X*: A relação existe, mas a direção da causalidade é invertida. Por exemplo, guarda-chuvas não causam chuva, mas a presença de chuva faz com que as pessoas usem guarda-chuvas.
3. *X e Y têm uma causa comum*: os valores estão relacionados porque existe uma terceira variável que explica ambos. As vendas de sorvetes e o uso de aparelhos de ar-condicionado aumentam no verão, mas nenhuma causa a outra. Temperaturas mais altas causam ambos os aumentos.

4. *Existe um ciclo de realimentação entre X e Y:* quando Y aumenta, X aumenta para compensar, o que leva a Y aumentando por sua vez, e assim por diante. Isso pode acontecer quando um cliente está em processo de desistência de um serviço. Menos interações levam a menos itens para sugerir ou lembrar, o que leva a menos interações e assim por diante. A falta de recomendações causou menos engajamento ou é o contrário?
5. *Não há relacionamento; é apenas aleatório:* pesquise o suficiente e você encontrará métricas correlacionadas, mesmo que não haja relação real entre elas.

## O conjunto de dados

Para este capítulo, usaremos um conjunto de dados para um jogo para celular do fictício Tanimura Studios. São quatro tabelas. A tabela `game_users` contém registros de pessoas que baixaram o jogo para celular, juntamente com a data e o país. Uma amostra dos dados é mostrada 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 de coisas que os usuários fizeram no jogo. Uma amostra dos dados é mostrada 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 do jogo em dólares americanos. Uma amostra dos dados é mostrada 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*

Finalmente, a tabela **exp\_assignment** contém registros de quais usuários variantes foram atribuídos para um experimento específico. Uma amostra dos dados é mostrada 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 nessas tabelas são fictícios, criados com geradores de números aleatórios, embora a estrutura seja semelhante ao que você pode ver no banco de dados de uma empresa de jogos digitais real.

## Tipos de Experimentos

Existe uma grande variedade de experimentos. Se você pode alterar algo que um usuário, cliente, constituinte ou outra entidade experimenta, você pode, em teoria, testar essa alteração. Do ponto de vista da análise, existem dois tipos principais de experimentos: aqueles com resultados binários e aqueles com resultados contínuos.

### Experimentos com resultados binários: o teste do qui-quadrado

Como você poderia esperar, um experimento de resultado binário tem apenas dois resultados: ou uma ação é executada ou não. Ou um usuário conclui um fluxo de registro ou não. Um consumidor clica em um anúncio do site ou não. Um aluno se forma ou não. Para esses tipos de experimentos, calculamos a proporção de cada variante que completa a ação. O numerador é o número de completadores, 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 nas variantes são estatisticamente diferentes, podemos usar o *teste qui-quadrado*, que é um teste estatístico para variáveis categóricas.<sup>1</sup> Os dados para um teste de qui-quadrado geralmente são mostrados na forma de uma *tabela de contingência*, que mostra a frequência de observações na interseção de dois atributos. Isso se parece com uma tabela dinâmica para aqueles que estão familiarizados com esse tipo de tabela.

Vamos dar uma olhada em um exemplo, usando nosso conjunto de dados de jogos para dispositivos móveis. Um gerente de produto introduziu uma nova versão do fluxo de integração, 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 completam a integração e iniciam sua primeira sessão de jogo. A nova versão foi introduzida em um experimento chamado "Onboarding" que atribuiu os usuários ao controle ou à variante 1, conforme rastreado na tabela `exp_assignment`. Um evento chamado "onboarding complete" na tabela `game_actions` indica se um usuário concluiu o fluxo de integração.

A tabela de contingência mostra a frequência na interseção da atribuição da variante (controle ou variante 1) e se a integração foi concluída ou não. Podemos usar uma consulta para encontrar os valores da tabela. Aqui `count` o número de usuários com e sem uma ação "onboarding complete" e *GROUP BY* a `variant`:

---

<sup>1</sup> Veja <https://www.mathsisfun.com/data/chi-square-test.html> para uma boa explicação deste teste.

```

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

```

Adicionar totais para cada linha e coluna transforma a saída em uma tabela de contingência, como na [Figura 7-5](#).

		Completed onboarding?	
Variant	Yes	No	Total
Control	36,268	13,629	49,897
Variant 1	38,280	11,995	50,275
Total	74,548	25,624	100,172

*Figura 7-5. Tabela de contingência para conclusões de integração*

Para usar uma das calculadoras de significância on-line, precisaremos do número de sucessos, ou vezes em que a ação foi realizada, e o número total coorteado para cada variante. O SQL para encontrar os pontos de dados necessários é simples. A variante atribuída e o `count` de usuários atribuídos a essa variante são consultadas na tabela `exp_assignment`. Em seguida, fizemos o `LEFT JOIN` na tabela `game_actions` para encontrar a `count` de usuários que concluíram a integração. O `LEFT JOIN` é obrigatório, pois esperamos que nem todos os usuários tenham concluído a ação relevante. Finalmente, encontramos a porcentagem concluída em cada variante dividindo o número de usuários que completaram pelo número total 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 a variante 1 realmente teve mais conclusões do que a experiência de controle, com 76,14% completando em comparação com 72,69%. Mas essa diferença é estatisticamente significativa, permitindo-nos rejeitar a hipótese de que não há diferença? Para isso, conectamos nossos resultados a uma calculadora on-line e confirmamos que a taxa de conclusão da variante 1 foi significativamente maior com um nível de confiança de 95% do que a taxa de conclusão do controle. A variante 1 pode ser declarada a vencedora.



Um nível de confiança de 95% é comumente usado, embora essa não seja a única opção. Existem muitos artigos e discussões online sobre o significado dos níveis de confiança, qual nível usar e ajustes em cenários nos quais você está comparando várias variantes a um controle.

Experimentos de resultados binários seguem esse padrão básico. Calcule os sucessos ou conclusões, bem como o total de membros em 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, voltaremos aos experimentos com resultados contínuos.

## Experimentos com resultados contínuos: o teste t

Muitos experimentos buscam melhorar *as métricas contínuas*, em vez dos resultados binários discutidos na última seção. Métricas contínuas podem assumir uma variedade de valores. Os exemplos incluem o valor gasto pelos clientes, o tempo gasto na página e os dias em que um aplicativo é usado. Os sites de comércio eletrônico geralmente desejam aumentar as vendas e, portanto, podem experimentar em páginas de produtos ou fluxos de checkout. Sites de conteúdo podem testar layout, navegação e manchetes para tentar aumentar o número de histórias lidas. Uma empresa que executa um aplicativo pode executar uma campanha de remarketing para lembrar os usuários de voltar ao aplicativo.

Para esses e outros experimentos com métricas de sucesso contínuo, o objetivo é descobrir se os valores médios em 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 95%. O teste estatístico tem três entradas, todas fáceis de calcular com SQL: a média, o desvio padrão e a contagem de observações.

Vamos dar uma olhada em um exemplo usando nossos dados de jogo. Na última seção, analisamos se um novo fluxo de integração aumentou a taxa de conclusão. Agora vamos considerar se esse novo fluxo aumentou os gastos do usuário com a moeda do jogo.

A métrica de sucesso é o valor gasto, portanto, precisamos calcular a média e o desvio padrão desse valor para cada variante. Primeiro precisamos calcular o valor por usuário, pois os usuários podem fazer várias compras. Recupere a atribuição de coorte da tabela `exp_assignment` e `count` os usuários. Em seguida, *LEFT JOIN* para a tabela `game_purchases` para coletar os dados de quantidade. O *LEFT JOIN* é obrigatório, pois nem todos os usuários fazem uma compra, mas ainda precisamos incluí-los nos cálculos de média e desvio padrão. Para usuários sem compras, o valor é definido como padrão de 0 com `coalesce`. Como as funções `avg` e `stddev` ignoram nulos, o padrão 0 é necessário para garantir que esses registros sejam incluídos. A consulta externa resume os valores de 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

```

Em seguida, inserimos esses valores em uma calculadora on-line e descobrimos que não há diferença significativa entre os grupos de controle e variante em um intervalo de confiança de 95%. O grupo "variante 1" parece ter aumentado as taxas de conclusão da integração, mas não o gasto `amount`.

Outra questão que podemos considerar é se a variante 1 afetou os gastos entre os usuários que concluíram a integração. Aqueles que não completam a integração nunca entram no jogo e, portanto, nem têm a oportunidade de fazer uma compra. Para responder a essa pergunta, podemos usar uma consulta semelhante à anterior, mas adicionaremos um *INNER JOIN* à tabela `game_actions` para restringir os usuários contados a apenas aqueles que possuem uma ação de “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 inserção desses valores na calculadora revela que a média do grupo de controle é estatisticamente significativamente maior do que a da variante 1 em um intervalo de confiança de 95%. Esse resultado pode parecer desconcertante, mas ilustra por que é tão importante concordar com a métrica de sucesso de um experimento desde o início. A variante do experimento 1 teve um efeito positivo na conclusão da integração e, portanto, pode ser considerada um sucesso. Não teve efeito sobre o nível geral de gastos. Isso pode ser devido a uma mudança de mix: os usuários adicionais que passaram pela integração na variante 1 eram menos propensos a pagar. Se a hipótese subjacente fosse que aumentar as taxas de conclusão da integração aumentaria a receita, então o experimento não deveria ser considerado um sucesso, e os gerentes de produto deveriam apresentar algumas novas ideias para testar.

## Desafios com experimentos e opções para resgatar experimentos falhos

Embora a experimentação seja o padrão-ouro para entender a causalidade, existem várias maneiras pelas quais os experimentos podem dar errado. Se toda a premissa for falha, não haverá muito que o SQL possa fazer para salvar o dia. Se a falha for de natureza mais técnica, podemos consultar os dados de forma a ajustar ou excluir pontos de dados problemáticos e ainda interpretar alguns resultados. A execução de experimentos tem um custo em termos de tempo gasto por engenheiros, designers ou profissionais de marketing que criam variantes. Também tem *custo de oportunidade*, ou o benefício perdido que poderia ter sido obtido ao enviar os clientes para um caminho de conversão ou experiência de produto ideal. Em um nível prático, usar o SQL para ajudar a organização pelo menos a aprender algo com um experimento geralmente é um tempo bem gasto.

## Atribuição de Variante

A atribuição aleatória de unidades experimentais (que podem ser usuários, sessões ou outras entidades) para grupos de controle e variantes é um dos principais elementos da experimentação. No entanto, às vezes ocorrem erros no processo de atribuição, seja devido a uma falha na especificação do experimento, uma falha técnica ou uma limitação no software de coorte. Como resultado, os grupos de controle e variantes podem ter tamanhos desiguais, menos entidades gerais podem ter sido coorteadas do que o esperado ou a atribuição pode não ter sido realmente aleatória.

O SQL às vezes pode ajudar a salvar um experimento no qual muitas unidades foram agrupadas. Já vi isso acontecer quando um experimento destina-se apenas a novos usuários, mas todos os usuários são agrupados. Outra maneira de isso acontecer é quando um experimento testa algo que apenas um subconjunto de usuários verá, seja porque são alguns cliques na experiência ou porque certas condições devem ser atendidas, como compras anteriores. Devido a limitações técnicas, todos os usuários são agrupados, mesmo que uma parte deles nunca veja o tratamento do experimento, mesmo que estejam neste grupo. A solução é adicionar um *JOIN* no SQL que restringe os usuários ou entidades considerados apenas àqueles que deveriam ser elegíveis. Por exemplo, no caso de um novo experimento de usuário, podemos adicionar um *INNER JOIN* a uma tabela ou subconsulta que contém a data de registro do usuário e definir uma condição *WHERE* para excluir usuários que se registraram muito antes do evento de coorte para serem considerados novos. A mesma estratégia pode ser usada quando uma determinada condição deve ser atendida até mesmo para ver o experimento. Restrinja as entidades incluídas, excluindo aquelas que não deveriam ser elegíveis através das condições *JOINS* e *WHERE*. Depois de fazer isso, você deve verificar se a população resultante é uma amostra grande o suficiente para produzir resultados significativos.

Se um número muito pequeno de usuários ou entidades for coorte, é importante verificar se a amostra é grande o suficiente para produzir resultados significativos. Caso contrário, execute o experimento novamente. Se o tamanho da amostra for grande o suficiente, uma segunda consideração é se há viés em quem ou o que foi coorte. Como exemplo, vi casos em que usuários em determinados navegadores ou versões mais antigas de aplicativos não foram coorteados devido a limitações técnicas. Se as populações que foram excluídas não são aleatórias e representam diferenças de localização, conhecimento técnico ou status socioeconômico, é importante considerar o tamanho dessa população em relação ao restante e se algum ajuste deve ser feito para incluí-la na análise final .

Outra possibilidade é que o sistema de atribuição de variantes seja falho e as entidades não sejam atribuídas aleatoriamente. Isso é bastante incomum com a maioria das ferramentas de experimentos modernas, mas se acontecer, invalidará todo o experimento. Resultados que são “bom demais para ser verdade” podem sinalizar um problema de atribuição de variantes. Já vi, por exemplo, casos em que usuários altamente engajados são atribuídos acidentalmente ao tratamento e ao controle devido a uma alteração na configuração do experimento.

A criação de perfil de dados cuidadosa pode verificar se as entidades foram atribuídas a várias variantes ou se os usuários com alto ou baixo engajamento antes do experimento estão agrupados em uma variante específica.



A execução de um teste A/A pode ajudar a descobrir falhas no software de atribuição de variantes. Nesse tipo de teste, as entidades são coortadas e as métricas de sucesso são comparadas, assim como em qualquer outro experimento. No entanto, nenhuma alteração é feita na experiência e ambas as coortes recebem a experiência de controle. Como os grupos recebem a mesma experiência, não devemos esperar diferenças significativas na métrica de sucesso. Se houver uma diferença, uma investigação mais aprofundada deve ser feita para descobrir e corrigir o problema.

## Outliers

Testes estatísticos para analisar métricas de sucesso contínuo dependem de médias. Como resultado, eles são sensíveis a valores discrepantes excepcionalmente altos ou baixos. Já vi experimentos em que a presença de um ou dois clientes que gastam particularmente em uma variante dá a essa variante uma vantagem estatisticamente significativa sobre as outras. Sem esses poucos grandes gastadores, o resultado pode ser neutro ou até o inverso. Na maioria dos casos, estamos mais interessados em saber se um tratamento tem efeito em uma variedade de indivíduos e, portanto, o ajuste para esses valores discrepantes pode tornar o resultado de um experimento mais significativo.

Discutimos a detecção de anomalias no [Capítulo 6](#), e a análise de experimentos é outro lugar em que essas técnicas podem ser aplicadas. Os valores discrepantes podem ser determinados analisando os resultados do experimento ou encontrando a taxa básica antes do experimento. Os outliers podem ser removidos por meio de uma técnica como winsorizing (também discutida no [Capítulo 6](#)), que remove valores além de um limite, como o percentil 95 ou 99. Isso pode ser feito em SQL antes de passar para o restante da análise do experimento.

Outra opção para lidar com valores discrepantes em métricas de sucesso contínuo é transformar a métrica de sucesso em um resultado binário. Por exemplo, em vez de comparar o gasto médio entre os grupos de controle e variante, que pode ser distorcido devido a alguns gastos muito altos, compare a taxa de compra 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 de controle e variante 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

