O'REILLY®

bhv®

# SQL для анализа данных

## Расширенные методы преобразования данных для аналитики

Кэти Танимура

# SQL for Data Analysis

## *Advanced Techniques for Transforming Data into Insights*

*Cathy Tanimura*

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY

Кэти Танимура

# SQL для анализа данных

## Расширенные методы преобразования данных для аналитики

Санкт-Петербург
«БХВ-Петербург»
2024

УДК 004.43
ББК 32.973.26-018.1
Т18

**Танимура К.**

Т18 SQL для анализа данных: Пер. с англ. — СПб.: БХВ-Петербург, 2024. — 384 с.: ил.

ISBN 978-5-9775-0958-9

Рассказывается о возможностях SQL применительно к анализу данных. Сравниваются различные типы баз данных, описаны методы подготовки данных для анализа. Рассказано о типах данных, структуре SQL-запросов, профилировнии, структурировании и очистке данных. Описаны методы анализа временных рядов, трендов, приведены примеры анализа данных с учетом сезонности. Отдельные главы посвящены когортному анализу, текстовому анализу, выявлению и обработке аномалий, анализу результатов экспериментов и A/B-тестирования. Описано создание сложных наборов данных, комбинирование методов анализа. Приведены практические примеры анализа воронки продаж и потребительской корзины.

*Для аналитиков, исследователей и специалистов по обработке данных*

УДК 004.43
ББК 32.973.26-018.1

**Группа подготовки издания:**

| | |
|---|---|
| Руководитель проекта | *Павел Шалин* |
| Зав. редакцией | *Людмила Гауль* |
| Перевод с английского | *Игоря Донченко* |
| Компьютерная верстка | *Натальи Смирновой* |
| Оформление обложки | *Зои Канторович* |



Подписано в печать 05.07.23.
Формат 70×100$^{1}/_{16}$. Печать офсетная. Усл. печ. л. 30,96.
Тираж 1500 экз. Заказ № 7204.
"БХВ-Петербург", 191036, Санкт-Петербург, Гончарная ул., 20.

Отпечатано с готового оригинал-макета
ООО "Принт-М", 142300, М.О., г. Чехов, ул. Полиграфистов, д. 1

ISBN 978-1-492-08878-3 (англ.)
ISBN 978-5-9775-0958-9 (рус.)

# Содержание

# Предисловие

В последние 20 лет я потратила немало рабочего времени, обрабатывая данные с помощью SQL. Я работала в основном в технологических компаниях из разных потребительских и межкорпоративных (B2B) отраслей. За это время объемы данных резко возросли, а технологии, которыми я пользуюсь, постоянно совершенствуются. Базы данных стали быстрее, чем когда-либо, а инструменты для отчетности и визуализации, используемые для представления информации о данных, стали еще более мощными. Но одно остается неизменным — это то, что SQL по-прежнему играет ключевую роль в моем инструментарии.

Я помню, когда впервые познакомилась с SQL. Я начинала свою карьеру в сфере финансов, где все работают с электронными таблицами (spreadsheet), и довольно хорошо научилась писать формулы и запомнила все сочетания клавиш. Однажды я даже перебрала на клавиатуре все клавиши, удерживая <Ctrl> и <Alt>, — просто чтобы посмотреть, что произойдет (а затем сделала шпаргалку для своих коллег). Это было сделано отчасти для развлечения и отчасти для выживания: чем быстрее я обрабатывала свои электронные таблицы, тем выше была вероятность, что я смогу закончить работу до полуночи, чтобы пойти домой и немного поспать. Навыки работы с электронными таблицами помогли мне получить мою следующую должность в стартапе, где я впервые познакомилась с базами данных и SQL.

Моя следующая работа, в частности, заключалась в обработке электронных таблиц с данными об инвентаризации, и эти наборы данных иногда состояли из десятков тысяч строк. В то время это были «большие данные» (big data), по крайней мере, для меня. Я могла успеть выпить чашечку кофе или даже пообедать, пока мой компьютер был занят выполнением волшебной функции VLOOKUP в Excel. Однажды мой менеджер ушел в отпуск и попросил меня заняться хранилищем данных, которое он создал на своем ноутбуке с помощью Microsoft Access. Обновление данных включало в себя несколько шагов: выполнение SQL-запросов, загрузка CSV-файлов в базу данных, получение отчетов в виде электронных таблиц. После первой успешной загрузки я захотела разобраться, как именно это работает, и попросила разработчиков объяснить мне, как изменять SQL-запросы.

Я увлеклась, и даже когда подумывала, что может быть сменить направление деятельности, я все равно возвращалась к работе с данными. Обработка данных, поиск ответов на вопросы, помощь коллегам, чтобы они могли работать лучше и эффек-

тивнее, а также изучение бизнес-процессов и окружающего мира через наборы данных — все это никогда не переставало приносить удовольствие и восторг.

Когда я начинала изучать SQL, было не так много доступных обучающих ресурсов. У меня была книга по основам синтаксиса, которую я прочла за ночь, а училась в основном методом проб и ошибок. Тогда я делала запросы напрямую к работающим производственным базам данных, и это не раз приводило к остановке вебсайтов из-за моих чрезмерно амбициозных (или, скорее всего, просто плохо написанных) запросов SQL. К счастью, с годами мои навыки улучшились, и я научилась работать с данными в таблицах и получать запрашиваемые выходные данные, решая различные технические и логические задачи с помощью правильных SQL-запросов. В итоге я стала заниматься проектированием и созданием общего хранилища данных, чтобы собирать данные из разных источников и избегать выхода из строя критически важных производственных баз данных. Я многое узнала о том, когда и как нужно обрабатывать данные перед написанием SQL-запросов, а когда надо оставлять данные в необработанном виде.

После общения с теми, кто начинал работать с данными примерно в то же время, что и я, стало ясно, что мы в основном учились одними и теми же способами. У некоторых счастливчиков были коллеги, которые могли поделиться с ними опытом. Большинство статей по SQL либо являются вводными и базовыми (и они безусловно нужны!), либо предназначены для разработчиков баз данных. И существует всего несколько ресурсов для опытных пользователей SQL, которые занимаются именно аналитической работой. Как правило, такими знаниями владеют только отдельные личности или небольшие команды. Цель этой книги — изменить такое положение дел и поделиться с практикующими специалистами знаниями о том, как можно решать общие задачи анализа данных с помощью SQL. И я надеюсь, что это вдохновит вас на новые исследования данных с помощью техник, которые вы, возможно, никогда раньше не использовали.

## Условные обозначения

В этой книге используются следующие типографские обозначения:

*Курсивный шрифт*

Курсивом выделены новые термины и ключевые слова.

**Полужирный шрифт**

Это URL-адреса, адреса электронной почты, имена файлов и их расширения.

`Моноширинный шрифт`

Используется для SQL-кода, а также при упоминании фрагментов кода внутри обычного текста, например операторов, функций и таблиц.

*`Моноширинный курсивный шрифт`*

Выделяет текст, который нужно заменить пользовательскими значениями или значениями, подходящими по контексту.

Эта иконка означает подсказку или предложение.

Эта иконка означает общее примечание.

Эта иконка указывает на предупреждение или предостережение.

## Использование примеров кода

Дополнительные материалы (примеры кода, наборы данных и т. д.) доступны для скачивания в репозитории GitHub: **https://github.com/cathytanimura/sql_book**.

Если у вас возникли технические вопросы или проблемы с использованием примеров кода, пожалуйста, отправьте письмо по адресу **bookquestions@oreilly.com**.

Эта книга написана для того, чтобы помочь вам в работе. Вы можете использовать в своих программах и документации примеры кода, приведенные в этой книге. Вам не нужно обращаться к нам за разрешением, если только вы не воспроизводите значительную часть кода. Например, написание программы, в которой используются несколько фрагментов кода из этой книги, не требует разрешения. Продажа или распространение примеров из книг O'Reilly требует разрешения. Ссылка на эту книгу с приведением примера кода не требует разрешения. Включение значительного количества примеров кода из этой книги в документацию вашего продукта требует разрешения.

Если вы считаете, что использование примеров кода выходит за рамки добросовестного использования или указанного выше разрешения, не стесняйтесь обращаться к нам по адресу **permissions@oreilly.com**.

## Благодарности

Эта книга была бы невозможна без помощи сотрудников компании O'Reilly. Энди Куон (Andy Kwan) привлек меня к этому проекту. Амелия Блевинс (Amelia Blevins) и Шира Эванс (Shira Evans) направляли меня в процессе работы и давали полезные советы. Кристен Браун (Kristen Brown) контролировала процесс выпуска книги. Артур Джонсон (Arthur Johnson) улучшил качество и понятность текста и ненароком заставил меня глубже изучить ключевые слова SQL.

Мои коллеги на протяжении многих лет играли важную роль в развитии моих навыков SQL, и я благодарна им за их помощь, советы и совместный код, а также за время, потраченное на мозговой штурм при поиске решения некоторых проблем анализа. Шэрон Лин (Sharon Lin) открыла мне глаза на регулярные выражения. Элайза Гордон (Elyse Gordon) дала мне множество советов по написанию книг. Дэйв Хох (Dave Hoch) и наши беседы об анализе экспериментов помогли мне с написанием *главы 7*. Дэн, Джим и Стю из Star Chamber долгое время были моими любимыми собеседниками. Я также благодарна всем коллегам, которые на протяжении многих лет задавали мне трудные вопросы, а получив на них ответы, задавали следующие, еще более трудные. Я хотела бы поблагодарить моего мужа Рика, сына Ши, дочерей Лили и Фиону, а также маму Джанет за их любовь, поддержку и, самое главное, за подаренное время для работы над этой книгой. Эми, Халле, Джесси и Дэн из Slack поддерживали мой рассудок и смешили меня на протяжении месяцев написания книги и локдауна во время пандемии.