```

Podemos olhar os números e observar que apesar de haver mais usuários na variante 1, há menos compradores. A porcentagem de usuários que compraram no grupo de controle é de 10%, em comparação com 9,91% para a variante 1. Em seguida, conectamos os pontos de dados em uma calculadora online. A taxa de conversão é estatisticamente significativamente maior para o grupo controle. Nesse caso, embora a taxa de compras tenha sido maior para o grupo de controle, em um nível prático podemos estar dispostos a aceitar esse pequeno declínio se acreditarmos que mais usuários concluindo o processo de integração tem outros benefícios. Mais jogadores podem aumentar as classificações, por exemplo, e os jogadores que gostam do jogo podem divulgá-lo para seus amigos via boca a boca, o que pode ajudar no crescimento e atrair outros novos jogadores que se tornarão compradores.

A métrica de sucesso também pode ser definida para um limite e o compartilhamento de entidades que atendem a esse limite comparado. Por exemplo, a métrica de sucesso pode ser ler pelo menos três histórias ou usar um aplicativo pelo menos duas vezes por semana. Um número infinito de métricas pode ser construído dessa maneira, por isso é importante entender o que é importante e significativo para a organização.

## Time Boxing

Os experimentos geralmente são executados ao longo de várias semanas. Isso significa que os indivíduos que entram no experimento mais cedo têm uma janela mais longa para concluir as ações associadas à métrica de sucesso. Para controlar isso, podemos aplicar o *time boxing* — impondo um período fixo de tempo em relação à data de entrada do experimento e considerando as ações apenas durante essa janela. Este conceito também foi abordado no [Capítulo 4](#).

Para experimentos, o tamanho apropriado da caixa de tempo depende do que você está medindo. A janela pode ser tão curta quanto uma hora ao medir uma ação que normalmente tem uma resposta imediata, como clicar em um anúncio. Para conversão de compra, os experimentadores geralmente permitem uma janela de 1 a 7 dias. Janelas mais curtas permitem que os experimentos sejam analisados mais cedo, uma vez que todas as entidades coorte precisam ter tempo integral para concluir as ações. As melhores janelas equilibram a necessidade de obter resultados com a dinâmica real da organização. Se os clientes normalmente convertem em alguns dias, considere uma janela de 7 dias; se os clientes costumam demorar 20 ou mais dias, considere uma janela de 30 dias.

Como exemplo, podemos revisar nosso primeiro exemplo de experimentos com resultados contínuos, incluindo apenas compras dentro de 7 dias da coorte do evento. Observe que é importante usar a hora em que a entidade foi atribuída a uma variante como ponto de partida da caixa de hora. Uma cláusula ON adicional é adicionada, restringindo os resultados às compras que ocorreram dentro do intervalo “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, de fato, estatisticamente não são significativamente diferentes umas das outras. Neste exemplo, a conclusão com time-box concorda com a conclusão quando não havia time-box.

Nesse caso, os eventos de compra são relativamente raros. Para métricas que medem eventos comuns e aqueles que se acumulam rapidamente, como visualizações de página, cliques, curtidas e artigos lidos, o uso de uma caixa de tempo pode impedir que os primeiros usuários coorteados pareçam substancialmente “melhores” do que os coortes posteriores.

## Experimentos de exposição repetida

Em discussões sobre experimentação online, a maioria dos exemplos são do que eu gosto de chamar de experiências “one-and-done”: o usuário encontra um tratamento uma vez, reage a ele e não passa por aquele caminho novamente. O registro do usuário é um exemplo clássico: um consumidor se inscreve em um determinado serviço apenas uma vez e, portanto, qualquer alteração no processo de inscrição afeta apenas os novos usuários. Analisar testes nessas experiências é relativamente simples.

Há outro tipo de experiência que chamo de “exposição repetida”, na qual um indivíduo entra em contato com a mudança muitas vezes durante o uso de um produto ou serviço.

Em qualquer experimento envolvendo essas mudanças, podemos esperar que os indivíduos as encontrem mais de uma vez. Alterações na interface do usuário de um aplicativo, como cor, texto e posicionamento de informações e links importantes, são experimentadas pelos usuários durante todo o uso do aplicativo. Os programas de marketing por e-mail que enviam lembretes ou promoções aos clientes regularmente também têm essa qualidade de exposição repetida. Os e-mails são vistos muitas vezes como linhas de assunto na caixa de entrada e como conteúdo se abertos.

Medir experimentos de exposição repetida é mais complicado do que medir experimentos únicos devido a efeitos de novidade e regressão à média. Um *efeito de novidade* é a tendência do comportamento mudar apenas porque algo é novo, não porque é necessariamente melhor. A *regressão à média* é a tendência dos fenômenos retornarem a um nível médio ao longo do tempo. Por exemplo, alterar qualquer parte de uma interface de usuário tende a aumentar o número de pessoas que interagem com ela, seja uma nova cor de botão, logotipo ou posicionamento de funcionalidade. Inicialmente, as métricas parecem boas, porque a taxa de cliques ou o engajamento aumentam. Este é o efeito novidade. Mas com o tempo, os usuários se acostumam com a mudança e tendem a clicar ou usar a funcionalidade em taxas que retornam mais próximas da linha de base. Esta é a regressão à média. A questão importante a ser respondida ao executar esse tipo de experimento é se a nova linha de base é maior (ou menor) que a anterior. Uma solução é permitir a passagem de um período de tempo suficientemente longo, no qual você pode esperar que a regressão aconteça, antes de avaliar os resultados. Em alguns casos, isso levará alguns dias; em outros, pode demorar algumas semanas ou meses.

Quando há muitas mudanças, ou o experimento vem em uma série, como campanhas de e-mail ou correio físico, descobrir se todo o programa faz a diferença pode ser um desafio. É fácil reivindicar o sucesso quando os clientes que recebem uma determinada variante de e-mail compram um produto, mas como sabemos se eles teriam feito essa compra de qualquer maneira? Uma opção é configurar um *holdout de longo prazo*. Este é um grupo configurado para não receber mensagens de marketing ou alterações na experiência do produto. Observe que isso é diferente de simplesmente comparar com usuários que optaram por não receber mensagens de marketing, pois geralmente há algum preconceito em quem desativa e quem não. Configurar retenções de longo prazo pode ser complicado, mas há poucas maneiras melhores de medir verdadeiramente os efeitos cumulativos de campanhas e alterações de produtos.

Outra opção é realizar uma análise de coorte (discutida no [Capítulo 4](#)) nas variantes. Os grupos podem ser acompanhados por um período maior de tempo, de semanas a meses. Métricas de retenção ou cumulativas podem ser calculadas e testadas para ver se os efeitos diferem entre as variantes no longo prazo.

Mesmo com os vários desafios que podem ser encontrados com os experimentos, eles ainda são a melhor maneira de testar e provar a causalidade em torno das mudanças feitas nas experiências, desde mensagens de marketing e criatividade até a experiência no produto.

No entanto, muitas vezes encontramos situações menos do que ideais na análise de dados, portanto, a seguir, abordaremos algumas opções de análise para quando o teste A/B não for possível.

## Quando os experimentos controlados não são possíveis: análises alternativas

Os experimentos randomizados são o padrão-ouro para ir além da correlação para estabelecer a causalidade. No entanto, há uma série de razões pelas quais um experimento aleatório pode não ser possível. Pode ser antiético dar tratamentos diferentes para grupos diferentes, particularmente em ambientes médicos ou educacionais. Os requisitos regulamentares podem impedir experimentos em outros ambientes, como serviços financeiros. Pode haver razões práticas, como a dificuldade de restringir o acesso a uma variante de tratamento apenas a um grupo randomizado. Sempre vale a pena considerar se há peças que valem a pena testar ou são testáveis dentro de limites éticos, regulatórios e práticos. Redação, posicionamento e outros elementos de design são alguns exemplos.

Uma segunda situação em que a experimentação não é possível é quando uma mudança aconteceu no passado e os dados já foram coletados. Além de reverter a alteração, voltar e executar um experimento não é uma opção. Às vezes, um analista de dados ou cientista de dados não estava disponível para aconselhar sobre a experimentação. Mais de uma vez, entrei para uma organização e fui solicitado a desvendar os resultados de mudanças que seriam muito mais fáceis de entender se houvesse um grupo de resistência. Outras vezes, a mudança não é intencional. Os exemplos incluem interrupções no site que afetam alguns ou todos os clientes, erros nos formulários e desastres naturais, como tempestades, terremotos e incêndios florestais.

Embora as conclusões causais não sejam tão fortes em situações em que não houve experimento, existem alguns métodos de análise quase experimental que podem ser usados para extrair insights dos dados. Eles se baseiam na construção de grupos a partir dos dados disponíveis que representam as condições de “controle” e “tratamento” o mais próximo possível.

### Pré-/Pós-Análise

Uma pré-/pós-análise compara populações iguais ou semelhantes antes e depois de uma alteração. A medição da população antes da mudança é usada como controle, enquanto a medição após a mudança é usada como variante ou tratamento.

A pré/pós-análise funciona melhor quando há uma mudança claramente definida que aconteceu em uma data bem conhecida, para que os grupos antes e depois possam ser divididos de forma clara. Nesse tipo de análise, você precisará escolher quanto tempo medir antes e depois da mudança, mas os períodos devem ser iguais ou próximos. Por exemplo, se duas semanas se passaram desde uma alteração, 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 mudança.

Se os resultados concordarem em todas essas janelas, você pode ter mais confiança no resultado do que teria se os resultados fossem diferentes.

Vamos percorrer um exemplo. Imagine que o fluxo de integração do nosso jogo para celular inclua uma etapa em que o usuário pode marcar uma caixa para indicar se deseja receber e-mails com novidades do jogo. Isso sempre foi verificado por padrão, mas um novo regulamento exige que agora seja desmarcado por padrão. Em 27 de janeiro de 2020, a mudança foi lançada no jogo e gostaríamos de descobrir se isso teve um efeito negativo nas taxas de inscrição por e-mail. Para fazer isso, compararemos as duas semanas antes da alteração com as duas semanas após a alteração e veremos se a taxa de adesão é estatisticamente significativamente diferente. Poderíamos usar períodos de uma semana ou três semanas, mas duas semanas são escolhidas porque são longas o suficiente para permitir alguma variabilidade de dia da semana e também curtas o suficiente para restringir o número de outros fatores que poderiam afetar a disposição dos usuários para optar.

As variantes são atribuídas na consulta SQL por meio de uma instrução CASE: os usuários que foram criados no intervalo de tempo anterior à alteração são rotulados como “pré”, enquanto os criados após a alteração são rotulados “post”. Em seguida, count o número de usuários em cada grupo da tabela `game_users`. Em seguida, count o número de usuários que optaram por participar, o que é realizado com um *LEFT JOIN* na tabela `game_actions`, restringindo aos registros com a ação “email\_optin”. Em seguida, dividimos os valores para encontrar a porcentagem que optou por participar. Gosto de incluir a contagem de dias em cada variante como uma verificação de qualidade, embora não seja necessário realizar o 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 reconhecerão as datas inseridas como strings, como em '2020-01-13'. Se o seu banco de dados não tiver, 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 mudança tiveram uma taxa de aceitação de e-mail muito maior — 58,75%, em comparação com 40,63% depois. Conectar os valores a uma calculadora online resulta na confirmação de que a taxa para o grupo “pré” é estatisticamente significativamente maior do que a taxa para o grupo “pós”. Neste exemplo, pode não haver muito que a empresa de jogos possa fazer, já que a mudança se deve a uma regulamentação. Testes adicionais podem determinar se o fornecimento de conteúdo de amostra ou outras informações sobre o programa de e-mail pode incentivar mais novos jogadores a participar, se essa for uma meta de negócios.

Ao realizar uma pré/pós-análise, lembre-se de que outros fatores além da mudança que você está tentando conhecer podem causar um aumento ou uma diminuição na métrica. Eventos externos, sazonalidade, promoções de marketing e assim por diante podem mudar drasticamente o ambiente e a mentalidade dos clientes, mesmo em poucas semanas. Como resultado, esse tipo de análise não é tão bom quanto um experimento aleatório verdadeiro para provar a causalidade. No entanto, às vezes essa é uma das poucas opções de análise disponíveis, podendo gerar hipóteses de trabalho que podem ser testadas e refinadas em futuros experimentos controlados.

## Análise de experimentos naturais

Um *experimento natural* ocorre quando entidades acabam com experiências diferentes por meio de algum processo que se aproxima da aleatoriedade. Um grupo recebe a experiência normal ou de controle, e outro recebe alguma variação que pode ter um efeito positivo ou negativo. Normalmente, eles não são intencionais, como quando um bug de software é introduzido ou quando um evento ocorre em um local, mas não em outros locais. Para que esse tipo de análise tenha validade, devemos ser capazes de determinar claramente quais entidades foram expostas. Além disso, é necessário um grupo de controle o mais semelhante possível ao grupo exposto.

O SQL pode ser usado para construir as variantes e calcular o tamanho das coortes e eventos de sucesso no caso de um evento de resultado binário (ou a média, desvio padrão e tamanhos de população no caso de um evento de resultado contínuo). Os resultados podem ser conectados a uma calculadora online como em qualquer outro experimento.

Como exemplo no conjunto de dados de videogame, imagine que, durante o período de tempo de nossos dados, os usuários no Canadá receberam accidentalmente uma oferta diferente na

página de compra de moeda virtual na primeira vez que olharam para ela: um zero extra foi adicionado a o número de moedas virtuais em cada pacote. Assim, por exemplo, em vez de 10 moedas, o usuário receberia 100 moedas de jogo, ou em vez de 100 moedas, receberia 1.000 moedas de jogo e assim por diante. A pergunta que gostaríamos de responder é se os canadenses se converteram em compradores a uma taxa mais alta do que outros usuários. Em vez de comparar com toda a base de usuários, compararemos apenas com usuários nos Estados Unidos. Os países são próximos geograficamente, e a maioria dos usuários nos dois países fala a mesma língua - e para fins de exemplo, vamos supor que fizemos outras análises mostrando que o comportamento deles é semelhante, enquanto o comportamento dos usuários em outros países difere o suficiente para excluí-los.

Para realizar a análise, criamos as “variantes” a partir de qualquer que seja a característica distintiva — neste caso, o campo `country` da tabela `game_users` — mas observe que às vezes será necessário SQL mais complexo, dependendo do conjunto de dados. As contagens de usuários coorte, e aqueles que compraram, são calculados 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 no Canadá que compraram é de fato maior — 24,83%, em comparação com 11,01% dos usuários nos Estados Unidos. Conectar esses valores a uma calculadora online confirma que a taxa de conversão no Canadá é estatisticamente significativamente maior em um intervalo de confiança de 95%.

A parte mais difícil de analisar um experimento natural tende a ser encontrar uma população comparável e mostrar que as duas populações são semelhantes o suficiente para apoiar as conclusões do teste estatístico. Embora seja praticamente impossível provar que não há fatores de confusão, a comparação cuidadosa da demografia e dos comportamentos da população confere credibilidade aos resultados. Como um experimento natural não é um experimento aleatório verdadeiro, a evidência de causalidade é mais fraca, e isso deve ser observado na apresentação de análises desse tipo.

## Análise de populações em torno de um limite

Em alguns casos, há um valor limite que resulta em algumas pessoas ou outras unidades de sujeitos recebendo tratamento, enquanto outras não. Por exemplo, uma determinada média de notas pode qualificar os alunos para uma bolsa de estudos, um determinado nível de renda pode qualificar as famílias para assistência médica subsidiada ou uma pontuação de risco de rotatividade alta pode acionar um representante de vendas para acompanhar um cliente. Nesses casos, podemos aproveitar a ideia de que os sujeitos em ambos os lados do valor limite são provavelmente bastante semelhantes entre si. Então, em vez de comparar todas as populações que receberam e não receberam a recompensa ou intervenção, podemos comparar apenas aqueles que estavam próximos do limiar tanto no lado positivo quanto no negativo. O nome formal para isso é *design de descontinuidade de regressão* (RDD).

Para realizar esse tipo de análise, podemos construir “variantes” dividindo os dados em torno do valor limite, semelhante ao que fizemos na pré/pós-análise. Infelizmente, não existe uma regra rígida sobre a largura das bandas de valores em ambos os lados do limite. As “variantes” devem ser semelhantes em tamanho e grandes o suficiente para permitir significância na análise dos resultados. Uma opção é realizar a análise várias vezes com alguns intervalos diferentes. Por exemplo, você pode analisar as diferenças entre o grupo “tratado” e o controle quando cada grupo contém indivíduos que se enquadram em 5%, 7,5% e 10% do limite. Se as conclusões dessas análises concordarem, há mais suporte para as conclusões. Se não concordarem, no entanto, os dados podem ser considerados inconclusivos.

Tal como acontece com outros tipos de análise não experimental, os resultados do RDD devem ser considerados como provando a causalidade de forma menos conclusiva. Fatores de confusão potenciais também devem receber atenção cuidadosa. Por exemplo, se clientes com alto risco de churn receberem intervenções de várias equipes ou um desconto especial para incentivá-los a reter, além de uma ligação de um representante de vendas, os dados podem ser contaminados por essas outras alterações.

## Conclusão

A análise de experimentos é um campo rico que geralmente incorpora diferentes tipos de análise vistos em outras partes deste livro, desde a detecção de anomalias até a análise de coorte. A criação de perfil de dados pode ser útil para rastrear problemas que ocorrem. Quando experimentos aleatórios não são possíveis, uma variedade de outras técnicas está disponível e o SQL pode ser usado para criar grupos de controle e variantes sintéticos.

No próximo capítulo, vamos nos voltar para a construção de conjuntos de dados complexos para análise, uma área que reúne vários tópicos que discutimos no livro até agora.

# Criando conjuntos de dados complexos para análise

Nos Capítulos 3 a 7, examinamos várias maneiras pelas quais o SQL pode ser usado para realizar análises de dados em bancos de dados. Além desses casos de uso específicos, às vezes o objetivo de uma consulta é montar um conjunto de dados que seja específico, mas de propósito geral o suficiente para que possa ser usado para realizar uma variedade de análises adicionais. O destino pode ser uma tabela de banco de dados, um arquivo de texto ou uma ferramenta de inteligência de negócios. O SQL necessário pode ser simples, exigindo apenas alguns filtros ou agregações. Muitas vezes, no entanto, o código ou a lógica necessária para obter o conjunto de dados desejado pode se tornar muito complexo. Além disso, esse código provavelmente será atualizado ao longo do tempo, à medida que as partes interessadas solicitarem pontos de dados ou cálculos adicionais. A organização, o desempenho e a capacidade de manutenção de seu código SQL tornam-se críticos de uma forma que não é o caso de análises únicas.

Neste capítulo, discutirei os princípios para organizar o código para que seja mais fácil compartilhar e atualizar. Em seguida, discutirei quando manter a lógica de consulta no SQL e quando considerar a mudança para tabelas permanentes por meio do código ETL (extract-transform-load). Em seguida, explicarei as opções para armazenar resultados intermediários — subconsultas, tabelas temporárias e expressões de tabela comum (CTEs) — e considerações para usá-los em seu código. Finalmente, terminarei com uma análise das técnicas para reduzir o tamanho do conjunto de dados e ideias para lidar com a privacidade dos dados e remover informações de identificação pessoal (PII).

## 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 como as tabelas são unidas e como os filtros são colocados na cláusula *WHERE* — até cálculos complexos que agregam, categorizam, analisam ou executam funções de janela em partições dos dados. Ao criar conjuntos de dados para análise posterior, escolher entre manter a lógica dentro da consulta SQL ou enviá-la upstream para um trabalho de ETL ou downstream para outra ferramenta geralmente é tanto arte quanto ciência.

Conveniência, desempenho e disponibilidade de ajuda de engenheiros são fatores importantes na decisão. Muitas vezes não há uma única resposta certa, mas você desenvolverá intuição e confiança quanto mais tempo trabalhar com SQL.

## Vantagens de usar SQL

SQL é uma linguagem muito flexível. Espero ter convencido você nos capítulos anteriores de que uma ampla variedade de tarefas de preparação e análise de dados pode ser realizada usando SQL. Essa flexibilidade é a principal vantagem de usar SQL ao desenvolver conjuntos de dados complexos.

Nos estágios iniciais de trabalho com um conjunto de dados, você pode executar muitas consultas. O trabalho geralmente começa com várias consultas de criação de perfil para entender os dados. Isso é seguido pela construção da consulta passo a passo, verificando as transformações e agregações ao longo do caminho para ter certeza de que os resultados retornados estão corretos. Isso pode ser intercalado com mais perfis, quando os valores reais forem diferentes de nossas expectativas. Conjuntos de dados complexos podem ser construídos combinando várias subconsultas que respondem a perguntas específicas com *JOINS* ou *UNIONS*. Executar uma consulta e examinar a saída é rápido e permite uma iteração rápida.

Além de contar com a qualidade e pontualidade dos dados nas tabelas, o SQL tem poucas dependências. As consultas são executadas sob demanda e não dependem de um engenheiro de dados ou de um processo de liberação. Frequentemente, as consultas podem ser incorporadas em ferramentas de business intelligence (BI) ou em código R ou Python pelo analista ou cientista de dados, sem solicitar suporte técnico. Quando um stakeholder precisa de outro atributo ou agregação adicionado à saída, as alterações podem ser feitas rapidamente.

Manter a lógica no próprio código SQL é ideal ao trabalhar em uma nova análise e quando você espera que a lógica e o conjunto de resultados sofram alterações com frequência. Além disso, quando a consulta é rápida e os dados são retornados rapidamente às partes interessadas, pode nunca haver a necessidade de mover a lógica para outro lugar.

## Quando construir em ETL em vez disso

Há momentos em que mover a lógica para um processo ETL é uma escolha melhor do que mantê-la toda em uma consulta SQL, especialmente ao trabalhar em uma organização que possui um data warehouse ou data lake. Os dois principais motivos para usar o ETL são desempenho e 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 rapidamente, principalmente em bancos de dados e hardware mais recentes, você inevitavelmente acabará escrevendo algumas consultas com cálculos complexos, envolvendo *JOINS* de grandes tabelas ou *JOINS* cartesianos ,ou fazendo com que o tempo de consulta diminua para minutos ou mais . Um analista ou um cientista de dados pode estar disposto a esperar o retorno de uma consulta.

No entanto, a maioria dos consumidores de dados está acostumada com os tempos de resposta rápidos dos sites e ficará frustrado se tiver que esperar mais do que alguns segundos pelos dados.

O ETL é executado nos bastidores em horários programados e grava o resultado em uma tabela. Como está nos bastidores, pode ser executado por 30 segundos, cinco minutos ou uma hora, e os usuários finais não serão afetados. As agendas geralmente são diárias, mas podem ser definidas para intervalos mais curtos. Os usuários finais podem consultar a tabela resultante diretamente, sem necessidade de *JOINS* ou outra lógica e, assim, experimentar tempos de consulta rápidos.

Um bom exemplo de quando o ETL costuma ser uma escolha melhor do que manter toda a lógica em uma consulta SQL é a tabela de instantâneos diários. Em muitas organizações, manter um instantâneo diário de clientes, pedidos ou outras entidades é útil para responder a perguntas analíticas. Para os clientes, talvez queiramos calcular o total de pedidos ou visitas até a data, o status atual em um pipeline de vendas e outros atributos que mudam ou se acumulam. Vimos como criar séries diárias, inclusive para dias em que uma entidade não estava presente, nas discussões sobre análise de séries temporais no [Capítulo 3](#) e análise de coorte no [Capítulo 4](#). No nível de entidade individual e por longos períodos de tempo, essas consultas podem se tornar lentas. Além disso, atributos como o status atual podem ser substituídos na tabela de origem, portanto, capturar um instantâneo diário pode ser a única maneira de preservar uma imagem precisa do histórico. Desenvolver o ETL e armazenar os resultados diários do snapshot em uma tabela geralmente vale o esforço.

A visibilidade é um segundo motivo para mover a lógica para o ETL. Frequentemente, as consultas SQL existem no computador de um indivíduo ou são enterradas no código do relatório. Pode ser difícil para outras pessoas até mesmo encontrar a lógica embutida na consulta, quanto mais entender e verificar se há erros. Mover a lógica para o ETL e armazenar o código ETL em um repositório como o GitHub torna mais fácil para outras pessoas em uma organização encontrar, verificar e iterar nele. A maioria dos repositórios usados pelas equipes de desenvolvimento também armazena o histórico de alterações, um benefício adicional que permite ver quando uma linha específica em uma consulta foi adicionada ou alterada.

Existem boas razões para considerar colocar lógica no ETL, mas essa abordagem também tem suas desvantagens. Uma é que novos resultados não estão disponíveis até que o trabalho de ETL tenha sido executado e atualizado os dados, mesmo que novos dados tenham chegado à tabela subjacente. Isso pode ser superado continuando a executar SQL nos dados brutos para registros muito novos, mas limitando-os a uma pequena janela de tempo para que a consulta seja executada rapidamente. Opcionalmente, isso pode ser combinado com uma consulta na tabela ETL, usando subconsultas ou *UNION*. Outra desvantagem de colocar a lógica no ETL é que se torna mais difícil alterá-la. Atualizações ou correções de bugs geralmente precisam ser entregues a um engenheiro de dados e o código testado, verificado no repositório e liberado no data warehouse de produção. Por esse motivo, geralmente opto por esperar até que minhas consultas SQL tenham passado do período de iteração rápida e os conjuntos de dados resultantes tenham sido revisados e estejam em uso pela organização, antes de movê-los para o ETL. Obviamente, tornar o código mais difícil de alterar e impor revisões de código são excelentes maneiras de garantir consistência e qualidade dos dados.

## Views como uma alternativa ao ETL

Quando você deseja a reutilização e a visibilidade do código do ETL, mas sem realmente armazenar os resultados e, portanto, ocupar espaço permanentemente, uma *view* pode ser uma boa opção. Uma visualização é essencialmente uma consulta salva com um alias permanente que pode ser referenciado como qualquer outra tabela no banco de dados. A consulta pode ser simples ou complexa e pode envolver junções de tabelas, filtros e quaisquer outros elementos de SQL.