# ГЛАВА 1
# Анализ с помощью SQL

Если вы читаете эту книгу, то, скорее всего, вас интересует анализ данных и использование языка SQL для выполнения такого анализа. Вы, может быть, уже занимались анализом данных, но не знакомы с SQL, или, возможно, у вас есть опыт работы с SQL, но вы мало знакомы с анализом данных. Или вы новичок и в том, и в том. Какой бы ни была ваша стартовая позиция, эта глава закладывает основу для тем, которые будут рассматриваться в остальных главах книги, чтобы у нас с вами обязательно был общий словарный запас. Я начну с объяснения того, что такое анализ данных, а затем перейду к описанию языка SQL: что это такое, почему он так популярен, какие у него есть преимущества по сравнению с другими инструментами и как он используется в анализе данных. А поскольку сегодня анализ данных тесно связан с вычислительной техникой, на которой он выполняется, то в конце я рассмотрю различные типы баз данных, с которыми вы можете столкнуться в своей работе, когда они используются и как это влияет на написание запросов.

## 1.1. Что такое анализ данных

Сбор и хранение данных для анализа — занятие, присущее только человеку. Системы отслеживания запасов зерна, налогов и населения используются тысячи лет, а основам статистики сотни лет[1]. Смежные дисциплины, в том числе статистическое управление процессами, исследование операций и кибернетика, активно развивались в XX веке. При описании анализа данных (data analysis) используется множество различных именований, таких как аналитика (analytics), бизнес-аналитика (business intelligence, BI), наука о данных (data science), наука о принятии решений (decision science), и у специалистов в этой области есть целый ряд названий должностей. Анализом данных также занимаются и маркетологи, менеджеры по продукту, бизнес-аналитики и многие другие. В этой книге я буду использовать термины *аналитик данных* (data analyst) и *специалист по данным* (data scientist) как синонимы, говоря о человеке, обрабатывающем данные с помощью SQL. Я буду называть программное обеспечение, используемое для создания отчетов и информационных панелей (dashboard), *инструментами BI*.

---

[1] **https://ru.wikipedia.org/wiki/Статистика**

Анализ данных в современном понимании стал возможен благодаря развитию вычислительной техники и очень зависит от нее. Он сформировался под влиянием тенденций в исследованиях и в коммерциализации. Анализ данных сочетает в себе мощь компьютерных вычислений и методы традиционной статистики. Он состоит из сбора данных, интерпретации данных и публикации данных. Очень часто целью анализа является улучшение процесса принятия решений как людьми, так и все чаще машинами за счет автоматизации.

Надежная методология имеет решающее значение, но анализ — это нечто большее, чем просто получение нужного числа. Здесь важную роль играют любопытство, постановка правильных вопросов и поиск ответов на «почему получились именно такие числа». Речь также пойдет о шаблонах и аномалиях, обнаружении и интерпретации подсказок к пониманию того, как ведут себя люди и бизнес. Иногда анализ выполняется на основе набора данных, собранного специально для ответа на конкретный поставленный вопрос, например при проведении научного эксперимента или онлайн-тестирования. Анализ может также выполняться на основе данных, сгенерированных в результате ведения бизнеса, например при продаже товаров компании, или сгенерированных для целей аналитики, например для отслеживания действий пользователей на веб-сайте или в мобильном приложении. Такие данные могут иметь очень широкое применение, от устранения ошибок до планирования улучшения пользовательского интерфейса (user interface, UI), но они часто собираются в таком виде и объеме, что требуется их предварительная обработка, прежде чем делать какие-то выводы. В *гл. 2* рассматривается подготовка данных для анализа, а в *гл. 8* обсуждаются некоторые вопросы конфиденциальности, с которыми должны быть знакомы все специалисты, работающие с данными.

Трудно представить себе отрасль, которую бы не коснулся анализ данных: он повлиял на производство, розничную торговлю, финансы, здравоохранение, образование и даже работу правительства. Спортивные команды использовали анализ данных с первых лет пребывания Билли Бина (Billy Beane) на посту генерального менеджера бейсбольной команды Oakland Athletics, прославившегося благодаря книге Майкла Льюиса[2] (Michael Lewis). Анализ данных используется в маркетинге, продажах, логистике, разработке новых товаров, дизайне пользовательского интерфейса, в службах поддержки клиентов, управлении персоналом и много где еще. Сочетание методов анализа, приложений и вычислительных мощностей привело к быстрому развитию смежных областей, таких как дата-инжиниринг (data engineering) и наука о данных (data science).

Анализ данных по определению выполняется на основе исторических данных, и важно понимать, что прошлое не обязательно предсказывает будущее. Мир динамичен, и бизнес тоже динамичен: внедряются новые товары и новые процессы, появляются и исчезают конкуренты, меняется социально-политический климат. Анализ данных критикуют за ретроспективный подход. Хотя эта характеристика верна, я видела, как организации извлекают огромную пользу из анализа историче-

---

[2] *Майкл Льюис.* MoneyBall. Как математика изменила самую популярную спортивную лигу в мире. М.: Манн, Иванов и Фербер, 2014.

ских данных. Обработка исторических данных помогает понять особенности и поведение клиентов, поставщиков и процессов. Исторические данные могут помочь нам сделать обоснованные оценки и спрогнозировать возможный диапазон результатов, которые иногда будут ошибочными, но довольно часто верны. Данные о прошедших событиях могут указать нам на пробелы, слабые стороны и новые возможности. Это позволяет организациям оптимизировать рабочие процессы, экономить деньги, снижать риски и мошенничество. Анализ исторических данных также помогает организациям находить новые возможности и может стать основой для создания новых товаров, которые порадуют покупателей.

В наши дни осталось очень мало организаций, которые не занимаются анализом данных в той или иной форме, но все же есть некоторые его противники. Почему некоторые организации не используют анализ данных? Одним из аргументов является соотношение цены и качества. Сбор, обработка и анализ данных требуют усилий и определенных финансовых вложений. Некоторые организации слишком молоды или слишком бессистемны. Если нет последовательного процесса, трудно генерировать данные, которые будут достаточно непротиворечивыми для анализа. Наконец, есть этические соображения. Сбор или хранение данных об определенных людях в определенных ситуациях может регулироваться или даже запрещаться. Например, данные о детях и медицинских вмешательствах являются конфиденциальными, а их сбор строго регламентирован. Даже организациям, которые в целом ориентированы на данные, необходимо позаботиться о конфиденциальности клиентов и серьезно относиться к тому, какие данные следует собирать, зачем они нужны и как долго их следует хранить. Такие нормативные акты, как Общий регламент Европейского союза по защите данных (General Data Protection Regulation, GDPR) и Федеральный закон Российской Федерации «О персональных данных» № 152-ФЗ, изменили представление компаний о потребительских данных. Мы обсудим эти правила более подробно в *гл. 8*. Как специалисты, работающие с данными, мы всегда должны думать об этических последствиях нашей работы.

Работая с организациями, я люблю говорить людям, что анализ данных — это не проект, который завершается в фиксированную дату, это образ жизни. Развитие аналитического мышления — это процесс, а поиск значимых результатов — это приключение. Неявное становится явным, сложные вопросы разбираются до тех пор, пока на них не будут получены ответы, и наиболее важные результаты встраиваются в информационные панели, которые влияют на принятие тактических и стратегических решений. Благодаря этой информации перед аналитиками ставятся новые и более сложные вопросы, а затем процесс повторяется.

Анализ данных одновременно прост для тех, кто начинает изучение, и сложен для тех, кто хочет поднять свой уровень. Технические методы можно изучить, особенно язык SQL. С многими проблемами, такими как оптимизация расходов на маркетинг или обнаружение мошенничества, сталкивались многие организации, и они знают как их решать. Каждая организация уникальна, и каждый набор данных имеет свои особенности, поэтому даже знакомые проблемы могут создавать новые трудности. Интерпретация результатов — это тоже навык. Чтобы научиться давать хорошие рекомендации и стать значимым сотрудником организации, требуется время.

По моему опыту, простой анализ, представленный убедительно, оказывает большее впечатление, чем сложный анализ, представленный плохо. Успешный анализ данных также невозможен без сотрудничества. У вас могут быть отличные идеи, но если их некому реализовать, вы не окажете особого влияния. Даже со всеми технологиями люди по-прежнему важны, и общение с коллегами имеет значение.

## 1.2. Почему SQL

В этом разделе рассказывается, что такое SQL, преимущества его использования, его сравнение с другими языками, часто используемыми для анализа, и, наконец, как SQL вписывается в рабочий процесс анализа.

### Что такое SQL

SQL — это язык, используемый для взаимодействия с базами данных. Аббревиатура расшифровывается как Structured Query Language (язык структурированных запросов) и произносится либо как «сиквел» (sequel), либо по буквам — «эс кю эль». Это только первое из многих противоречий и несоответствий, связанных с SQL, которые мы увидим, но большинство людей поймут, что вы имеете в виду, независимо от того, как вы это произнесете. Ведутся споры о том, является ли SQL вообще языком программирования. Это не язык общего назначения, как C или Python. Сам по себе SQL без базы данных и без данных в таблицах — это просто текстовый файл. SQL не может создать веб-сайт, но он позволяет эффективно работать с данными в базах данных. На практическом уровне важнее всего то, что SQL помогает выполнять работу по анализу данных.

Компания IBM первой разработала базы данных с SQL на основе реляционной модели, изобретенной Эдгаром Коддом (Edgar Codd) в 1960-х гг. Эта реляционная модель представляла собой теоретическое описание управления данными с помощью связей. Создав первые базы данных, IBM помогла развить теорию, но у нее были также и коммерческие интересы, как у Oracle, Microsoft и любой другой компании, которые с тех пор коммерциализируют базы данных. С самого начала существовало противоречие между компьютерной теорией и коммерческой необходимостью. SQL стал стандартом Международной организации по стандартизации (International Organization for Standards, ISO) в 1987 г. и стандартом Американского национального института стандартов (American National Standards Institute, ANSI) в 1986 г. Хотя во всех основных базах данных SQL реализован, опираясь на эти стандарты, многие реализации имеют свои особенности и функции, облегчающие жизнь пользователям этих баз данных. Это привело к тому, что SQL-код стало сложнее перемещать между базами данных без каких-либо изменений.

Язык SQL используется для доступа, управления и извлечения данных из объектов базы данных. Базы данных могут иметь одну или несколько *схем* (schema), каждая из которых поддерживает свою структуру данных и содержит разные объекты. Внутри схемы основными объектами, которые чаще всего используются при анали-

зе данных, являются таблицы, представления и функции. *Таблицы* (table) содержат поля, в которых хранятся данные. Таблица может иметь один или несколько *индексов* (index) — это особый тип структуры данных, который позволяет более эффективно извлекать данные. Индексы обычно определяются администратором базы данных (database administrator, DBA). *Представления* (view) — это, по сути, сохраненные SQL-запросы, на которые можно ссылаться так же, как на таблицы. *Функции* (function) позволяют сохранить часто используемые вычисления или операции и легко использовать их в запросах. Обычно они создаются администратором базы данных. На рис. 1.1 представлен пример простой структуры базы данных.


**Рис. 1.1.** Пример структуры базы данных и ее объектов

Для взаимодействия с базами данных SQL включает в себя четыре категории для решения различных задач, и они в основном являются стандартными для всех типов баз данных. Специалистам, занимающимся анализом данных, не нужно ежедневно помнить названия этих категорий, но они могут всплывать в разговоре с администраторами баз данных или разработчиками, поэтому я кратко их приведу. Все операторы SQL гибко работают друг с другом.

*DQL* (*data query language*), или *язык запросов*, — это то, чему в основном и посвящена эта книга. Он используется для запрашивания данных, и его можно рассматривать как код для составления вопросов к базе данных. DQL включает в себя один оператор `SELECT`, который знаком всем, кто когда либо работал с SQL, но, по моему опыту, аббревиатура DQL используется очень редко. SQL-запросы могут быть очень короткими и состоять из одной строки, а могут занимать несколько десятков строк кода. Запросы могут обращаться к одной таблице (или представлению) или объединять данные из нескольких таблиц, а также использовать несколько схем

одной и той же базы данных. SQL, как правило, не может выполнять запросы между разными базами данных, но в некоторых случаях можно использовать умные настройки сети или дополнительное программное обеспечение, чтобы извлекать данные из нескольких источников, даже из баз данных разных типов. Сами по себе SQL-запросы автономны и кроме таблиц не ссылаются ни на переменные, ни на результаты выполнения предыдущих шагов, которых нет в запросе, что отличает SQL от сценарных языков.

*DDL* (*data definition language*), или *язык определения данных*, используется для создания и изменения таблиц, представлений, пользователей и других объектов базы данных. Он работает со структурой, а не с содержанием. Есть три общих команды: `CREATE`, `ALTER` и `DROP`. Команда `CREATE` используется для создания новых объектов, `ALTER` изменяет структуру объекта, например добавляет столбец в таблицу, а `DROP` удаляет весь объект и его структуру. Вы могли слышать, как администраторы баз данных и специалисты по обработке данных говорят о работе с DDL — на самом деле речь, как правило, идет о файлах или фрагментах кода, которые создают, изменяют или удаляют какие-то объекты. Примером DDL в контексте анализа является код для создания временных таблиц.

*DCL* (*data control language*), или *язык управления данными*, используется для управления доступом. Он включает в себя команды `GRANT` и `REVOKE` — предоставление привилегий и удаление привилегий соответственно. В контексте анализа вам может понадобиться команда `GRANT`, чтобы ваш коллега мог выполнить запрос к созданной вами таблице. Вы также можете столкнуться с этой командой, когда кто-то говорит вам, что таблица существует в базе данных, но вы ее не видите — возможно, вашему пользователю необходимо дать соответствующую привилегию.

*DML* (*data manipulation language*), или *язык манипулирования данными*, используется для работы с самими данными. Это команды `INSERT`, `UPDATE` и `DELETE`. Команда `INSERT` добавляет новые записи и, по сути, является первым шагом в процессе ETL (extract, transform, load). Команда `UPDATE` изменяет значение в поле таблицы, а `DELETE` удаляет строки целиком. Вы столкнетесь с этими командами, если вам придется работать с временными таблицами или с таблицами-песочницами, или если вы окажетесь в роли и владельца базы данных, и аналитика данных.

Эти четыре категории SQL реализованы во всех основных базах данных. В книге я буду использовать главным образом DQL. Я упомяну несколько команд DDL и DML в *гл. 8*, и вы также можете найти несколько примеров в репозитории GitHub для этой книги[3], где я использую эти команды для создания и заполнения таблиц. Благодаря этому универсальному набору команд SQL-код, написанный для любой базы данных, будет выглядеть знакомым для любого, кто работает с SQL. Однако при чтении SQL-кода из другой базы данных вам может показаться, что кто-то говорит на вашем же языке, но приехал из другого региона или другой страны. Базовая структура языка та же, но говорок другой, а некоторые слова вообще имеют разные значения. Вариации SQL от базы к базе часто называют *диалектами*,

[3] **https://github.com/cathytanimura/sql_book**

и пользователи баз данных могут писать на PL/SQL от Oracle, T-SQL от Microsoft SQL Server и др.

Тем не менее, если вы освоите SQL, вы сможете работать с разными типами баз данных, нужно будет только обратить внимание на такие детали, как обработка `null`-значений, дат и временных меток, целочисленное деление и чувствительность к регистру.

В этой книге для примеров используется PostgreSQL или Postgres, хотя я постараюсь уточнять, где код будет существенно отличаться от других типов баз данных. Вы можете установить Postgres[4] на ваш компьютер, чтобы работать с примерами.

## Преимущества SQL

Есть много веских причин использовать SQL для анализа данных, в том числе его вычислительная мощность, широкое распространение в инструментах анализа данных и его гибкости.

Возможно, главная причина использования SQL заключается в том, что основная часть всех данных в мире уже хранится в базах данных. Скорее всего, в вашей организации тоже уже есть одна или несколько баз данных. Если же ваши данные еще не записываются в базу данных, то создание такой базы и загрузка в нее данных могут иметь смысл — это позволит воспользоваться преимуществами хранения и вычислений, особенно по сравнению с такими альтернативами, как электронные таблицы (spreadsheet). В последние годы мощность вычислительной техники резко возросла, а хранилища и инфраструктура данных сильно эволюционировали, и этим преимуществом нужно пользоваться. Некоторые новые облачные базы данных позволяют загружать огромные объемы данных в память, и это еще больше ускоряет работу. Дни, когда ожидание результатов запроса занимало минуты или часы, заканчиваются, хотя аналитики в ответ на это могут просто начать писать все более и более сложные запросы.

SQL является стандартом де-факто для взаимодействия с базами данных и извлечения из них данных. Огромное количество популярного программного обеспечения подключается к базам данных с помощью SQL, от электронных таблиц до инструментов BI и визуализации, а также такие языки программирования, как Python и R (они будут обсуждаться в следующем разделе). Благодаря доступности вычислительных ресурсов, выполнение как можно большего числа операций и агрегирований на стороне базы данных часто дает выигрыш для последующих этапов анализа. Мы подробно разберем стратегии создания сложных наборов данных для их дальнейшей обработки в других инструментах в *гл. 8*.

Базовые структурные элементы SQL можно комбинировать бесконечным числом способов. Даже с помощью относительно небольшого количества кода SQL можно выполнять широкий спектр задач. SQL можно писать последовательно, просматри-

[4] **https://www.postgresql.org/download**

вая промежуточные результаты по ходу работы. Возможно, это и не полноценный язык программирования, но он может очень многое, от преобразования данных до выполнения сложных вычислений.

И, наконец, SQL относительно прост в изучении благодаря ограниченному синтаксису. Вы можете быстро освоить основные ключевые слова и структуру кода, а затем со временем отточить свое мастерство, работая с различными наборами данных. Практическое применение SQL почти безгранично, если принять во внимание огромное разнообразие наборов данных в мире и всевозможные вопросы, которые ставятся в отношении данных. SQL преподают во многих университетах, а многие люди приобретают эти навыки на работе. Даже сотрудники, которые еще не умеют работать с SQL, могут легко научиться, и обучение будет проще, чем для других языков программирования. Поэтому организации вполне логично выбирают реляционные базы для хранения данных для анализа.

## SQL против R или Python

Хотя SQL является популярным языком для анализа данных, это не единственный вариант. Есть еще R и Python — одни из самых широко используемых языков для анализа данных. R — это язык статистики и графиков, а Python — язык программирования общего назначения, который отлично подходит для работы с данными. Оба языка имеют открытый исходный код, могут быть установлены на ноутбук и имеют активные сообщества пользователей, которые разрабатывают дополнительные пакеты и расширения для различных задач обработки и анализа данных. Сравнение R и Python выходит за рамки этой книги, но в интернете ведется множество дискуссий о преимуществах каждого из них. Здесь я рассмотрю их вместе как альтернативу языку SQL.

Одним из основных различий между SQL и другими языками программирования является то, где выполняется код и, следовательно, сколько вычислительной мощности доступно. SQL всегда выполняется на сервере базы данных, используя его вычислительные ресурсы. R и Python, как правило, запускаются локально на вашем компьютере, поэтому вычислительные ресурсы ограничены его мощностью. Конечно, есть много исключений: база данных может быть развернута на ноутбуке, а R и Python можно запускать на сервере с большой мощностью. Когда вы работаете с большим набором данных и выполняете что-то посложнее, чем простой анализ, то хорошим решением будет перенос работы на сторону сервера базы данных с большим количеством ресурсов. Поскольку базы данных, как правило, постоянно получают новые данные, то SQL также будет хорошим выбором в случаях, когда нужно периодически обновлять отчеты или информационные панели.