As exibições podem ser usadas para garantir que todos que consultam os dados usem as mesmas definições, conforme definido na consulta subjacente — por exemplo, sempre filtrando as transações de teste. Eles podem ser usados para proteger os usuários da complexidade da lógica subjacente, úteis para usuários mais novatos ou ocasionais de um banco de dados. As visualizações também podem ser usadas para fornecer uma camada extra de segurança, restringindo o acesso a determinadas linhas ou colunas em um banco de dados. Por exemplo, uma exibição pode ser criada para excluir PII, como endereços de e-mail, mas permitir que os escritores de consulta exibam outros atributos de cliente aceitáveis.

As vistas têm algumas desvantagens, no entanto. Eles são objetos no banco de dados e, portanto, requerem permissões para criar e atualizar suas definições. As visualizações não armazenam os dados, portanto, cada vez que uma consulta é executada com uma visualização, o banco de dados deve voltar à tabela ou tabelas subjacentes para buscar os dados. Como resultado, eles não substituem o ETL que cria uma tabela de dados pré-computada.

A maioria dos principais bancos de dados também possui *visualizações materializadas*, que são semelhantes às visualizações, mas armazenam os dados retornados em uma tabela. O planejamento para a criação e atualização de visualizações materializadas geralmente é feito melhor em consulta com um administrador de banco de dados experiente, pois há considerações de desempenho diferenciadas além do escopo deste livro.

## Quando colocar lógica em outras ferramentas

O código SQL e a saída dos resultados da consulta em seu editor de consulta geralmente são apenas parte de uma análise. Os resultados geralmente são incorporados em relatórios, visualizados em tabelas e gráficos ou manipulados em uma variedade de ferramentas, desde planilhas e software de BI até ambientes nos quais o código estatístico ou de aprendizado de máquina é aplicado. Além de escolher quando mover a lógica upstream para ETL, também temos opções sobre quando mover a lógica downstream para outras ferramentas. Tanto o desempenho quanto os casos de uso específicos são fatores-chave na decisão.

Cada tipo de ferramenta tem pontos fortes e limitações de desempenho. As planilhas são muito flexíveis, mas não são conhecidas por serem capazes de lidar com um grande número de linhas ou cálculos complexos em muitas linhas. Os bancos de dados definitivamente têm uma vantagem de desempenho, portanto, geralmente é melhor realizar o máximo de cálculo possível no banco de dados e passar o menor conjunto de dados possível para a planilha.

As ferramentas de BI têm uma variedade de recursos, por isso é importante entender como o software lida com os cálculos e como os dados serão usados. Algumas ferramentas de BI podem *cache* (manter uma cópia local) em um formato otimizado, acelerando os cálculos. Outros emitem uma nova consulta cada vez que um campo é adicionado ou removido de um relatório e, portanto, aproveitam principalmente o poder computacional do banco de dados. Certos cálculos, como `count distinct` e `median`, exigem dados detalhados em nível de entidade. Se não for possível antecipar todas as variações desses cálculos com antecedência, pode ser necessário passar um conjunto de dados maior e mais detalhado do que o ideal. Além disso, se o objetivo for criar um conjunto de dados que permita a exploração e o fatiamento de muitas maneiras diferentes, mais detalhes geralmente são melhores. Descobrir a melhor combinação de computação de ferramentas SQL, ETL e BI pode levar alguma iteração.

Quando o objetivo é realizar análises estatísticas ou de aprendizado de máquina no conjunto de dados usando uma linguagem como R ou Python, os dados detalhados geralmente são melhores. Ambas as linguagens podem executar tarefas que se sobrepõem ao SQL, como agregação e manipulação de texto. Muitas vezes, é melhor realizar o máximo de cálculo possível em SQL, para aproveitar o poder computacional do banco de dados, mas nada mais. Flexibilidade para iterar geralmente é uma parte importante do processo de modelagem. A escolha de realizar cálculos em SQL ou em outra linguagem também pode depender de sua familiaridade e nível de conforto com cada um. Aqueles que são muito confortáveis em SQL podem preferir fazer mais cálculos no banco de dados, enquanto aqueles que são mais proficientes em R ou Python podem preferir fazer mais cálculos lá.



Embora existam poucas regras para decidir onde colocar a lógica, vou encorajá-lo a seguir uma regra em particular: evite etapas manuais. É bastante fácil abrir um conjunto de dados em uma planilha ou editor de texto, fazer uma pequena alteração, salvar e seguir em frente. Mas quando você precisa iterar ou quando novos dados chegam, é fácil esquecer essa etapa manual ou executá-la de forma inconsistente. Na minha experiência, não existe um pedido verdadeiramente “único”. Coloque a lógica no código em algum lugar, se possível.

SQL é uma ótima ferramenta e é incrivelmente flexível. Ele também está dentro do fluxo de trabalho de análise e dentro de um ecossistema de ferramentas. Decidir onde colocar os cálculos pode exigir algumas tentativas e erros à medida que você percorre o que é viável entre as ferramentas SQL, ETL e downstream. Quanto mais familiaridade e experiência você tiver com todas as opções disponíveis, melhor será capaz de estimar as compensações e continuar a melhorar o desempenho e a flexibilidade do seu trabalho.

## Organização de código

SQL tem poucas regras de formatação, o que pode levar a consultas indisciplinadas. As cláusulas de consulta devem estar na ordem correta: *SELECT* é seguido por *FROM* e *GROUP BY* não pode preceder *WHERE*, por exemplo. Algumas palavras-chave como *SELECT* e *FROM* são *reservadas* (ou seja, não podem ser usadas como nomes de campos, nomes de tabelas ou aliases). No entanto, ao contrário de alguns outros idiomas, novas linhas, espaços em branco (além dos espaços que separam palavras) e maiúsculas não são importantes e são ignorados pelo banco de dados. Qualquer uma das consultas de exemplo neste livro poderia ter sido escrita em uma única linha e com ou sem letras maiúsculas, exceto em strings entre aspas. Como resultado, o ônus da organização do código recai sobre a pessoa que escreve a consulta. Felizmente, temos algumas ferramentas formais e informais para manter o código organizado, desde comentários até formatação “cosmética”, como recuo e 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 um comentário e ignorado durante a execução. SQL tem duas opções. A primeira é usar dois traços, o que transforma tudo na linha que segue em um comentário:

```
-- Este é um comentário
```

A segunda opção é usar os caracteres barra (/) e asterisco (\*) para iniciar um bloco de comentários, que pode se estender por várias linhas, seguido por uma estrela e uma barra para finalizar o bloco de comentários:

```
/*
Este é um bloco de
comentários com várias
linhas
*/
```

Muitos editores SQL ajustam o estilo visual do código dentro dos comentários, tornando-os acinzentados ou alterando de outra forma a cor, para torná-los mais fáceis de detectar.

Comentar código é uma boa prática, mas reconhecidamente é uma prática que muitas pessoas lutam para fazer regularmente. O SQL geralmente é escrito rapidamente e, especialmente durante os exercícios de exploração ou criação de perfil, não esperamos manter nosso código a longo prazo. Um código com comentários excessivos pode ser tão difícil de ler quanto um código sem comentários. E todos nós sofremos com a ideia de que, desde que escrevemos o código, sempre seremos capazes de lembrar *por* que escrevemos o código. No entanto, quem herdou uma longa consulta escrita por um colega ou se afastou de uma consulta por alguns meses e depois voltou para atualizá-la saberá que pode ser frustrante e demorado decifrar o código.

Para equilibrar o fardo e o benefício de comentar, tento seguir algumas regras práticas. Primeiro, adicione um comentário em qualquer lugar que um valor tenha um significado não óbvio. Muitos sistemas de origem codificam valores como números inteiros, e seu significado é fácil de esquecer.

Deixar uma nota torna o significado claro e torna o código mais fácil de mudar, se necessário:

```
WHERE status in (1,2) -- 1 is Active, 2 is Pending
```

Segundo, comente sobre quaisquer outros cálculos ou transformações não óbvios. Isso pode ser qualquer coisa que alguém que não tenha passado o tempo criando o perfil do conjunto de dados pode não saber, desde erros de entrada de dados até a existência de valores discrepantes:

```
case when status = 'Live' then 'Active'  
      else status end  
/* we used to call customers Live but in  
2020 we switched it to Active */
```

A terceira prática que tento seguir comentando é deixar notas quando a consulta contém várias subconsultas. Uma linha rápida sobre o que cada subconsulta calcula facilita pular para a parte relevante ao voltar mais tarde para a verificação de qualidade ou editar uma consulta mais longa:

```
SELECT...  
FROM  
( --find the first date for each customer  
    SELECT ...  
    FROM ...  
) a  
JOIN  
( --find all of the products for each customer  
    SELECT ...  
    FROM ...  
) b on a.field = b.field  
...  
;
```

Comentar bem requer prática e alguma disciplina, mas vale a pena fazer para a maioria das consultas que são mais longas do que algumas linhas. Os comentários também podem ser usados para adicionar informações úteis à consulta geral, como finalidade, autor, data de criação e assim por diante. Seja gentil com seus colegas e com você mesmo no futuro, colocando comentários úteis em seu código.

## **Capitalização, recuo, parênteses e outros truques de formatação**

A formatação, e principalmente a formatação consistente, é uma boa maneira de manter o código SQL organizado e legível. Os bancos de dados ignoram letras maiúsculas e espaços em branco (espaços, tabulações e novas linhas) no SQL, para que possamos usá-los a nosso favor para formatar o código em blocos mais legíveis. Os parênteses podem controlar a ordem de execução, que discutiremos mais adiante, e também agrupar visualmente os elementos de cálculo.

As palavras em maiúsculas se destacam do resto, como qualquer pessoa que tenha recebido um e-mail com uma linha de assunto em maiúsculas pode confirmar.

Eu gosto de usar letras maiúsculas apenas para as cláusulas principais: *SELECT*, *FROM*, *JOIN*, *WHERE* e assim por diante. Particularmente em consultas longas ou complexas, ser capaz de identificá-las rapidamente e, assim, entender onde a cláusula *SELECT* termina e a *FROM* começa me economiza muito tempo.

O espaço em branco é outra maneira importante de organizar e tornar as partes da consulta mais fáceis de encontrar e de entender quais partes se juntam logicamente. Qualquer consulta SQL pode ser escrita em uma única linha no editor, mas na maioria dos casos isso levaria a muita rolagem para a esquerda e para a direita no código. Eu gosto de iniciar cada cláusula (*SELECT*, *FROM*, etc.) em uma nova linha, que, junto com a capitalização, me ajuda a acompanhar onde cada uma começa e termina. Além disso, acho que colocar agregações em suas próprias linhas, bem como funções que ocupam algum espaço, ajuda na organização. Para instruções CASE com mais de 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 o *type* e *mag* (magnitude), parse *place*, e então contar os registros na 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 organizado visualmente. Adicionar espaços ou tabulações para alinhar os itens WHEN dentro de uma instrução CASE é um exemplo. Você também viu subconsultas recuadas em exemplos ao longo do livro. Isso faz com que as subconsultas se destaquem visualmente e, quando uma consulta tem vários níveis de subconsultas aninhadas, fica mais fácil ver e entender a ordem em que elas serão avaliadas e quais subconsultas são pares em termos de nível:

```

SELECT...
FROM
(
  SELECT...
  FROM
  (
    SELECT...
  
```

```

        FROM...
) a
JOIN
(
    SELECT...
    FROM
) b on...
) a
...
;

```

Qualquer número de outras opções de formatação podem ser feitas e a consulta retornará os mesmos resultados. Os escritores de SQL de longo prazo tendem a ter suas próprias preferências de formatação. No entanto, a formatação clara e consistente facilita muito a criação, a manutenção e o compartilhamento do código SQL.

Muitos editores de consulta SQL fornecem alguma forma de formatação e coloração de consulta. Normalmente, as palavras-chave são coloridas, tornando-as mais fáceis de identificar em uma consulta. Essas dicas visuais facilitam muito o desenvolvimento e a revisão de consultas SQL. Se você sempre escreveu SQL em um editor de consultas, tente abrir um arquivo `.sql` em um editor de texto simples para ver a diferença que a coloração faz. A Figura 8-1 mostra um exemplo de um editor de consulta SQL, e o mesmo código é mostrado em um editor de texto simples na Figura 8-2 (observe que eles podem aparecer em escala de cinza em algumas versões deste livro).

```

1 SELECT field1, field2, sum(field3) as sum_field3
2 FROM some_table
3 WHERE field1 is not null
4 GROUP BY 1,2
5 HAVING sum_field3 > 100
6 ;
7

```

Figura 8-1. Captura de tela da coloração de palavras-chave no editor de consultas SQL DBVisualizer

```

1 SELECT field1, field2, sum(field3) as sum_field3
2 FROM some_table
3 WHERE field1 is not null
4 GROUP BY 1,2
5 HAVING sum_field3 > 100
6 ;
7

```

Figura 8-2. O mesmo código como texto simples no editor de texto Atom

Formatting 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 ajuda bastante a manter seu código legível, facilitando o compartilhamento e a manutenção.

## Armazenando código

Depois de se dar ao trabalho de comentar e formatar o código, é uma boa ideia armazená-lo em algum lugar, caso você precise usá-lo ou referenciá-lo mais tarde.

Muitos analistas de dados e cientistas trabalham com um editor SQL, geralmente um software de desktop. Os editores SQL são úteis porque geralmente incluem ferramentas para navegar no esquema do banco de dados ao lado de uma janela de código. Eles salvam arquivos com 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.

Como são texto, os arquivos de código SQL são fáceis de armazenar em repositórios de controle de alterações, como o GitHub. Usar um repositório fornece uma boa opção de backup e facilita o compartilhamento com outras pessoas. Os repositórios também rastreiam o histórico de alterações dos arquivos, o que é útil quando você precisa descobrir quando uma alteração específica foi feita ou quando o histórico de alterações é necessário por motivos regulatórios. A principal desvantagem do GitHub e de outras ferramentas é que geralmente não são uma etapa obrigatória no fluxo de trabalho de análise. Você precisa se lembrar de atualizar seu código periodicamente e, como em qualquer etapa manual, é fácil esquecer de fazer isso.