Второе отличие заключается в том, как данные хранятся и как они организованы. Реляционные базы данных всегда организуют данные в строки и столбцы таблиц, поэтому SQL использует эту структуру в каждом запросе. В R и Python есть более широкие возможности для организации данных: переменные, списки, словари и т. д. Это обеспечивает бóльшую гибкость, но требует хороших навыков програм-

мирования. Для облегчения анализа в R есть фреймы данных (data frame), которые похожи на таблицы базы и организуют данные в строки и столбцы. Пакет pandas позволяет работать с фреймами данных и в Python. Даже когда есть и другие варианты, структура таблицы остается самой удобной для анализа.

Организация циклов — еще одно важное отличие SQL от большинства языков программирования. *Цикл* — это инструкция или последовательность инструкций, которая повторяется до тех пор, пока не будет выполнено заданное условие выхода из цикла. Функции агрегирования в SQL неявно перебирают набор данных без какого-либо дополнительного кода. Мы увидим примеры, когда из-за невозможности циклического перебора полей может получиться очень длинный SQL-запрос для сворачивания или разворачивания данных. Более подробное обсуждение циклов выходит за рамки этой книги, но некоторые поставщики баз данных создали свои расширения для SQL, например PL/SQL от Oracle и T-SQL от Microsoft SQL Server, в которых реализована такая функциональность, как циклы.

Недостатком SQL может считаться то, что ваши данные должны храниться в базе данных[5], тогда как код на R и Python может импортировать данные из файлов, хранящихся локально, или даже может получить доступ к файлам, хранящимся на серверах. Это удобно для многих проектов. Хотя базу данных можно развернуть и на ноутбуке, но это все равно создает дополнительные накладные расходы. С другой стороны, такие пакеты, как dbplyr для R и SQLAlchemy для Python, позволяют приложениям, написанным на этих языках, подключаться к базе данных, выполнять SQL-запросы и использовать полученные результаты для последующий обработки. В этом смысле R или Python могут дополнять SQL.

В R и Python есть сложные статистические функции, которые либо встроены, либо доступны в сторонних пакетах. Хотя в SQL есть, например, функции для вычисления среднего значения и стандартного отклонения, расчеты *p*-значений и статистической достоверности, которые необходимы для анализа экспериментов (*см. гл. 7*), не могут быть выполнены только с помощью SQL. Кроме сложной статистики, есть еще машинное обучение (machine learning, ML), для которого тоже лучше выбрать R или Python.

При принятии решения о том, какой язык использовать для анализа — SQL, R или Python, учитывайте следующие моменты.

- Где находятся ваши данные — в базе данных, в файлах, на сервере?
- Каков объем данных?
- Для чего будут использоваться данные — для отчетов, визуализации, статистического анализа?
- Нужно ли будет обновлять или добавлять новые данные? Как часто?
- Что уже использует ваша команда или организация? Насколько важно соответствовать принятым в организации стандартам?

---

[5] Уже существует несколько новых технологий, позволяющих выполнять SQL-запросы к данным, хранящимся в нереляционных источниках.

Не прекращаются споры о том, какие языки и инструменты лучше всего подходят для анализа данных. Как и во многих случаях, всегда существует более одного способа сделать анализ. Языки программирования постоянно развиваются и набирают популярность, и нам повезло жить и работать во времена, когда есть так много хороших вариантов. SQL существует уже достаточно давно и, вероятно, останется популярным еще долгие годы. В конечном счете, нужно стремиться использовать лучший доступный инструмент для работы. Эта книга поможет вам извлечь максимальную выгоду от использования SQL при анализе данных, независимо от того, что еще есть в вашем инструментарии.

## SQL как часть процесса анализа данных

Теперь, когда мы обсудили, что такое SQL и в чем его преимущество, и сравнили его с другими языками, давайте поговорим о том, какое место SQL занимает в процессе анализа данных. Работа над анализом данных всегда начинается с постановки вопроса, например, сколько новых клиентов было привлечено, какова динамика продаж или почему одни пользователи остаются с нами надолго, в то время как другие знакомятся с сервисом и никогда больше не возвращаются. После того, как вопрос сформулирован, мы рассматриваем, откуда приходят данные, где они хранятся, определяем план анализа и то, как результаты анализа будут представлены аудитории. На рис. 1.2 показаны этапы этого процесса. Эта книга посвящена этапу «Запросы и анализ», хотя я кратко расскажу и о других этапах, чтобы вы понимали окружающий контекст.


**Рис. 1.2.** Этапы процесса анализа данных

На первом этапе данные генерируются *исходными системами* (source systems), и этот термин включает в себя любой человеческий или автоматический процесс, генерирующий нужные нам данные. Данные могут генерироваться людьми вручную, например, когда кто-то заполняет анкету или карту пациента во время приема у врача. Данные также могут быть сгенерированы автоматически, например, когда приложение записывает в базу данных информацию о покупке, или система регистрации событий фиксирует клик по ссылке на веб-сайте, или система управления маркетингом отмечает открытие электронного письма. Исходные системы могут генерировать множество данных различных типов и форматов. Далее в *гл. 2* более подробно рассматриваются типы данных и то, как тип источника данных может повлиять на анализ.

Второй этап — перемещение данных и запись их в базу данных для анализа. Я буду использовать термины *хранилище данных* (data warehouse), представляющее собой базу данных, в которую записываются данные обо всей деятельности организации, и *склад данных* (data store), относящийся к любому типу системы хранения данных, к которой можно обращаться. Вам могут встретиться и другие термины: *витрина данных* (data mart), которая обычно является подмножеством хранилища данных или более специализированным хранилищем данных; *озеро данных* (data lake), означающее, что данные хранятся в файловой системе или в базе данных, но не в таком структурированном виде, как в хранилище данных. Хранилища данных могут быть как маленькими и простыми, так и огромными и дорогими. Для того чтобы следовать примерам из этой книги, вам будет достаточно базы данных, установленной на ноутбуке. Важно, чтобы все данные, необходимые для выполнения анализа, хранились в одном месте.

Обычно за загрузку данных в хранилище данных отвечает человек или команда. Этот процесс называется *ETL* (extract, transform, load — извлечение, преобразование, загрузка). На этапе извлечения данные выгружаются из исходной системы. На этапе преобразования дополнительно изменяется структура данных, выполняется их очистка или агрегирование. Этап загрузки помещает данные в базу данных. Есть также альтернативный процесс, который называется *ELT* (извлечение, загрузка, преобразование), — разница только в последовательности действий: вместо преобразования данных перед загрузкой сначала загружаются все данные, а потом выполняются преобразования, как правило, с помощью SQL. В контексте ETL вы также можете услышать термины *источник данных* (source) и *получатель данных* (target). Источник — это откуда поступают данные, а получатель — база данных и таблицы в ней. Даже когда для преобразования данных используется SQL, то другой язык, такой как Python или Java, как правило, используется для последовательного запуска шагов, управления расписанием и обработки ошибок, если что-то пойдет не так. Существуют как коммерческие приложения, так и инструменты с открытым исходным кодом, поэтому вам не нужно создавать систему ETL полностью с нуля.

Как только данные сохранены в базу данных, можно переходить к следующему этапу — выполнению запросов и анализу. На этом этапе для изучения, профилирования, очистки, структурирования и анализа результатов применяется SQL. На рис. 1.3 показан общий ход процесса. Изучение данных включает в себя знакомство с предметной областью, в которой они были сгенерированы, и с таблицами базы данных, в которых они хранятся. Профилирование включает в себя проверку возможных значений и их распределений в наборе данных. К очистке данных относятся исправление неправильных или неполных данных, добавление категорий и флагов, обработка null-значений. Структурирование — это процесс организации данных в строки и столбцы, необходимые в результирующем наборе. И, наконец, анализ результатов включает в себя просмотр результатов для выявления тенденций, подведения итогов и формулирования предположений. Хотя этот процесс изображен линейно, на практике он часто цикличен, например, когда при структурировании или анализе результатов обнаруживаются данные, которые необходимо очистить.


**Рис. 1.3.** Последовательные шаги при выполнении этапа «Запросы и анализ»

Представление результатов в окончательном виде является последним этапом в общем процессе анализа данных. Ваши заказчики не поймут файла с SQL-кодом, они ожидают, что вы представите им графики, диаграммы и интересные выводы. Чтобы анализ данных принес какую-то пользу, вам нужен способ поделиться результатами с другими людьми. В некоторых случаях вам может потребоваться более сложный статистический анализ, что можно сделать с помощью SQL, или вы можете передать данные в алгоритм машинного обучения. К счастью, большинство инструментов отчетности и визуализации имеют коннекторы SQL, которые позволяют вам получать данные напрямую из таблиц или с помощью предварительно написанных SQL-запросов. Статистическое программное обеспечение и языки, используемые для машинного обучения, обычно тоже имеют коннекторы SQL.

Весь процесс анализа данных состоит из нескольких этапов и часто использует различные инструменты и технологии. SQL-запросы и анализ результатов лежат в основе многих исследований, и именно на них мы сосредоточимся в следующих главах. А в оставшихся разделах этой главы мы рассмотрим типы баз данных, с которыми вы можете столкнуться при выполнении различных анализов.

## 1.3. Типы баз данных

Если вы планируете работать с SQL, вам придется разбираться и в базах данных. Есть много разных типов баз данных: с открытым исходным кодом или проприетарные, строчные или колоночные. Существуют как локальные, так и облачные базы данных, а также гибридные базы данных, когда организация сама разворачивает базу данных в облачной инфраструктуре. Существуют также хранилища данных, которые, вообще говоря, не являются базами данных, но к которым можно обращаться с помощью SQL.

Не все базы данных работают одинаково. Когда дело доходит до выполнения анализа данных, каждый тип базы данных имеет свои сильные и слабые стороны. В отличие от инструментов, применяемых на других этапах анализа данных, вы, вероятно, не можете сами решать, какой тип базы данных будет использоваться в вашей организации. Знание всех нюансов имеющейся у вас базы данных поможет вам работать более эффективно и использовать преимущества специальные функции SQL, которые в ней реализованы. Знание других типов баз данных пригодится, если вам придется работать над проектом по созданию или миграции в новое хра-

нилище данных. Вы можете установить базу данных на ноутбук или получить экземпляр облачного хранилища для небольших личных проектов.