## Organizando Computações

Dois problemas relacionados que enfrentamos ao criar conjuntos de dados complexos são obter a lógica correta e obter um bom desempenho de consulta. A lógica deve estar correta, ou os resultados não terão sentido. O desempenho da consulta para fins de análise, diferentemente dos sistemas transacionais, geralmente tem uma faixa de “bom o suficiente”. As consultas que não retornam são problemáticas, mas a diferença entre esperar 30 segundos e esperar um minuto pelos resultados pode não importar muito. Com o SQL, geralmente há mais de uma maneira de escrever uma consulta que retorna os resultados corretos. Podemos usar isso a nosso favor para garantir a lógica correta e ajustar o desempenho de consultas de longa duração. Existem três maneiras principais de organizar o cálculo de resultados intermediários em SQL: a subconsulta, as tabelas temporárias e as expressões de tabela comum (CTEs). Antes de nos aprofundarmos neles, revisaremos a ordem de avaliação no SQL. Para encerrar a seção, apresentarei `grouping sets`, que podem substituir a necessidade de consultar `UNION` juntos em certos casos.

## Entendendo a Ordem de Avaliação da Cláusula SQL

Os dados traduzem o código SQL em um conjunto de operações que serão realizadas para retornar os dados solicitados. Embora não seja necessário entender exatamente como isso acontece para ser bom em escrever SQL para análise, entender a ordem em que o

banco de dados executará suas operações é incrivelmente útil (e às vezes é necessário depurar resultados inesperados).



Muitos bancos de dados modernos têm otimizadores de consulta sofisticados que consideram várias partes da consulta para criar o plano de execução mais eficiente. Embora eles possam considerar partes da consulta em uma ordem diferente daquela discutida aqui e, portanto, possam precisar de menos otimização de consulta por parte de humanos, eles não calcularão resultados intermediários em uma ordem diferente daquela discutida aqui.

A ordem geral de avaliação é mostrada na [Tabela 8-1](#). As consultas SQL geralmente incluem apenas um subconjunto de cláusulas possíveis, portanto, a avaliação real inclui apenas as etapas relevantes para a consulta.

*Tabela 8-1. Ordem de avaliação da consulta SQL*

1	FROM incluindo <i>JOINS</i> e suas <i>ON</i> cláusulas
2	WHERE
3	GROUP BY incluindo agregações
4	HAVING
5	Funções
6	SELECT
7	DISTINCT
8	UNION
9	ORDER BY
10	LIMIT e OFFSET

Primeiro as tabelas na *FROM* são avaliadas, juntamente com quaisquer *JOINS*. Se a cláusula *FROM* incluir quaisquer subconsultas, elas serão avaliadas antes de prosseguir para o restante das etapas. Em um *JOIN*, a cláusula *ON* especifica como as tabelas devem ser *JOINed*, o que também pode filtrar o conjunto de resultados.



*FROM* é sempre avaliado primeiro, com uma exceção: quando a consulta não contém uma cláusula *FROM*. Na maioria dos bancos de dados, é possível consultar usando apenas uma cláusula *SELECT*, como visto em alguns exemplos deste livro. Uma *SELECT* pode retornar informações do sistema, como data e versão do banco de dados. Ele também pode aplicar matemática, data, texto e outras funções a constantes. Embora haja pouco uso para essas consultas em análises finais, elas são úteis para testar funções ou iterar rapidamente cálculos complicados.

Em seguida, a cláusula *WHERE* é avaliada para determinar quais registros devem ser incluídos em cálculos adicionais. Observe que *WHERE* está no início da ordem de avaliação e, portanto, não pode incluir os resultados de cálculos que ocorrem em uma etapa posterior.

*GROUP BY* é calculado em seguida, incluindo as agregações relacionadas, como `count`, `sum` e `avg`. Como você pode esperar, *GROUP BY* incluirá apenas os valores que existem nas *FROM* tabelas *JOINing* e filtragem na *WHERE*.

*HAVING* é avaliado em seguida. Como segue *GROUP BY*, *HAVING* pode realizar filtragem em valores agregados retornados por *GROUP BY*. A única outra maneira de filtrar por valores agregados é colocar a consulta em uma subconsulta e aplicar os filtros na consulta principal. Por exemplo, podemos querer encontrar todos os estados que têm pelo menos mil termos na tabela `legislators_terms`, e vamos ordenar por termos em ordem decrescente para uma boa medida:

```
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 a seguir. Curiosamente, como os agregados já foram calculados neste ponto, eles podem ser usados na definição da função da janela. Por exemplo, no conjunto de dados `legislators` do [Capítulo 4](#), poderíamos calcular os termos servidos por estado e os termos médios em 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
...     ...    ...
```

Agregados também podem ser usados na cláusula *OVER*, como na consulta a seguir que classifica os estados em ordem decrescente pelo número total de termos:

```
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 ponto, a cláusula *SELECT* é finalmente avaliada. Isso é um pouco contraintuitivo, pois agregações e funções de janela são digitadas na seção *SELECT* da consulta. No entanto, o banco de dados já cuidou dos cálculos e os resultados estão disponíveis para manipulação posterior ou para exibição como estão. Por exemplo, uma agregação pode ser colocada em uma instrução *CASE* e ter funções matemáticas, de data ou de texto aplicadas se o resultado da agregação for um desses tipos de dados.



Os agregadores *sum*, *count* e *avg* retornam valores numéricos. No entanto, as funções *min* e *max* retornam o mesmo tipo de dados que a entrada e usam a ordenação inerente desse tipo de dados. Por exemplo, *min* e *max* datas retornam as primeiras e últimas datas do calendário, enquanto *min* e *max* nos campos de texto, use a ordem alfabética para determinar o resultado.

Após *SELECT* é *DISTINCT*, se presente na consulta. Isso significa que todas as linhas são calculadas e, em seguida, ocorre a desduplicação.

*UNION* (ou *UNION ALL*) é executado em seguida. Até este ponto, cada consulta que compõe um *UNION* é avaliada de forma independente. Este estágio é quando os conjuntos de resultados são reunidos em um. Isso significa que as consultas podem realizar seus cálculos de maneiras muito diferentes ou de conjuntos de dados diferentes. Tudo *UNION* procura o mesmo número de colunas e para que essas colunas tenham tipos de dados compatíveis.

*ORDER BY* é quase a última etapa da avaliação. Isso significa que ele pode acessar qualquer um dos cálculos anteriores para classificar o conjunto de resultados. A única ressalva é que, se *DISTINCT* for usado, *ORDER BY* não poderá incluir nenhum campo que não seja retornado na cláusula *SELECT*. Caso contrário, é totalmente possível ordenar um conjunto de resultados por um campo que não aparece na consulta.

*LIMIT* e *OFFSET* são avaliados por último na sequência de execução da consulta. Isso garante que o subconjunto de resultados retornado terá resultados totalmente calculados conforme especificado por qualquer uma das outras cláusulas que estão na consulta. Isso também significa que *LIMIT* tem um uso limitado no controle da quantidade de trabalho que o banco de dados faz antes que os resultados sejam retornados a você. Isso talvez seja mais perceptível quando uma consulta contém um valor grande *OFFSET*. Para *OFFSET* em, digamos, três milhões de registros, o banco de dados ainda precisa calcular todo o conjunto de resultados, descobrir onde estão os três milionésimos mais um registro e, em seguida, retornar os registros especificados pelo *LIMIT*. Isso não significa que *LIMIT* não seja útil. A verificação de alguns resultados pode confirmar os cálculos sem sobrecarregar a rede ou sua máquina local com dados. Além disso, usar *LIMIT* o mais cedo possível em uma consulta, como em uma subconsulta, ainda pode reduzir drasticamente o trabalho exigido pelo banco de dados à medida que você desenvolve uma consulta mais complexa.

Agora que entendemos bem a ordem em que os bancos de dados avaliam as consultas e realizam os cálculos, veremos algumas opções para controlar essas operações no contexto de uma consulta maior e complexa: subconsultas, tabelas temporárias e CTEs.

## Subconsultas

*Subconsultas* são geralmente a primeira maneira de aprendermos a controlar a ordem de avaliação em SQL ou a realizar cálculos que não podem ser realizados em uma única consulta principal. Eles são versáteis e podem ajudar a organizar consultas longas em partes menores com propósitos distintos.

Uma subconsulta é colocada entre parênteses, uma notação que deve ser familiar da matemática, onde os parênteses também forçam a avaliação de alguma parte de uma equação antes do resto. Dentro dos parênteses está uma consulta autônoma que é avaliada antes da consulta externa principal. Assumindo que a subconsulta está na cláusula *FROM*, o conjunto de resultados pode ser consultado pelo código principal, assim como qualquer outra tabela. Já vimos muitos exemplos com subconsultas neste livro.

Uma exceção à natureza autônoma de uma subconsulta é 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, imagine que quiséssemos analisar a filiação partidária anterior para legisladores atualmente em exercício. Poderíamos encontrar o primeiro ano em que eles eram membros de um partido diferente e verificar o quanto comum isso é quando agrupados por seu partido atual. Na primeira subconsulta, encontramos os legisladores atualmente em exercício. Na segunda, subconsulta lateral, usamos os resultados da primeira subconsulta para retornar a primeira `term_start` onde a parte é diferente da parte 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

```

Isso acaba sendo bastante incomum . Apenas três legisladores atuais mudaram de partido, e nenhum partido teve mais trocadores do que outros partidos. Existem outras maneiras de retornar o mesmo resultado — por exemplo, alterando a consulta para um *JOIN* e movendo os critérios na 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. Na minha experiência, o uso de *LATERAL* é menos comum e, portanto, menos compreendido do que outras sintaxes, portanto, é bom reservá-lo para casos de uso que não podem ser resolvidos eficientemente de outra maneira.

As subconsultas permitem muita flexibilidade e controle sobre a ordem dos cálculos. No entanto, uma série complexa de cálculos no meio de uma consulta maior pode se tornar difícil de entender e manter. Outras vezes, o desempenho das subconsultas é muito lento ou a consulta não retorna resultados. Felizmente, o SQL tem algumas opções adicionais que podem ajudar nessas situações: tabelas temporárias e expressões de tabela comuns.

## Tabelas temporárias

Uma *tabela temporária* (*temp*) é criada de forma semelhante a qualquer outra tabela no banco de dados, mas com uma diferença fundamental: ela persiste apenas durante a sessão atual. As tabelas temporárias são úteis quando você está trabalhando com apenas uma pequena parte de uma tabela muito grande, pois as tabelas pequenas são muito mais rápidas de consultar. Eles também são úteis quando você deseja usar um resultado intermediário em várias consultas. Como a tabela temporária é uma tabela independente, ela pode ser consultada várias vezes na mesma sessão. Outra vez que eles são úteis é quando você está trabalhando em determinados bancos de dados, como Redshift ou Vertica, que particionam dados entre nós. *INSERIR* dados em uma tabela temporária pode alinhar o particionamento a outras tabelas que serão *JOIN* em uma consulta subsequente. Existem duas desvantagens principais nas tabelas temporárias. Primeiro, eles exigem privilégios de banco de dados para gravar dados, o que pode não ser permitido por motivos de segurança. Em segundo lugar, algumas ferramentas de BI, como Tableau e Metabase, permitem que apenas uma única instrução SQL crie um conjunto de dados,<sup>1</sup> enquanto uma tabela temporária requer pelo menos duas: a declaração de dados *CREATE* e *INSERT* na tabela temporária e a consulta usando a tabela temporária.

Para criar uma tabela temporária, use o comando *CREATE*, seguido da palavra-chave *TEMPORARY* e o nome que você deseja dar a ela. A tabela pode então ser definida e uma segunda instrução usada para preenchê-la, ou você pode usar *CREATE as SELECT* para criar e preencher em uma etapa. Por exemplo, você pode criar uma tabela temporária com os estados distintos para os quais existem legisladores:

```
CREATE temporary table temp_states
(
    state varchar primary key
)
;

INSERT into temp_states
SELECT distinct state
```

---

<sup>1</sup> No caso do Tableau, você pode contornar isso com a opção SQL inicial.

```
FROM legislators_terms
;
```

A primeira instrução cria a tabela, enquanto a segunda instrução preenche a tabela temporária com valores de uma consulta. Observe que, definindo a tabela primeiro, preciso especificar o tipo de dados para todas as colunas (neste caso, `varchar` da coluna `state`) e, opcionalmente, posso usar outros elementos de definição de tabela, como definir uma chave primária. Eu gosto de prefixar nomes de tabelas temporárias com “`temp_`” ou “`tmp_`” para me lembrar do fato de que estou usando uma tabela temporária na consulta principal, mas isso não é estritamente necessário.

A maneira mais rápida e fácil de gerar uma tabela temporária é o método *CREATE as SELECT*:

```
CREATE temporary table temp_states
as
SELECT distinct state
FROM legislators_terms
;
```

Nesse caso, o banco de dados decide automaticamente o tipo de dados com base nos dados retornados pela instrução *SELECT* e nenhuma chave primária é definida. A menos que você precise de um controle refinado por motivos de desempenho, esse segundo método servirá bem.

Como as tabelas temporárias são gravadas em disco, se você precisar repovoá-las durante uma sessão, será necessário *DROP* e recriar a tabela ou *TRUNCATE* os dados. Desconectar e reconectar ao banco de dados também funciona.

## **Expressões de Tabela Comuns (Common Table Expressions)**

Os CTEs são relativamente recém-chegados à linguagem SQL, tendo sido introduzidos em muitos dos principais bancos de dados apenas no início dos anos 2000. Eu escrevi SQL por anos sem eles, me contentando com subconsultas e tabelas temporárias. Devo dizer que, desde que os conheci há alguns anos, eles cresceram constantemente em mim.