Базы данных и хранилища данных относятся к динамично развивающимся областям технологий с самого момента их появления. Некоторые тенденции начала XXI в. привели к тому, что эти технологии сегодня по-настоящему востребованы специалистами по обработке данных. Во-первых, объемы данных невероятно выросли с развитием Интернета, мобильных устройств и Интернета вещей (Internet of Things, IoT). В 2020 г. компания IDC предсказала[6], что к 2025 г. объем данных, хранящихся во всем мире, вырастет до 175 зеттабайт. О таком масштабе данных трудно даже подумать, и не все они будут храниться в базах данных и использоваться для целей анализа. В наши дни компании нередко владеют терабайтами и петабайтами данных — объемы, которые было бы невозможно обработать с помощью технологий 1990-х гг. Во-вторых, снижение затрат на хранение данных и вычислительные мощности, а также появление облачных технологий удешевили и упростили сбор и хранение этих огромных объемов данных для организаций. Оперативная память стала дешевле, а это значит, что бóльшие объемы данных могут быть загружены, для них выполнены вычисления и получены результаты, и все это без чтения и записи на диск, что значительно увеличивает скорость. В-третьих, распределенные вычисления позволяют разделить рабочие нагрузки между множеством машин. Это делает возможным выполнять большие объемы вычислений для сложных задач обработки данных.

Базы данных и хранилища данных объединяют в себе эти тенденции разными способами, что позволяет оптимизировать их под конкретные типы задач. Существуют два типа баз данных, которые имеют отношение к аналитической работе: строчные и колоночные базы данных. В следующих разделах я разберу, чем они схожи и чем отличаются друг от друга, и расскажу о том, как тип базы влияет на проведение анализа хранящихся в ней данных. Наконец, я представлю некоторые дополнительные типы хранилищ данных, которые отличаются от баз данных и с которыми вы можете встретиться по работе.

## Строчные базы данных

*Строчные базы данных* (row-store database, базы данных с хранением данных по строкам), также называемые *транзакционными*, предназначены для эффективной обработки транзакций: `INSERT`, `UPDATE` и `DELETE`. Среди популярных строчных баз данных с открытым исходным кодом можно назвать MySQL и Postgres. Из коммерческих баз данных широко используются Microsoft SQL Server, Oracle и Teradata. Хотя на самом деле они не оптимизированы для анализа, в течение ряда лет строчные базы данных были единственным вариантом для создания корпоративных хранилищ данных. Благодаря тщательной настройке и правильному дизайну схем эти

[6] **https://www.seagate.com/files/www-content/our-story/rethink-data/files/Rethink_Data_Report_2020.pdf**

базы данных можно использовать для аналитики. Привлекательными их также делает низкая стоимость программных продуктов с открытым исходным кодом и то, что с ними знакомы многие спецаилисты и администраторы баз данных. Многие организации начинают построение своей рабочей инфраструтуры с разворачивания именно баз данных. По всем этим причинам аналитикам и специалистам по данным на каком-то этапе своей карьеры, скорее всего, приходится знакомиться со строчными базами данных.

Мы представляем таблицу в виде строк и столбцов, но данные должны быть сериализованы для хранения. Запрос выполняет поиск необходимых данных на жестком диске. Жесткие диски организованы в виде последовательности блоков фиксированного размера. Сканирование жесткого диска требует времени и ресурсов, поэтому важно свести к минимуму объем диска, который необходимо просканировать для получения результатов запроса. Строчные базы данных решают эту проблему, сериализуя данные по строкам. На рис. 1.4 показан пример построчного хранения данных. При запросе вся строка считывается в память. Этот подход быстрый при обновлении строк, но медленный при выполнении вычислений со множеством строк, если используются только некоторые столбцы.

| id | sku | type | color | size | price |
|---|---|---|---|---|---|
| 1 | 123 | tshirt | black | S | 19.99 |
| 2 | 124 | shorts | green | M | 24.99 |

**Рис. 1.4.** Строчное хранилище,
в котором каждая строка записывается целиком на диск

Чтобы уменьшить ширину таблиц, строчные базы данных обычно моделируются в *третьей нормальной форме* — это подход к проектированию баз данных, при котором нужно стремиться хранить каждый фрагмент информации только один раз, чтобы избежать дублирования и несоответствий. Это эффективно для работы с транзакциями, но часто приводит к большому количеству таблиц в базе данных, каждая из которых имеет всего несколько столбцов. Для анализа таких данных может потребоваться множество соединений, и не-разработчикам бывает сложно понять, как таблицы связаны друг с другом и где хранится конкретная часть данных. При проведении анализа обычно приходится выполнять денормализацию, т. е. собирать все данные в одном месте.

Таблицы обычно имеют *первичный ключ* (primary key), который обеспечивает уникальность записей — другими словами, он не позволяет создавать более одной записи для одного и того же объекта. Таблицы часто имеют столбец с именем `id` (идентификатор), который представляет собой автоматически увеличивающееся целое число, и каждая новая запись получает следующее по порядку целое число или буквенно-цифровое значение, создаваемое генератором первичного ключа. Также в таблице может быть набор столбцов, которые вместе делают строку уникальной — эта комбинация полей называется *составным ключом* (composite key) или иногда *бизнес-ключом*. Например, в таблице с данными о клиентах столбцы

`first_name`, `last_name` и `birthdate` вместе могут считаться уникальной комбинацией. Номер социального страхования `Social_security_id` тоже может быть уникальным идентификатором, как и столбец `person_id`.

Таблицы могут иметь индексы, которые ускоряют поиск записей и ускоряют соединения с участием этих столбцов. Индексы сохраняют значения в каком-то поле или в нескольких полях, проиндексированные как отдельные фрагменты данных вместе с указателем на строки, и, поскольку индексы меньше, чем вся таблица, их быстрее сканировать. Обычно индексируется первичный ключ, но могут индексироваться и другие поля или набор полей. При работе со строчными базами данных полезно знать, какие поля в таблицах, с которыми вы работаете, имеют индексы. Обычные соединения можно ускорить, добавив индексы на соответствующие поля, поэтому стоит выяснить, не будут ли аналитические запросы выполняться слишком долго. У индексов есть свои недостатки: они занимают много места в памяти и замедляют загрузку данных, т. к. при каждой вставке необходимо добавлять новые значения и в индекс. Администраторы баз данных могут отказаться индексировать все, что может быть полезно для анализа. Помимо отчетности, анализ вряд ли является рутинной операций, чтобы заниматься оптимизацией индексов специально для него. Исследовательские и сложные запросы часто используют хитрые соединения, но мы всегда можем отказаться от такого подхода в пользу другого, более эффективного способа решения проблемы.

*Схема звезды*[7] (star schema) была разработана отчасти для того, чтобы сделать строчные базы данных более удобными для аналитической работы. Основы такого моделирования хранилища данных изложены в книге The Data Warehouse Toolkit[8], в которой рекомендуется моделировать данные в виде совокупности таблиц фактов и таблиц измерений. Таблицы фактов представляют собой события, например транзакции розничного магазина. Таблицы измерений содержат разные дескрипторы, такие как имя клиента и тип товара. Поскольку данные не всегда можно точно разложить в категории фактов и измерений, существует расширение этой схемы, которое называется *схемой снежинки*[9] (snowflake schema), где некоторые измерения имеют свои собственные измерения.

## Колоночные базы данных

*Колоночные базы данных* (column-store database, базы данных с хранением данных по столбцам) появились в начале XXI в., хотя их теоретическая основа появилась одновременно со строчными базами данных. Колоночные базы данных хранят вместе значения столбцов, а не значения строк. Этот дизайн оптимизирован для запросов, которые считывают много записей, но только по нескольким столбцам. К популярным колоночным базам данных относятся Amazon Redshift, Snowflake и Vertica.

---

[7] **https://ru.wikipedia.org/wiki/Схема_звезды**

[8] *Ralph Kimball* and *Margy Ross*, The Data Warehouse Toolkit, 3rd ed. (Indianapolis: Wiley, 2013).

[9] **https://ru.wikipedia.org/wiki/Схема_снежинки**

Колоночные базы данных эффективны при хранении больших объемов данных благодаря сжатию. Отсутствующие и повторяющиеся значения могут быть представлены лишь очень маленькими маркерами вместо полного значения. Например, вместо того, чтобы хранить строковое значение «Российская Федерация» тысячи или миллионы раз, колоночная база данных будет хранить закодированное значение, которое занимает очень мало места, и ссылку на полное значение «Российская Федерация». Колоночные базы данных также сжимают данные, оптимизируя повторы значений в отсортированных данных. Например, в базе данных может храниться метка, что закодированное значение «Российская Федерация» повторяется 100 раз, и она занимает намного меньше места, чем 100 записей этого закодированного значения.

Колоночные базы данных не требуют использования первичных ключей, и в них нет индексов. Повторяющиеся значения не являются проблемой благодаря сжатию. В результате схемы могут быть адаптированы для аналитических запросов, когда все данные находятся в одном месте, а не в нескольких таблицах, которые нужно было бы объединять. Однако без первичных ключей легко могут появиться дубликаты данных, поэтому важно знать источник данных и проверять их качество.

Обновления и удаления записей являются дорогостоящими операциями в большинстве колоночных баз данных, поскольку данные одной строки не хранятся вместе. Для очень больших таблиц может быть принята политика «только для записи», поэтому нужно также знать и то, как генерируются данные, чтобы выяснить, какие записи использовать в анализе. Данные также могут считываться чуть медленнее, поскольку перед использованием их необходимо распаковать.

Колоночные базы данных обычно являются лучшим выбором для быстрой аналитической работы. Они используют стандартный SQL (с некоторыми вариациями от поставщика), и во многих отношениях работа с ними ничем не отличается от работы со строчной базой данных, с точки зрения написания запросов. Размер данных имеет значение, как и выделенные ресурсы для хранения данных и вычислений. Я видела, как агрегации на миллионах и миллиардах записей обрабатывались за секунды. Это чудеса производительности.