Você pode pensar em uma *expressão de tabela comum* como uma subconsulta levantada e colocada no início da execução da consulta. Ele cria um conjunto de resultados temporário que pode ser usado em qualquer lugar na consulta subsequente. Uma consulta pode ter vários CTEs e os CTEs podem usar resultados de CTEs anteriores para realizar cálculos adicionais.

CTEs são particularmente úteis quando o resultado será usado várias vezes no restante da consulta. A alternativa, definir a mesma subconsulta várias vezes, é lenta (já que o banco de dados precisa executar a mesma consulta várias vezes) e propensa a erros. Esquecer de atualizar a lógica em cada subconsulta idêntica introduz erro no resultado final. Como os CTEs fazem parte de uma única consulta, eles não requerem nenhuma permissão especial de banco de dados. Eles também podem ser uma maneira útil de organizar o código em partes discretas e evitar subconsultas aninhadas extensas.

A principal desvantagem dos CTEs decorre do fato de serem definidos no início, separados de onde são usados.

Isso pode tornar uma consulta mais difícil de decifrar para outras quando a consulta for muito longa, pois é necessário rolar até o início para verificar a definição e depois voltar para onde o CTE é usado para entender o que está acontecendo. O bom uso dos comentários pode ajudar com isso. Um segundo desafio é que os CTEs dificultam a execução de seções de consultas longas. Para verificar resultados intermediários em uma consulta mais longa, é bastante fácil selecionar e executar apenas uma subconsulta em uma ferramenta de desenvolvimento de consulta. Se um CTE estiver envolvido, no entanto, todo o código ao redor deve ser comentado primeiro.

Para criar um CTE, usamos a palavra-chave *WITH* no início da consulta geral, seguida de um nome para o CTE e, em seguida, da consulta que o compõe entre parênteses. Por exemplo, poderíamos criar um CTE que calculasse o primeiro termo para cada legislador e, em seguida, usaria esse resultado em cálculos adicionais, como o cálculo de coorte apresentado 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 igual ao retornado pela consulta alternativa usando subconsultas vistas no [Capítulo 4](#). Vários CTEs podem ser usados na mesma consulta, separados por vírgulas:

```
WITH first_cte AS
(
    SELECT...
),
second_cte AS
(
    SELECT...
)
SELECT...
```

CTEs são uma maneira útil de controlar a ordem de avaliação, melhorar o desempenho em alguns casos e organizar seu código SQL. Eles são fáceis de usar quando você está familiarizado com a sintaxe e estão disponíveis na maioria dos principais bancos de dados. Muitas vezes, existem várias maneiras de realizar algo em SQL e, embora não sejam obrigatórios, os CTEs adicionam flexibilidade útil à sua caixa de ferramentas de habilidades SQL.

## agrupamento de conjuntos

Embora este próximo tópico não seja estritamente sobre o controle da ordem de avaliação, é uma maneira prática de evitar *UNIONs* e fazer com que o banco de dados faça todo o trabalho em uma única instrução de consulta. Dentro da cláusula *GROUP BY*, a sintaxe especial está disponível em muitos bancos de dados importantes que incluem **grouping sets**, **cube**, e **rollup** (embora o Redshift seja uma exceção e o MySQL tenha apenas **rollup**). Eles são úteis quando o conjunto de dados precisa conter subtotais para várias combinações de atributos.

Para exemplos nesta seção, usaremos um conjunto de dados de vendas de videogames que está [disponível no Kaggle](https://www.kaggle.com/datasets/gregorut/videogamesales) (<https://www.kaggle.com/datasets/gregorut/videogamesales>). Ele contém atributos para o nome de cada jogo, bem como a plataforma, ano, gênero e editora do jogo. Os números de vendas são fornecidos para a América do Norte, UE, Japão, Outros (o resto do mundo) e o total global. O nome da tabela é `videogame_sales`. [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 tabelavideogame\\_vendas](#)

Assim, no conjunto de dados do videogame, por exemplo, podemos querer agrregar `global_sales` por plataforma, gênero e editora como agregações independentes (em vez de apenas as combinações dos três campos que existem nos dados), mas a saída os resultados em um conjunto de consultas. Isso pode ser feito por *UNIONing* em conjunto três consultas. Observe que cada consulta deve conter pelo menos espaços reservados para todos os três campos de 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 pode ser obtido em uma consulta mais compacta usando `grouping sets`. Dentro da cláusula `GROUP BY`, `grouping sets` é seguido pela lista de agrupamentos a serem calculados. 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 `grouping sets` podem incluir espaços em branco, bem como 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 de lista que é apenas um par de parênteses. Também limparemos a saída substituindo “All” pelos itens nulos usando `coalesce`:

```
SELECT coalesce(platform,'All') as platform
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
300     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 editora, como os subtotais individuais recém-calculados, mais todas as combinações de plataforma e gênero, plataforma e editora e gênero e editora, poderíamos especificar todas essas combinações nos `grouping sets`. Ou podemos usar a sintaxe útil `cube`, que lida com 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 é a função `rollup`, que retorna um conjunto de dados que possui combinações determinadas pela ordenação dos campos entre parênteses, em vez de todas as combinações possíveis. Portanto, a consulta anterior com a seguinte cláusula:

```
GROUP BY rollup (platform, genre, publisher)
```

retorna agregações para as combinações de:

```
platform, genre, publisher  
platform, genre  
platform
```

Mas 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, pois resultam em menos linhas de código e menos varreduras das tabelas de banco de dados subjacentes. Certa vez, criei uma consulta de centenas de linhas usando `UNIONs` para gerar saída para um gráfico de site dinâmico que precisava ter todas as combinações possíveis de filtros pré-calculadas. A verificação da qualidade era uma tarefa enorme, e atualizá-la era ainda pior. Aproveitar `grouping sets` e CTEs poderia ter ajudado bastante a tornar o código mais compacto e fácil de escrever e manter.

## Gerenciando o tamanho do conjunto de dados e as preocupações com a privacidade

Após cuidarmos de trabalhar adequadamente a lógica em nosso SQL, organizar nosso código e torná-lo eficiente, muitas vezes nos deparamos com outro desafio: o tamanho do conjunto de resultados. O armazenamento de dados está cada vez mais barato, o que significa que as organizações estão armazenando conjuntos de dados cada vez maiores. O poder computacional também está sempre aumentando, o que nos permite processar esses dados da maneira sofisticada que vimos nos capítulos anteriores. No entanto, gargalos ainda ocorrem, seja em sistemas downstream, como ferramentas de BI, ou na largura de banda disponível para passar grandes conjuntos de dados entre sistemas. Além disso, a privacidade dos dados é uma grande preocupação que afeta a forma como lidamos com dados confidenciais. Por esses motivos, nesta seção, discutirei algumas maneiras de limitar o tamanho dos conjuntos de dados, bem como considerações sobre privacidade de dados.

### Amostragem com %, mod

Uma maneira de reduzir o tamanho de um conjunto de resultados é usar uma amostra dos dados de origem. *Amostragem* significa tomar apenas um subconjunto dos pontos de dados ou observações. Isso é apropriado quando o conjunto de dados é grande o suficiente e um subconjunto é representativo de toda a população. Muitas vezes, você pode experimentar o tráfego do site e ainda reter a maioria das informações úteis, por exemplo.

Há duas escolhas a serem feitas durante a amostragem. O primeiro é o tamanho da amostra que atinge 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. A segunda escolha é a entidade na qual realizar a amostragem. Podemos amostrar 1% das *visitas*, mas se o objetivo da análise for entender como os usuários navegam no site, a amostragem de 1% dos *visitantes* seria uma escolha melhor para preservar todos os pontos de dados para os usuários na amostra.

A maneira mais comum de amostra é filtrar os resultados da consulta na cláusula *WHERE* aplicando uma função a um identificador de nível de entidade. Muitos campos de ID são armazenados como números inteiros. Se este for o caso, fazer um módulo é uma maneira rápida de alcançar o resultado certo. O módulo é o resto do número inteiro quando um número é dividido por outro. Por exemplo, 10 dividido por 3 é igual a 3 com resto (módulo) de 1. SQL tem duas maneiras equivalentes de encontrar o módulo — com o sinal % e com a mod :

```
SELECT 123456 % 100 as mod_100;
mod_100
-----
56

SELECT mod(123456,100) as mod_100;
mod_100
-----
56
```

Ambos retornam a mesma resposta, 56, que também são os dois últimos dígitos do valor de entrada 123456. Para gerar uma amostra de 1% do conjunto de dados, coloque qualquer sintaxe na cláusula *WHERE* e defina-a como um inteiro — neste caso, 7:

```
SELECT user_id, ...
FROM table
WHERE user_id % 100 = 7
;
```

Um mod de 100 cria uma amostra de 1%, enquanto um mod de 1.000 cria uma amostra de 0,1% e um mod de 10 cria uma amostra de 10%. Embora a amostragem em múltiplos de 10 seja comum, não é necessária, e qualquer número inteiro funcionará.

A amostragem de identificadores alfanuméricos que incluem letras e números não é tão simples quanto a amostragem de identificadores puramente numéricos. As funções de análise de strings podem ser usadas para isolar apenas os primeiros ou os últimos caracteres, e os filtros podem ser aplicados a eles. Por exemplo, podemos amostrar apenas identificadores que terminam na letra “b” analisando o último caractere de uma string usando 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'
;
```



Ao fazer a amostragem, vale a pena validar se a função que você usa para gerar uma amostra cria uma amostragem aleatória ou quase aleatória dos dados. Em uma das minhas funções anteriores, descobrimos que certos tipos de usuários eram mais propensos a ter certas combinações dos dois últimos dígitos em seus IDs de usuário. Nesse caso, usar a `mod` para gerar uma amostra de 1% resultou em um viés perceptível nos resultados. Os identificadores alfanuméricos em particular geralmente têm padrões comuns no início ou no final da string que a criação de perfil de dados pode ajudar a identificar.

A amostragem é uma maneira fácil de reduzir o tamanho do conjunto de dados em ordens de magnitude. Ele pode acelerar os cálculos dentro de instruções SQL e permitir que o resultado final seja mais compacto, tornando mais rápido e fácil a transferência para outra ferramenta ou sistema. Às vezes, a perda de detalhes da amostragem não é aceitável, no entanto, outras técnicas são necessárias.

## Reduzindo a Dimensionalidade

O número de combinações distintas de atributos, ou *dimensionalidade*, impacta muito o número de registros em um conjunto de dados. Para entender isso, podemos fazer um simples experimento mental. Imagine que temos um campo com 10 valores distintos, e `count` o número de registros e `GROUP BY` nesse campo. A consulta retornará 10 resultados. Agora adicione um segundo campo, também com 10 valores distintos, `count` o número de registros e `GROUP BY` nos dois campos. A consulta retornará 100 resultados. Adicione um terceiro campo com 10 valores distintos e o resultado da consulta aumentará para 1.000 resultados.

Mesmo que nem todas as combinações dos três campos realmente existam na tabela consultada, fica claro que adicionar campos adicionais a uma consulta pode aumentar drasticamente o tamanho dos resultados.

Ao realizar a análise, muitas vezes podemos controlar o número de campos e filtrar os valores incluídos para obter uma saída gerenciável. No entanto, ao preparar conjuntos de dados para análise adicional em outras ferramentas, o objetivo geralmente é fornecer flexibilidade e, portanto, incluir muitos atributos e cálculos diferentes. Para reter o máximo de detalhes possível ao gerenciar o tamanho geral dos dados, podemos usar uma ou mais técnicas de agrupamento.

A granularidade de datas e horas costuma ser um local óbvio para procurar reduzir o tamanho dos dados. converse com seus stakeholders para determinar se os dados diários são necessários, por exemplo, ou se as agregações semanais ou mensais também funcionariam. Agrupar dados por mês e dia da semana pode ser uma solução para agregar dados e ainda fornecer visibilidade em padrões que diferem nos dias da semana e nos fins de semana. Restringir o período de retorno é sempre uma opção, mas isso pode restringir a exploração de tendências de longo prazo. Já vi equipes de dados fornecerem um conjunto de dados que se agrava a um nível mensal e abrange vários anos, enquanto um conjunto de dados complementar inclui os mesmos atributos, mas com dados diários ou mesmo horários para uma janela de tempo muito menor.

Os campos de texto são outro local para verificar possíveis economias de espaço. Diferenças na ortografia ou capitalização podem resultar em muito mais valores distintos do que são úteis. Aplicando funções de texto discutidas no Capítulo 5, como `lower`, `trim` ou `initcap`, padroniza valores e geralmente torna os dados mais úteis também para as partes interessadas. As instruções `REPLACE` ou `CASE` podem ser usadas para fazer ajustes mais sutis, como ajustar a ortografia ou alterar um nome que foi atualizado para um novo valor.

Às vezes, apenas alguns valores de uma lista mais longa são relevantes para análise, portanto, é eficaz reter os detalhes deles enquanto agrupa o restante. Tenho visto isso com frequência ao trabalhar com localizações geográficas. Existem cerca de duzentos países no mundo, mas muitas vezes apenas alguns têm clientes suficientes ou outros pontos de dados para fazer com que os relatórios sobre eles valham a pena individualmente. O conjunto de dados `legislators` usado no Capítulo 4 contém 59 valores para estado, que inclui os 50 estados mais os territórios dos EUA que possuem representantes. Podemos querer criar um conjunto de dados com detalhes para os cinco estados com as maiores populações (atualmente Califórnia, Texas, Flórida, Nova York e Pensilvânia) e, em seguida, agrupar o restante em uma categoria “outros” 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, abaixo de 59, o que representa uma diminuição significativa. Para tornar a lista mais dinâmica, podemos primeiro classificar os valores em uma subconsulta, neste caso pelos valores distintos de `id_bioguide` (ID do legislador), e depois retornar o valor `state` para os 5 primeiros e “Outro” 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 mudam nesta segunda lista. Se continuarmos a atualizar o conjunto de dados com novos pontos de dados, a consulta dinâmica garantirá que a saída sempre reflita os valores superiores atuais.