Есть несколько нюансов, о которых следует знать. Поскольку некоторые виды сжатия основаны на сортировке, знание полей, по которым отсортирована таблица, и использование их для фильтрации запросов повышает производительность. Объединение таблиц может быть очень медленным, если обе таблицы достаточно большие.

В конце концов, вы можете выбрать любой тип базы данных, с которым проще или быстрее работать, — ничто не помешает вам выполнить какой-либо анализ, описанный в этой книге. Как и всегда, использование инструмента, достаточно мощного для вашего объема данных и сложности ваших задач, позволяет сосредоточиться на проведении качественного анализа.

## Другие типы хранилищ данных

Базы данных — не единственный способ хранения данных, и с каждым годом появляется все больше вариантов хранения данных, которые можно использовать для анализа и работы приложений. Системы хранения файлов, иногда называемые *озерами данных* (data lake), вероятно, являются основной альтернативой базам данных. Базы данных NoSQL и хранилища данных на основе поиска — это альтернативные системы хранения данных, которые обеспечивают быстрое время ожидания для приложений и при поиске в лог-файлах. Хотя они, как правило, не используются в процессе анализа, но все чаще становятся частью инфраструктуры организаций, поэтому я кратко расскажу о них в этом разделе. Следует отметить одну интересную тенденцию: хотя эти новые системы хранения данных изначально были нацелены на то, чтобы выйти за рамки ограничений баз данных SQL, в конечном итоге многие из них реализовали интерфейс SQL для получения данных.

Система Hadoop, также известная как HDFS (Hadoop distributed filesystem, распределенная файловая система Hadoop), представляет собой систему хранения файлов, которая использует преимущества постоянно снижающейся стоимости хранения данных, вычислительных мощностей и распределенных систем. Файлы разбиваются на блоки, и Hadoop распределяет их по файловой системе, которая хранится в узлах (компьютерах) кластера. Код для выполнения операций отправляется в узлы, и они обрабатывают данные параллельно. Большим преимуществом системы Hadoop стало то, что она позволяет дешево хранить огромные объемы данных. Многие крупные интернет-компании с огромными объемами часто неструктурированных данных сочли это более выгодным для себя по сравнению со стоимостью и ограничениями традиционных баз данных. Ранние версии Hadoop имели два основных недостатка: для извлечения и обработки данных требовались специальные навыки, т. к. они не был совместимы с SQL, и время выполнения программ часто было довольно большим. С тех пор Hadoop стала более эффективной, и были разработаны различные инструменты, которые обеспечивают SQL или напоминающий SQL доступ к данным и ускоряют время выполнения запросов.

За последние несколько лет на рынке появились другие коммерческие системы и приложения с открытыми исходными кодами, которые предоставляют дешевое хранилище данных и быструю обработку данных, а также возможность использования SQL-запросов. Некоторые из них даже позволяют аналитику написать всего один запрос, который будет возвращать данные из нескольких источников. Это очень удобно для всех, кто работает с большими объемами данных, и еще раз подтверждает тот факт, что SQL останется с нами навсегда.

NoSQL — это технология, позволяющая моделировать данные, которые не являются строго реляционными. Это обеспечивает очень низкую задержку при выполнении операций хранения и извлечения, что имеет решающее значение для многих онлайн-приложений. Такие системы включают в себя хранилища данных в виде пар «ключ-значение», графовые базы данных, которые организованы в формате «узел-ребро», и хранилища документов. Примерами таких хранилищ данных, о которых вы могли слышать в своей организации, являются Cassandra, Couchbase, DynamoDB, Memcached, Giraph и Neo4j. Раньше класс систем NoSQL позициони-

ровался как замена устаревшему SQL, но в последнее время эта аббревиатура интерпретируется как Not Only SQL (не только SQL). Чтобы использовать данные, хранящиеся в NoSQL как «ключ-значение», в целях анализа обычно требуется поместить их в более традиционное SQL-хранилище, поскольку NoSQL не оптимизирован для работы одновременно со многими записями. Графовые базы данных применяются в сетевом анализе (network analysis), и аналитическая работа может выполняться непосредственно в них с помощью специального языка запросов. Однако инструментарий постоянно развивается, и, возможно, когда-нибудь мы сможем анализировать эти данные и с помощью SQL.

Хранилища данных на основе поиска — это Elasticsearch и Splunk. Они часто используются для анализа машинных данных, например журналов (логов). В подобных системах используются языки запросов, отличные от SQL, но если вы знаете SQL, вы сможете легко их понять. Осознавая то, насколько широко распространен SQL, некоторые хранилища данных, такие как Elasticsearch, добавили интерфейс для выполнения SQL-запросов. Эти хранилища удобны и эффективны для тех случаев, для которых они были разработаны, но они плохо подходят для анализа данных, который рассматривается в этой книге. Как я объясняла людям на протяжении многих лет, такие хранилища отлично подходят для поиска иголки в стоге сена, но они не очень хороши в оценке самого сена.

Независимо от типа базы данных или типа хранилища данных тенденция очевидна: по мере роста объемов данных и усложнения задач язык SQL по-прежнему остается стандартным инструментом для доступа к данным. Его широкая популярность среди пользователей, легкая обучаемость и большие возможности для выполнения аналитических задач приводят к тому, что даже те технологии, которые пытаются отойти подальше от SQL, все равно возвращаются и приспосабливаются к его использованию.

## 1.4. Заключение

Анализ данных — интересная дисциплина, применяемая в самых разных отраслях бизнеса. SQL обладает многими преимуществами для работы с данными, особенно с теми, которые хранятся в базах данных. Составление запросов и анализ результатов являются частью более крупного процесса анализа, и существует несколько типов хранилищ данных, с которыми может столкнуться специалист по данным. Теперь, когда мы разобрались, что такое анализ, SQL и хранилище данных, остальные главы книги будут посвящены использованию языка SQL для выполнения детального анализа данных. *Глава 2* посвящена подготовке данных к анализу, начиная с определения типов данных и переходя затем к профилированию, очистке и структурированию данных. В *главах* с *3* по *7* разобраны примеры различных анализов данных, в том числе анализ временных рядов, когортный анализ, текстовый анализ, обнаружение аномалий и анализ экспериментов. В *главе 8* рассматриваются методы обработки сложных наборов данных для последующего анализа с помощью других инструментов. Наконец, в завершающей *главе 9* приведены некоторые идеи о том, как можно комбинировать типы анализов для получения интересных результатов, и перечислены некоторые дополнительные ресурсы, которые помогут вам в этом приключении.

# ГЛАВА 2

# Подготовка данных для анализа

Оценки того, сколько времени специалисты по данным тратят на подготовку своих данных, различаются, но можно с уверенностью сказать, что этот шаг занимает значительную часть всего времени, затрачиваемого на работу с данными. В 2014 г. New York Times сообщила[1], что специалисты по данным тратят от 50% до 80% своего времени на очистку и обработку данных. Опрос, проведенный компанией CrowdFlower в 2016 г., показал[2], что специалисты по данным тратят 60% своего времени на очистку и систематизацию данных, чтобы подготовить их для анализа или моделирования. Подготовка данных — настолько распространенная задача, что для ее описания появились новые термины, например перебирание данных (data munging[3]), обработка данных (data wrangling) и подготовка данных (data prep). Является ли эта работа по подготовке данных простой примитивной задачей или это важная часть процесса?

Подготовка данных упрощается, если для набора данных есть *словарь данных* — документ или репозиторий с четкими описанием полей, возможных значений, как данные были собраны и как они связаны с другими данными. К сожалению, чаще всего такого словаря нет. Документация, как правило, имеет низкий приоритет даже для людей, которые понимают ее ценность, или она быстро устаревает по мере добавления новых полей и таблиц или изменения способа заполнения данных. Профилирование данных позволяет заполнить многие элементы словаря данных, поэтому, если в вашей организации уже есть такой словарь, самое время открыть его и внести свой вклад в его формирование. Если словаря данных пока нет, подумайте о том, чтобы создать его! Это один из самых ценных подарков, который вы можете сделать своей команде и самому себе в будущем. Актуальный словарь данных позволяет ускорить процесс профилирования данных, опираясь на выполненное ранее профилирование и не повторяя его. Это также повысит качество результатов вашего анализа, т. к. вы сможете убедиться, что правильно использовали поля и применили корректные фильтры.

---

[1] **https://www.nytimes.com/2014/08/18/technology/for-big-data-scientists-hurdle-to-insights-is-janitor-work.html**

[2] **https://www.forbes.com/sites/gilpress/2016/03/23/data-preparation-most-time-consuming-least-enjoyable-data-science-task-survey-says/**

[3] Mung — это аббревиатура от «Mash Until No Good» — «перемешивай пока не испортишь», чем я, конечно, иногда и занимаюсь.

Даже если словарь данных уже существует, вам, вероятно, все равно придется выполнять подготовку данных в рамках вашего анализа. В этой главе я вначале рассмотрю типы данных, с которыми вы, скорее всего, столкнетесь. Затем разберу структуру SQL-запроса. Далее я расскажу о профилировании данных, чтобы узнать их содержимое и проверить их качество. После этого я приведу некоторые полезные способы очистки данных, чтобы решать любые проблемы с качеством. Наконец, я расскажу о некоторых методах структурирования данных, чтобы получить именно те столбцы и строки, которые необходимы для дальнейшего анализа.

# 2.1. Типы данных

Данные являются основой анализа, и все данные можно отнести к какому-то стандартному типу для базы данных, а также к одной или нескольким категориям данных. Четкое представление о различных формах, которые могут принимать данные, поможет вам стать более эффективным аналитиком данных. Я начну с типов данных, наиболее часто встречающихся при анализе. Затем я расскажу о некоторых способах категоризации данных, которые могут помочь нам понять источник данных, их качество и возможные области применения.