A dimensionalidade também pode ser reduzida transformando os dados em valores de sinalizador. Os sinalizadores geralmente são binários (ou seja, eles têm apenas dois valores). BOOLEAN TRUE e FALSE podem ser usados para codificar sinalizadores, assim como 1 e 0, “Yes” e “No” ou qualquer outro par de strings significativas. Os sinalizadores são úteis quando um valor de limite é importante, mas os detalhes além disso são menos interessantes. Por exemplo, podemos querer saber se um visitante do site concluiu ou não uma compra, mas os detalhes sobre o número exato de compras são menos importantes.

No conjunto de dados `legislators`, há 28 números distintos de mandatos atendidos pelos legisladores. Em vez do valor exato, no entanto, podemos querer incluir em nossa saída apenas se um legislador serviu pelo menos dois mandatos, o que podemos fazer transformando os valores detalhados em um sinalizador:

```
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
```

Cerca de duas vezes mais legisladores serviram pelo menos dois mandatos em comparação com aqueles com apenas um mandato. Quando combinado com outros campos em um conjunto de dados, esse tipo de transformação pode resultar em conjuntos de resultados muito menores.

Às vezes, um simples indicador de verdadeiro/falso ou de presença/ausência não é suficiente para capturar a nuance necessária. Nesse caso, os dados numéricos podem ser transformados em vários níveis para manter alguns detalhes adicionais. Isso é feito com uma instrução CASE, e o valor de retorno pode ser um número ou string.

Podemos querer incluir não apenas se um legislador cumpriu um segundo mandato, mas também outro indicador para aqueles 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 reduzimos 28 valores distintos para 3, mantendo a noção de único legisladores de mandato, aqueles que foram reeleitos e aqueles que são excepcionalmente bons em permanecer no cargo. Tais agrupamentos ou distinções ocorrem em muitos domínios. Tal como acontece com todas as transformações discutidas aqui, pode levar a algumas tentativas e erros para encontrar os limites exatos que são mais significativos para as partes interessadas. Encontrar o equilíbrio certo de detalhes e agregação pode diminuir bastante o tamanho do conjunto de dados e, portanto, geralmente acelera o tempo de entrega e o desempenho do aplicativo downstream.

## PII e privacidade de dados

A privacidade de dados é uma das questões mais importantes que os profissionais de dados enfrentam atualmente. Grandes conjuntos de dados com muitos atributos permitem análises mais robustas com insights e recomendações detalhadas. No entanto, quando o conjunto de dados é sobre indivíduos, precisamos estar atentos às dimensões ética e regulatória dos dados coletados e usados. Os regulamentos sobre a privacidade de pacientes, estudantes e clientes de serviços financeiros existem há muitos anos. As leis que regulam os direitos de privacidade de dados dos consumidores também entraram em vigor nos últimos anos. O Regulamento Geral de Proteção de Dados (GDPR) aprovado pela UE é provavelmente o mais conhecido. Outras regulamentações incluem a Lei de Privacidade do Consumidor da Califórnia (CCPA), os Princípios de Privacidade Australianos e a Lei Geral de Proteção de Dados do Brasil (LGPD).

Esses e outros regulamentos abrangem o manuseio, armazenamento e (em alguns casos) exclusão de *personally identifiable information (PII)*. Algumas categorias de PII são óbvias: nome, endereço, e-mail, data de nascimento e número do Seguro Social. PII também inclui indicadores de saúde, como frequência cardíaca, pressão arterial e diagnósticos médicos. Informações de localização, como coordenadas de GPS, também são consideradas PII, pois um pequeno número de localizações de GPS pode identificar exclusivamente um indivíduo. Por exemplo, leituras de GPS em minha casa e na escola de meus filhos podem identificar exclusivamente alguém em minha casa. Um terceiro ponto de GPS no meu escritório poderia me identificar de forma única. Como profissional de dados, vale a pena se familiarizar com o que esses regulamentos abrangem e discutir como eles afetam seu trabalho com os advogados de privacidade de sua organização, que terão as informações mais atualizadas.

Uma prática recomendada ao analisar dados que incluem PII é evitar incluir a própria PII nas saídas. Isso pode ser feito agregando dados, substituindo valores ou valores de hash.

Para a maioria das análises, o objetivo é encontrar tendências e padrões. Contar clientes e calcular a média de seu comportamento, em vez de incluir detalhes individuais na saída, geralmente é o objetivo. As agregações geralmente removem PII; no entanto, esteja ciente de que uma combinação de atributos que têm uma contagem de usuários de 1 pode estar vinculada a um indivíduo. Estes podem ser tratados como outliers e removidos do resultado para manter um maior grau de privacidade.

Se dados individuais forem necessários por algum motivo - para poder calcular usuários distintos em uma ferramenta downstream, por exemplo - podemos substituir valores problemáticos por valores alternativos aleatórios que mantém a exclusividade. A função `row_number` pode ser usada para atribuir um novo valor a cada indivíduo em uma tabela:

```
SELECT email
, row_number() over (order by ...)
FROM users
;
```

O desafio neste caso é encontrar um campo para colocar no *ORDER BY* que torne a ordenação suficientemente aleatória para que possamos considerar o identificador de usuário resultante anonimizado.

Valores de hash é outra opção. O hashing recebe 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, tornando esta uma boa opção para manter a exclusividade enquanto obscurece valores sensíveis. A função `md5` pode ser usada para gerar um valor com hash:

```
SELECT md5('my info');

md5
-----
0fb1d3f29f5dd1b7cabbad56cf043d1a
```



A função `md5` hash dos valores de entrada, mas não os criptografa e, portanto, pode ser revertida para obter o valor original. Para dados altamente confidenciais, você deve trabalhar com um administrador de banco de dados para realmente criptografar os dados.

Evitar PII na saída de suas consultas SQL é sempre a melhor opção, se possível, pois você evita proliferar em outros sistemas ou arquivos. Substituir ou mascarar os valores é a segunda melhor opção. Você também pode explorar métodos seguros para compartilhar dados, como desenvolver um pipeline de dados seguro diretamente entre um banco de dados e um sistema de e-mail para evitar gravar endereços de e-mail em arquivos, por exemplo. Com cuidado e parceria com colegas técnicos e jurídicos, é possível obter análises de alta qualidade e, ao mesmo tempo, preservar a privacidade dos indivíduos.

## Conclusão

Em torno de cada análise, há várias decisões a serem tomadas em torno da organização do código, gerenciamento da complexidade, otimização do desempenho da consulta e proteção da privacidade na saída. Neste capítulo, discutimos várias opções e estratégias e a sintaxe SQL especial que pode ajudar nessas tarefas. Tente não ficar sobrecarregado com todas essas opções ou se preocupar que, sem o domínio desses tópicos, você não pode ser um analista de dados ou cientista de dados eficiente. Nem todas as técnicas são necessárias em todas as análises, e muitas vezes há outras maneiras de fazer o trabalho. Quanto mais tempo você passar analisando dados com SQL, maior a probabilidade de se deparar com situações em que uma ou mais dessas técnicas sejam úteis.

## CAPÍTULO 9

# Conclusão

Ao longo do livro, vimos como o SQL é uma linguagem flexível e poderosa para uma série de tarefas de análise de dados. De perfis de dados a séries temporais, análise de texto e detecção de anomalias, o SQL pode atender a vários requisitos comuns. Técnicas e funções também podem ser combinadas em qualquer instrução SQL para realizar análises de experimentos e construir conjuntos de dados complexos. Embora o SQL não possa cumprir todas as metas de análise, ele se encaixa bem no ecossistema de ferramentas de análise.

Neste capítulo final, discutirei alguns tipos adicionais de análise e mostrarei como várias técnicas SQL abordadas no livro podem ser combinadas para realizá-las. Em seguida, encerrarei com alguns recursos que você pode usar para continuar sua jornada de domínio da análise de dados ou para aprofundar tópicos específicos.

## Análise de Funil

Um funil consiste em uma série de etapas que devem ser concluídas para atingir uma meta definida. O objetivo pode ser registrar-se em um serviço, concluir uma compra ou obter um certificado de conclusão de curso. As etapas em um funil de compra do site, por exemplo, podem incluir clicar no botão “Adicionar ao carrinho”, preencher as informações de envio, inserir um cartão de crédito e, finalmente, clicar no botão “Fazer pedido”.

A análise de funil combina elementos de análise de séries temporais, discutidos no [Capítulo 3](#), e análise de coorte, discutidos no [Capítulo 4](#). Os dados para análise de funil vêm de uma série temporal de eventos, embora, neste caso, esses eventos correspondam a ações distintas do mundo real, em vez de serem repetições do mesmo evento. Medir a retenção passo a passo é um dos principais objetivos da análise de funil, embora nesse contexto geralmente usamos o termo *conversão*. Normalmente, as entidades desistem ao longo das etapas do processo, e um gráfico de seu número em cada estágio acaba parecendo um funil doméstico – daí o nome.

Esse tipo de análise é usado para identificar áreas de atrito, dificuldade ou confusão. As etapas nas quais um grande número de usuários desistem, ou que muitos não conseguem concluir, fornecem informações sobre oportunidades de otimização. Por exemplo, um processo de checkout que solicite informações do cartão de crédito antes de mostrar o valor total, incluindo o frete, pode desanistar alguns possíveis compradores. Mostrar o total antes desta etapa pode incentivar mais compras concluídas. Tais mudanças são frequentemente objetos de experimentos, discutidos no [Capítulo 7](#). Os funis também podem ser monitorados para detectar eventos externos inesperados. Por exemplo, mudanças nas taxas de conclusão podem corresponder a boas (ou más) relações públicas ou a uma mudança nos preços ou nas táticas de um concorrente.

A primeira etapa em uma análise de funil é descobrir a população base de todos os usuários, clientes ou outras entidades qualificadas para entrar no processo. Em seguida, monte o conjunto de dados de conclusão para cada etapa de interesse, incluindo o objetivo final. Muitas vezes, isso inclui um ou mais *LEFT JOINs* para incluir toda a população base, juntamente com aqueles que concluíram cada etapa. Em seguida, `count` os usuários em cada etapa e divida essas contagens por etapas pela `count`. Há duas maneiras de configurar as consultas, dependendo se todas as etapas são necessárias.

Quando todas as etapas do funil forem necessárias — ou se você quiser incluir apenas usuários que concluíram todas as etapas — *LEFT JOIN* cada tabela para a tabela anterior:

```
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 poderem pular uma etapa, ou se você quiser permitir essa possibilidade, *LEFT JOIN* cada tabela para aquela que contém a população completa e calcule a participação 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 vale a pena prestar atenção e adaptar ao contexto específico. Considere incluir caixas de tempo, para incluir apenas os usuários que concluem uma ação dentro de um período de tempo específico, se os usuários puderem entrar novamente no funil após uma longa ausência.

As análises de funil também podem incluir dimensões adicionais, como coorte ou outros atributos de entidade, para facilitar comparações e gerar hipóteses adicionais sobre por que um funil está ou não funcionando bem.

## Churn, Lapse e outras definições de saída

O tópico churn surgiu no [Capítulo 4](#), já que churn é essencialmente o oposto de retenção. Muitas vezes, as organizações querem ou precisam criar uma definição específica de churn para medi-la diretamente. Em alguns casos, há uma data de término contratualmente definida, como no software B2B. Mas muitas vezes o churn é um conceito mais confuso, e uma definição baseada no tempo é mais apropriada. Mesmo quando há uma data de término contratual, medir quando um cliente deixa de usar um produto pode ser um sinal de alerta antecipado de um cancelamento iminente do contrato. As definições de churn também podem ser aplicadas a determinados produtos ou recursos, mesmo quando o cliente não sai totalmente da organização.

Uma métrica de desligamento baseado em tempo conta os clientes como desligados quando eles não compraram ou interagiram com um produto por um período de tempo, geralmente variando de 30 dias a até um ano. A duração exata depende muito do tipo de negócio e dos padrões de uso típicos. Para chegar a uma boa definição de churn, você pode usar a análise de lacunas para encontrar períodos típicos entre compras ou uso. Para fazer a análise de lacunas, você precisará de uma série temporal de ações ou eventos, a função `lag` e alguns cálculos de data.

Como exemplo, podemos calcular as lacunas típicas entre os mandatos dos deputados, usando o conjunto de dados dos legisladores apresentado no [Capítulo 4](#). Ignoraremos o fato de que os políticos são muitas vezes eleitos fora do cargo em vez de optarem por sair, pois, caso contrário, esse conjunto de dados tem a estrutura certa para esse tipo de análise. Primeiro vamos encontrar a diferença média. Para fazer isso, criamos uma subconsulta que calcula o intervalo entre a `data_inicial` e a `data_inicial` para cada legislador para cada mandato e, em seguida, encontramos o valor médio na consulta externa. A `data_inicial` pode ser encontrada usando a função `lag`, e o intervalo como intervalo de tempo é calculado 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

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 suportado em seu banco de dados, a `extract` pode ser usada como alternativa. (Consulte o Capítulo 3 para obter uma explicação e exemplos.) A saída pode ser plotada, como na Figura 9-1. Como há uma longa cauda de meses, esse gráfico é ampliado para mostrar o intervalo no qual a maioria das lacunas se encontra. O intervalo mais comum é de 24 meses, mas também há

várias centenas de instâncias por mês até 32 meses. Há outro pequeno aumento para mais de 100 aos 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 tenha sido reeleito dentro dessa janela “desapareceu”.

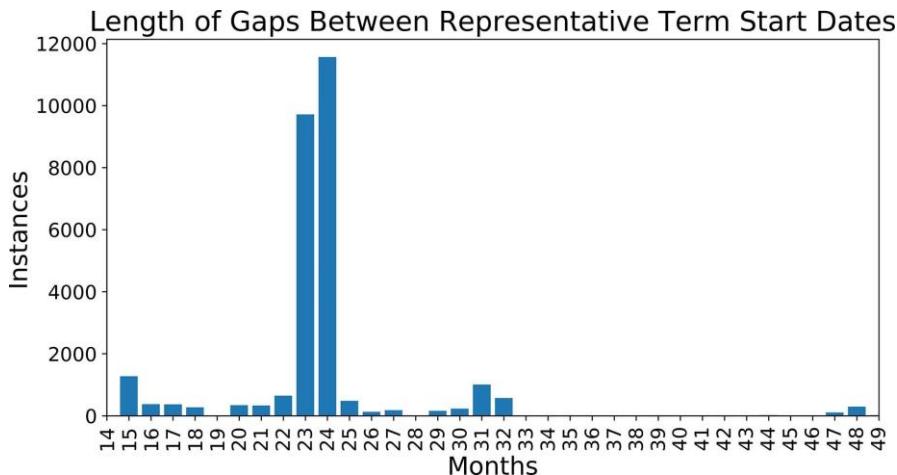


Figura 9-1. Distribuição da duração do intervalo entre as datas de início dos mandatos representativos, mostrando o intervalo de 10 a 59 meses

Uma vez que você tenha um limite definido para churn, você pode monitorar a base de clientes com uma análise de “tempo desde a última vez”. Isso pode se referir à última compra, último pagamento, última vez que um aplicativo foi aberto ou qualquer métrica baseada em tempo relevante para a organização. Para este cálculo, você precisa de um conjunto de dados que tenha a data ou hora mais recente para cada cliente. Se começar com uma série temporal, primeiro localize o carimbo de data/hora mais recente para cada cliente em uma subconsulta. Em seguida, aplique a matemática de data para encontrar o tempo decorrido entre essa data e a data atual, ou a data mais recente no conjunto de dados, se tiver decorrido algum tempo desde que os dados foram reunidos.

Por exemplo, podemos encontrar a distribuição de anos desde a última eleição na tabela `legislators_terms`. Na subconsulta, calcule a última data de início usando a função `max` e encontre o tempo decorrido desde então usando a função `age`. Nesse caso, são usados os dados máximos no conjunto de dados, 5 de maio de 2019. Em um conjunto de dados com dados atualizados, substitua `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 “caducado”, que é frequentemente usado como um estágio intermediário entre clientes totalmente ativos e clientes inativos e pode, alternativamente, ser chamado de “inativo”. Um cliente inativo pode estar em maior risco de desistência porque não o vemos há algum tempo, mas ainda temos uma boa probabilidade de retornar com base em nossa experiência anterior. Nos serviços ao consumidor, vi períodos de cobertura “caducados” de 7 a 30 dias, com “churned” definido como um cliente que não usa o serviço há mais de 30 dias. As empresas costumam experimentar a reativação de usuários inativos, usando táticas que vão desde o e-mail até a divulgação da equipe de suporte. Os clientes em cada estado podem ser definidos primeiro encontrando seu “tempo desde o último” conforme acima e, em seguida, marcando-os com uma declaração CASE usando o número apropriado de dias ou meses. Por exemplo, podemos agrupar os representantes de acordo com há quanto tempo 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