## Типы в базах данных

Все поля в таблицах баз данных имеют определенный тип данных. Большинство баз имеют полную документацию по типам, которые они поддерживают, и это всегда полезный источник любой необходимой информации, в дополнение к тому, что описано здесь. Вам не обязательно быть экспертом и разбираться в нюансах типов данных для проведения хорошего анализа, но далее в книге мы столкнемся с ситуациями, в которых важно учитывать тип данных, поэтому в этом разделе рассмотрим необходимый базис. Основными типами данных являются строковые, числовые, логические типы и дата/время, которые перечислены в табл. 2.1. Типы данных в этой таблице относятся к базе Postgres, но они одинаковы для большинства баз данных.

**Таблица 2.1.** Основные типы данных

| Тип | Название | Описание |
|---|---|---|
| Строковый | CHAR, VARCHAR | Содержит строки. CHAR всегда имеет фиксированную длину, тогда как VARCHAR имеет переменную длину до некоторого максимального размера (например, до 256 символов). |
| | TEXT, BLOB | Содержит более длинные строки, которые не помещаются в VARCHAR. В этих полях может храниться описание или произвольный текст, введенный участниками опроса. |

**Таблица 2.1** (окончание)

| Тип | Название | Описание |
|---|---|---|
| Числовой | INT, SMALLINT, BIGINT | Содержит целые числа. В некоторых базах данных есть типы SMALLINT и/или BIGINT. Тип SMALLINT можно использовать для чисел с небольшим количеством цифр, и он занимает меньше памяти, чем обычный INT. А BIGINT позволяет хранить числа с бóльшим количеством цифр, чем INT, но он и занимает больше памяти. |
| | FLOAT, DOUBLE, DECIMAL | Содержит десятичные числа, иногда с указанным количеством десятичных разрядов. |
| Логический | BOOLEAN | Содержит значения TRUE (ИСТИНА) или FALSE (ЛОЖЬ). |
| Дата/время | DATETIME, TIMESTAMP | Содержит дату и время. Обычно в формате YYYY-MM-DD hh:mi:ss, где:<br>YYYY — четырехзначный год,<br>MM — двузначный номер месяца,<br>DD — двузначный день,<br>hh — двузначный час (обычно от 0 до 23),<br>mi — две цифры для минут,<br>ss — две цифры для секунд.<br>Некоторые базы данных хранят временные метки без часового пояса, тогда как другие базы имеют специальные типы для временных меток с часовыми поясами и без них. |
| | TIME | Содержит только время. |

Строковые типы данных являются наиболее универсальными. Они могут содержать буквы, цифры и специальные символы, в том числе и невидимые символы, такие как табуляция и перевод строки. Строковые поля могут быть определены с фиксированным или переменным количеством символов. Например, для записи аббревиатур для штатов США можно определить поле CHAR из двух символов, а поле, в котором будут храниться полные названия штатов, должно быть VARCHAR для переменного количества символов. Для хранения очень длинных строк можно определить поля типа TEXT, CLOB (Character Large Object, большой символьный объект) или BLOB (Binary Large Object, большой двоичный объект, который может содержать дополнительные типы данных, например изображения), в зависимости от базы данных, и, хотя они могут занимать много памяти, используются достаточно редко. Если при загрузке данных приходят строки, которые слишком длинны для указанного типа данных, они будут или обрезаны, или пропущены целиком. В SQL есть ряд строковых функций, которые мы будем использовать для различных целей анализа.

Числовые типы данных хранят числа, как положительные, так и отрицательные. К таким полям можно применять математические функции и арифметические операторы. К числовым типам относятся целочисленный `INT`, а также `FLOAT`, `DOUBLE` и `DECIMAL`, допускающие десятичные разряды. Целочисленные типы данных используются чаще, потому что занимают меньше памяти, чем их десятичные друзья. В некоторых базах данных, таких как Postgres, деление целых чисел вернет целое число, а не значение с десятичными разрядами, как можно было бы ожидать. В *разд. 2.5* мы рассмотрим преобразование типов данных для получения правильных результатов.

Логический тип данных называется `BOOLEAN`. Он может принимать только два значения, `TRUE` или `FALSE`, и является очень экономным способом хранения информации для тех случаев, когда это применимо. Операции сравнения двух полей также возвращают значение типа `BOOLEAN`. Этот тип данных часто используется для *флагов* — полей, которые указывают на наличие или отсутствие какого-то свойства. Например, таблица, в которой хранятся данные об электронных письмах, может иметь поле `BOOLEAN has_opened`, означающее, открывалось ли письмо.

К типам даты/времени относятся `DATE`, `TIMESTAMP` и `TIME`. Когда это возможно, данные о дате и времени должны записываться в поле одного из этих типов, поскольку в SQL есть ряд полезных функций для работы с ними. Временные метки и даты очень распространены в базах данных и имеют важное значение для многих видов анализа, особенно для анализа временных рядов (будет рассматриваться в *гл. 3*) и для когортного анализа (в *гл. 4*). В *гл. 3* обсуждаются также форматирование значений даты и времени, их преобразования и вычисления.

Другие типы данных, такие как JSON или географические типы, поддерживаются некоторыми базами данных, но не всеми. Я не буду здесь подробно останавливаться на них, т. к. они выходят за рамки этой книги. Но такие типы данных являются признаком того, что SQL продолжает развиваться для решения более новых задач анализа.

Помимо типов данных, существует несколько концептуальных способов категоризации данных. Это может влиять на то, как данные будут храниться и как мы будем их анализировать.

## Структурированные и неструктурированные данные

Данные часто характеризуются как структурированные или неструктурированные, а иногда и как полуструктурированные. Большинство баз данных предназначены для работы со *структурированными данными*, где каждый атрибут хранится в отдельном столбце, а экземпляр каждой сущности представлен в виде отдельной строки. Сначала создается модель данных, а затем данные записываются в соответствии с этой моделью. Например, в таблице адресов могут быть поля для региона, города, улицы и почтового индекса. Каждая строка будет содержать адрес конкретного клиента. Каждое поле имеет свой тип данных и позволяет вводить данные только этого типа. Когда структурированные данные записываются в таблицу, для

каждого поля проверяется, что вставляемые данные соответствуют типу поля. Структурированные данные легко запросить с помощью SQL.

*Неструктурированные данные* противоположны структурированным данным. У них нет заранее определенной структуры, модели или типов данных. Неструктурированные данные часто представляют собой «все остальное», что не относится к базе данных. Документы, электронные письма и веб-страницы представляют собой неструктурированные данные, как и фотографии, изображения, видео- и аудиофайлы. Они не соответствуют стандартным типам данных, поэтому реляционным базам труднее хранить их эффективно и сложнее использовать в запросах SQL. Поэтому неструктурированные данные часто хранятся вне реляционных баз данных. Это позволяет быстро их загружать, но отсутствие верификации таких данных может привести к ухудшению их качества. Как упоминалось в *гл. 1*, технологии продолжают развиваться, и разрабатываются новые инструменты, позволяющие выполнять SQL-запросы ко многим типам неструктурированных данных.

*Полуструктурированные* (*слабоструктурированные*) *данные* находятся между этими двумя категориями. Многие неструктурированные данные все-таки имеют какую-то структуру, которую мы можем использовать. Например, электронные письма содержат адреса отправителя и получателя, тему, основной текст и время отправления, которые можно хранить отдельно в модели данных с такими полями. Метаданные (данные о данных) можно извлечь и из файлов других типов и сохранить для анализа. Например, музыкальные аудиофайлы могут иметь теги с исполнителем, названием песни, жанром и продолжительностью. Кроме того, SQL часто используется для синтаксического разбора произвольного текста или извлечения из него структурированных данных любым другим способом для последующего анализа. Мы увидим некоторые примеры на эту тему в *гл. 5*.

## Количественные и качественные данные

*Количественные данные* являются числовыми. Они измеряют людей, предметы и события. Количественные данные могут включать в себя дескрипторы, например информация о клиенте, тип продукта или конфигурация устройства, а также численную информацию, например цена, количество или продолжительность посещения. Для таких данных можно вычислить общее количество, сумму, среднее значение или применить другие числовые функции. В наши дни количественные данные часто генерируются компьютерами, но это не всегда так. Рост, вес и артериальное давление, записанные на бумажном бланке при приеме пациента, тоже являются количественными данными, как и оценки учащихся, введенные учителем в электронную таблицу.

*Качественные данные* обычно представляются в текстовом виде и содержат мнения, чувства или описания, которые точно не являются количественными. Уровни температуры и влажности являются количественными данными, а такие описания, как «жарко и влажно», — качественными. Цена, которую покупатель заплатил за товар, является количественной, а понравился он покупателю или не нравится —

это качественные данные. Отзывы на сервис, запросы в службу поддержки клиентов и посты в социальных сетях являются качественными данными. Есть целые профессии, которые имеют дело с такими данными. В контексте анализа данных мы обычно пытаемся количественно оценить качество. Одним из методов является выделение ключевых слов и словосочетаний и подсчет количества их появлений. Мы рассмотрим это более подробно, когда будем изучать анализ текста в *гл. 5*. Еще одним методом является анализ тональности текста, когда на основе структуры языка выполняется интерпретация значений используемых слов в дополнение к их частоте. Предложения или другие фрагменты текста могут быть оценены по их тональности: позитивные или негативные, а затем можно посчитать их количество или среднее значение, чтобы получить обобщенную информацию, которую трудно получить иными способами. Были достигнуты впечатляющие успехи в области обработки естественного языка (natural language processing, NLP), хотя большая часть этих результатов получена с помощью таких простых инструментов, как Python.

## Первичные, вторичные и третичные данные

*Первичные данные* собираются самой организацией. Для этого могут использоваться логи серверов, базы данных, которые отслеживают транзакции и хранят информацию о клиентах, или другие системы, которые созданы и контролируются самой организацией и генерируют данные, представляющие интерес для анализа. Поскольку эти системы были реализованы собственными силами организации, обычно можно найти тех людей, которые их создавали, и узнать у них, как именно генерируются данные. Аналитики данных также могут влиять на то, как создаются и хранятся определенные фрагменты данных, или контролировать это, особенно если ошибки в реализации приводят к низкому качеству данных.