```

Este conjunto de dados contém mais de duzentos anos de mandatos de legisladores, então é claro que muitas das pessoas incluídas morreram, e alguns ainda estão vivos, mas estão aposentados. No contexto de um negócio, esperamos que nossos clientes inativos não ultrapassem nossos clientes atuais por uma margem tão ampla, e gostaríamos de saber mais sobre os clientes inativos.

A maioria das organizações está muito preocupada com o churn, já que os clientes geralmente são mais caros para adquirir do que reter. Para saber mais sobre os clientes em qualquer status ou sobre o intervalo de tempo desde a última visualização, essas análises podem ser divididas ainda mais por qualquer um dos atributos do cliente disponíveis no conjunto de dados.

## Análise da cesta

Tenho três filhos e, quando vou ao supermercado, minha cesta (ou, mais frequentemente, meu carrinho de compras) se enche rapidamente com itens de mercearia para alimentá-los durante a semana. Leite, ovos e pão geralmente estão lá, mas outros itens podem mudar dependendo do produto da estação, se as crianças estão na escola ou no intervalo, ou se estamos planejando cozinhar uma refeição especial. A análise de cestas leva o nome da prática de analisar os produtos que os consumidores compram juntos para encontrar padrões que possam ser usados para marketing, posicionamento da loja ou outras decisões estratégicas. O objetivo de uma análise de cesta pode ser encontrar grupos de itens comprados juntos. Também pode ser enquadrado em torno de um determinado produto: quando alguém compra sorvete, o que mais compra?

Embora a análise da cesta tenha sido originalmente estruturada em torno de itens comprados juntos em uma única transação, o conceito pode ser estendido de várias maneiras. Um varejista ou uma loja de comércio eletrônico pode estar interessado na cesta de itens que um cliente compra ao longo da vida. Serviços e uso de recursos de produtos também podem ser analisados dessa maneira. Os serviços normalmente adquiridos em conjunto podem ser agrupados em uma nova oferta, como quando os sites de viagens oferecem promoções se um voo, hotel e aluguel de carro forem reservados juntos. Os recursos do produto que são usados juntos podem ser colocados na mesma janela de navegação ou usados para fazer sugestões de onde ir a seguir em um aplicativo. A análise de cestas também pode ser usada para identificar as personas ou segmentos das partes interessadas, que são usadas em outros tipos de análise.

Para encontrar as cestas mais comuns, usando todos os itens de uma cesta, podemos usar a função `string_agg` (ou uma análoga, dependendo do tipo de banco de dados — veja o Capítulo 5).

Por exemplo, imagine que temos uma tabela de `purchases` que possui uma linha para cada `product` por um `customer_id`. Primeiro, use a função `string_agg` para encontrar a lista de produtos adquiridos por cada cliente em uma subconsulta. Então *GROUP BY* nesta lista e `count` o número de clientes:

```
SELECT products
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, auto`JOIN` a tabela `purchases` a si mesma, *JOINing* no `customer_id`. A segunda *JOIN* resolve o problema de entradas duplicadas que diferem apenas em sua ordem. Por exemplo, imagine um cliente que comprou maçãs e bananas – sem essa cláusula, o conjunto de resultados incluiria “apples, bananas” e “bananas, apples”. A cláusula `b.product > a.product` garante que apenas uma dessas variações seja incluída e também filtra os resultados nos quais um produto corresponde a si mesmo:

```
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
;
```

Isso pode ser estendido para incluir três ou mais produtos adicionando *JOINs* adicionais. Para incluir cestas que contenham apenas um item, altere o *JOIN* para *LEFT JOIN*.

Existem alguns desafios comuns ao executar uma análise de cesta. O primeiro é o desempenho, principalmente quando há um grande catálogo de produtos, serviços ou recursos. Os cálculos resultantes podem se tornar lentos no banco de dados, principalmente quando o objetivo é encontrar grupos de três ou mais itens e, portanto, o SQL contém três ou mais auto-*JOINs*. Considere filtrar as tabelas com cláusulas *WHERE* para remover

itens comprados com pouca frequência antes de executar os *JOINS*. Outro desafio ocorre quando alguns itens são tão comuns que inundam todas as outras combinações. Por exemplo, o leite é comprado com tanta frequência que os grupos com ele e qualquer outro item encabeçam a lista de combinações. Os resultados da consulta, embora precisos, ainda podem não ter sentido prático. Nesse caso, considere remover completamente os itens mais comuns, novamente com uma cláusula *WHERE*, antes de executar os *JOINS*. Isso deve ter o benefício adicional de melhorar o desempenho da consulta, tornando o conjunto de dados menor.

Um desafio final com a análise da cesta é a profecia auto-realizável. Os itens que aparecem juntos em uma análise de cesta podem ser comercializados juntos, aumentando a frequência com que são comprados juntos. Isso pode fortalecer a possibilidade de comercializá-los ainda mais, levando a mais compras conjuntas e assim por diante. Produtos que combinam ainda melhor podem nunca ter uma chance, simplesmente porque não apareceram na análise original e se tornaram candidatos à promoção. A famosa [correlação cerveja e fraldas](#) é apenas um exemplo disso. Várias técnicas de aprendizado de máquina e grandes empresas online tentaram resolver esse problema, e há muitas direções interessantes para análise nessa área ainda a serem desenvolvidas.

## Recursos

A análise de dados como profissão (ou mesmo como hobby!) requer uma combinação de proficiência técnica, conhecimento de domínio, curiosidade e habilidades de comunicação. Pensei em compartilhar alguns dos meus recursos favoritos para que você possa aproveitá-los enquanto continua sua jornada, tanto para aprender mais quanto para praticar suas novas habilidades em conjuntos de dados reais.

### Livros e blogs

Embora este livro suponha um conhecimento prático de SQL, bons recursos para o básico ou para uma atualização são:

- Forta, Ben. *Sams Aprenda SQL em 10 minutos por dia*. 5<sup>a</sup> edição. Hoboken, NJ: Sams, 2020.
- A empresa de software Mode oferece um [tutorial SQL](#) com uma interface de consulta interativa, útil para praticar suas habilidades.

Não existe um estilo SQL universalmente aceito, mas você pode encontrar o [estilo SQL Guia](#) e o [Guia de Estilo SQL Moderno](#) útil. Observe que seus estilos não correspondem exatamente aos usados neste livro ou entre si. Acredito que usar um estilo que seja consistente consigo mesmo e legível é a consideração mais importante.

Sua abordagem de análise e comunicação dos resultados muitas vezes pode ser tão importante quanto o código que você escreve. Dois bons livros para aprimorar ambos os aspectos são:

- Hubbard, Douglas W. *How to Measure Anything: Finding the Value of "Intangibles" in Business*. 2<sup>a</sup> edição. Hoboken, NJ: Wiley, 2010.
- Kahneman, Daniel. *Pensando, Rápido e Lento*. Nova York: Farrar, Straus e Giroux, 2011.

O blog [Towards Data Science](#) é uma ótima fonte de artigos sobre muitos tópicos de análise. Embora muitos dos posts ali se concentrem no Python como linguagem de programação, abordagens e técnicas podem ser adaptadas ao SQL.

Para uma visão divertida sobre correlação versus causalidade, veja [Spurious](#), de Tyler Vigen. [Correlações](#).

Expressões regulares podem ser complicadas. Se você deseja aumentar sua compreensão ou resolver casos complexos não abordados neste livro, um bom recurso é:

- Forta, Ben. *Aprendendo Expressões Regulares*. Boston: Addison-Wesley, 2018.

Testes randomizados têm uma longa história e atingem muitos campos das ciências naturais e sociais. Em comparação com as estatísticas, no entanto, a análise de experimentos online ainda é relativamente nova. Muitos textos clássicos de estatística dão uma boa introdução, mas discutem problemas em que o tamanho da amostra é muito pequeno, então eles não abordam muitas das oportunidades e desafios únicos dos testes online. Alguns bons livros que discutem experimentos online são:

- Georgiev, Georgi Z. *Statistical Methods in Online A/B Testing*. Sofia, Bulgária: autopublicado, 2019.
- Kohavi, Ron, Diane Tang e Ya Xu. *Experimentos controlados on-line confiáveis: um guia prático para testes A/B*. Cambridge, Reino Unido: Cambridge University Press, 2020.

[Ferramentas A/B Incríveis de Evan Miller](#) tem calculadoras para experimentos de resultados binários e contínuos, bem como vários outros testes que podem ser úteis para projetos de experimentos além do escopo deste livro.

## Conjuntos de dados

A melhor maneira de aprender e melhorar suas habilidades de SQL é colocá-las em uso em dados reais. Se você está empregado e tem acesso a um banco de dados em sua organização, esse é um bom lugar para começar, pois provavelmente já tem contexto sobre como os dados são produzidos e o que eles significam.

No entanto, existem muitos conjuntos de dados públicos interessantes que você pode analisar, e eles abrangem uma ampla variedade de tópicos. Listados abaixo estão alguns bons lugares para começar a procurar conjuntos de dados interessantes:

- **Dados são plurais** é um boletim informativo de conjuntos de dados novos e interessantes, e o [arquivo Data Is Plu-ral](#) é um tesouro pesquisável de conjuntos de dados.
- **Cinco Trinta e Oito** é um site de jornalismo que cobre política, esportes e ciência através de uma lente de dados. Os conjuntos de dados por trás das histórias estão no [FiveThirtyEight Site do GitHub](#).
- **Gapminder** é uma fundação sueca que publica dados anuais para muitos indicadores de desenvolvimento humano e econômico, incluindo muitos provenientes do Banco Mundial.
- As Nações Unidas publicam uma série de estatísticas. O Departamento de Assuntos Econômicos e Sociais da ONU produz dados sobre a [dinâmica populacional](#) em um formato relativamente fácil de usar.
- Kaggle hospeda competições de análise de dados e possui uma [biblioteca de conjuntos de dados](#) que podem ser baixados e analisados mesmo fora das competições formais.
- Muitos governos em todos os níveis, do nacional ao local, adotaram o movimento de dados abertos e publicam várias estatísticas. [Data.gov](#) mantém uma lista de sites nos Estados Unidos e em todo o mundo que é um bom ponto de partida.

## Finais

Espero que você tenha achado úteis as técnicas e o código deste livro. Acredito que é importante ter uma boa base nas ferramentas que você está usando, e existem muitas funções e expressões SQL úteis que podem tornar suas análises mais rápidas e precisas. Desenvolver grandes habilidades de análise não é apenas aprender as últimas técnicas ou linguagem sofisticadas, no entanto. A grande análise vem de fazer boas perguntas; dedicar tempo para entender os dados e o domínio; aplicar técnicas de análise apropriadas para obter respostas confiáveis e de alta qualidade; e, finalmente, comunicar os resultados ao seu público de uma forma que seja relevante e apoie a tomada de decisões. Mesmo depois de quase 20 anos trabalhando com SQL, ainda me empolgo em encontrar novas maneiras de aplicá-lo, novos conjuntos de dados para aplicá-lo e todos os insights do mundo esperando pacientemente para serem descobertos.