*Вторичные данные* поступают от поставщиков, которые предоставляют услуги или выполняют бизнес-функции для организации. Часто это SaaS-решения (software as a service, программное обеспечение как услуга). Типичными примерами являются CRM-системы, инструменты для автоматизации email-маркетинга, программное обеспечение для электронной коммерции, а также веб- и мобильные трекеры. Такие данные аналогичны первичным данным, поскольку они относятся к самой организации, и созданы ее сотрудниками и клиентами. Однако код, генерирующий и хранящий данные, и модель данных контролируются извне, и аналитик данных обычно не может повлиять на эти аспекты. Вторичные данные чаще всего импортируются в хранилище данных организации для анализа. Это может быть выполнено с помощью специально разработанного кода, ETL-коннекторов или с помощью поставщиков SaaS, которые предлагают интеграцию данных.

Многие поставщики SaaS предоставляют некоторый функционал для создания отчетности, поэтому может возникнуть вопрос, стоит ли выполнять копирование данных в свое хранилище. Отдел, который будет использовать этот функционал, может счесть их отчетность достаточной, например служба поддержки клиентов, которая составляет отчеты о выполненных запросах при помощи своего программного

обеспечения. С другой стороны, информация о работе службы поддержки клиентов может стать важным вкладом в стратегию удержания клиентов, что потребует интеграции этих данных в хранилище организации с данными о продажах и отказах. Есть хорошее практическое правило при принятии решения об импорте данных из определенного источника: если данные будут создавать ценность в сочетании с данными из других систем, импортируйте их; если нет, подождите, пока не появится веская причина для выполнения этой работы.

*Третичные данные* могут быть приобретены или получены из бесплатных источников, например из публикаций правительственных органов. Если данные не были собраны специально для организации, аналитики обычно не могут контролировать их формат, частоту и качество. Этим данным часто не хватает детализации, в отличие от первичных и вторичных данных. Например, в большинстве сторонних источников нет данных на уровне пользователя, вместо этого данные приходится объединять с первичными данными по почтовому индексу или городу, или на более высоком уровне. Тем не менее, третичные данные могут содержать уникальную и полезную информацию, такую как модели совокупных расходов, демографические показатели и рыночные тенденции, собрать которую иным способом было бы очень дорого или невозможно.

## Разреженные данные

*Разреженные данные* возникают, когда в большом наборе пустой или неважной информации содержится лишь небольшое количество нужной информации. Разреженные данные могут содержать множество `null`-значений и только несколько значений в определенном столбце. Значение `null` отличается от значения 0 и означает *отсутствие* данных (мы разберем это подробнее в *разд. 2.5*). Разреженные данные могут возникать, когда события происходят редко, например ошибки программного обеспечения или покупки товаров из самого конца длинного каталога товаров. Это также может произойти в первые дни запуска функционала или продукта, когда доступ есть только у тестировщиков или пользователей бета-версии. JSON — это один из форматов, который был разработан для записи и хранения разреженных данных, поскольку он сохраняет только те данные, которые присутствуют, а остальное игнорирует. Это отличается от строчной базы данных, которая выделяет память для поля, даже если в него не записано никакого значения.

Разреженные данные могут быть проблематичными для анализа. Когда события происходят редко, тенденции могут быть не заметны, а корреляции трудно отличить от случайных отклонений. Имеет смысл выполнить профилирование ваших данных, как описано далее в этой главе, чтобы понять, являются ли они разреженными и где именно. Некоторые способы решения этой проблемы заключаются в группировке редких событий или элементов в отдельные категории, в полном исключении разреженных данных или временного интервала из анализа или в выводе описательной статистики вместе с предупреждением о том, что тенденции не достоверные.

Существуют различные типы данных и множество способов их категоризации, при этом многие категории перекрываются и не являются взаимоисключающими. Разбираться в типах данных нужно не только для написания хорошего SQL, но и для принятия правильных решений о том, как лучше их анализировать. Вы не всегда можете заранее знать типы данных, поэтому профилирование данных очень важно. Прежде чем мы перейдем к нему и к нашим первым примерам кода, я дам краткий обзор структуры SQL-запросов.

## 2.2. Структура SQL-запроса

Все SQL-запросы имеют общие разделы и синтаксис, хотя их можно комбинировать почти бесконечное число раз для достижения целей анализа. Предполагается, что у вас есть некоторое знание SQL, но я все равно рассмотрю основы, чтобы у нас был общий базис для последующих примеров кода.

После ключевого слова `SELECT` определяются столбцы, которые будут возвращены запросом. Для каждого выражения в операторе `SELECT` возвращается один столбец, выражения разделяются запятыми. Выражение может быть полем таблицы, агрегатной функцией, такой как `sum`, или любым вычислением с использованием, например, операторов `CASE`, преобразований типов и различных функций, которые будут обсуждаться далее в этой главе и на протяжении всей книги.

В разделе `FROM` определены таблицы, которые используются в выражениях оператора `SELECT`. Здесь «таблицей» может быть таблица базы данных, представление (view — тип сохраненного запроса, который в целом функционирует как таблица) или подзапрос. Подзапрос сам по себе является запросом, заключенным в круглые скобки, и запрос, в котором он используется, обрабатывает его результат как любую другую таблицу. Запрос может ссылаться на несколько таблиц в разделе `FROM`, но нужно использовать один из типов соединений `JOIN` вместе с условием, определяющим, как связаны таблицы. В условии `JOIN` обычно указывается равенство между полями таблиц, например `orders.customer_id = customers.customer_id`. Условия `JOIN` могут включать в себя несколько полей каждой таблицы, а также могут содержать неравенства или диапазоны значений, например диапазон дат. На протяжении всей книги мы увидим множество разных условий `JOIN`, которые позволяют достичь конкретных целей анализа. Внутреннее соединение `INNER JOIN` возвращает только те записи из обеих таблиц, которые подходят по условию соединения. Левое соединение `LEFT JOIN` возвращает все записи из первой таблицы, и только те записи второй таблицы, для которых выполняется условие. Правое соединение `RIGHT JOIN` возвращает все записи из второй таблицы и только те записи первой таблицы, для которых выполняется условие. Полное внешнее соединение `FULL OUTER JOIN` возвращает все записи из обеих таблиц. В результате соединения может получиться декартово произведение, когда каждой записи в первой таблице соответствует более одной записи во второй таблице. Как правило, следует избегать таких соединений, хотя существуют некоторые особые случаи, когда мы будем намеренно их использовать,

например при генерации данных для заполнения временного ряда. И, наконец, каждой таблице в разделе FROM можно задать *псевдоним*, т. е. дать более короткое имя, состоящее из одной или нескольких букв, на которое можно ссылаться в других разделах запроса. Псевдонимы избавляют авторов запросов от необходимости многократно указывать длинные имена таблиц и упрощают чтение запросов.

Хотя оба соединения, LEFT JOIN и RIGHT JOIN, могут использоваться в одном и том же запросе, намного проще следить за логикой, когда вы придерживаетесь только одного типа соединений. На практике LEFT JOIN используется гораздо чаще, чем RIGHT JOIN.

В разделе WHERE указываются ограничения или фильтры, необходимые для исключения или удаления строк из результатов. Этот раздел является необязательным.

Раздел GROUP BY требуется, если оператор SELECT содержит агрегатные выражения и хотя бы одно неагрегированное поле. Легко запомнить, что в GROUP BY должны быть перечислены все поля, которые не используются в агрегатных функциях. В большинстве баз данных существуют два способа перечисления полей в GROUP BY: либо по имени поля, либо по позиции, например 1, 2, 3. Некоторые люди предпочитают указывать имена полей, и в Microsoft SQL Server используется только этот способ. Но я предпочитаю позиционную нотацию, особенно когда поля в GROUP BY содержат сложные выражения или когда у меня много правок. В этой книге будет использоваться в основном позиционная нотация.

**Как не убить вашу базу данных: *LIMIT* и выборка**

Таблицы базы данных могут быть очень большими и содержать миллионы или даже миллиарды записей. Выполнение запросов по всем этим записям может привести к проблемам, вплоть до сбоя базы данных. Чтобы избежать блокировки базы данных и неприятных разговоров с ее администратором, рекомендуется ограничивать возвращаемые результаты при выполнении запросов для профилирования или тестирования. Ограничение с помощью ключевого слова LIMIT и частичная выборка (sampling) — это два метода, которые должны войти в ваш инструментарий.

Предложение LIMIT добавляется в самый конец запроса или подзапроса и за ним указывается любое положительное целое число:

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

При использовании LIMIT в подзапросе ограничение будет применено только на этом шаге, и этот урезанный результат будет использоваться внешним запросом:

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

Microsoft SQL Server не поддерживает LIMIT, но аналогичный результат можно получить с помощью ключевого слова TOP:

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

Частичную выборку можно получить с помощью функции, примененной к полю идентификатора (ID). Например, функция mod возвращает остаток от деления одного целого числа на другое. Если поле идентификатора целочисленное, можно использовать эту функцию, чтобы найти последние одну, две или более цифр и отфильтровать результат:

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

Это вернет каждую запись, для которой две последние цифры идентификатора равны 06, что должно составить около 1% от общего числа записей. Если поле идентификатора является буквенно-цифровым, вы можете использовать функцию right, чтобы найти определенное количество символов в конце строки:

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

Это вернет каждую запись, идентификатор которой заканчивается на символ 'B', что будет составлять около 3% от общего количества записей (только в том случае, если все буквы и цифры одинаково распространены в идентификаторах — это предположение, которое хорошо бы проверить заранее).

Дополнительное ограничение с помощью LIMIT также ускорит вашу работу, но имейте в виду, что полученные подмножества данных могут не содержать всех вариантов значений и граничных случаев, существующих в полном наборе данных. Не забудьте убрать LIMIT или условие для выборки, прежде чем запускать окончательный анализ или отчет с вашим запросом, иначе вы получите забавный результат!

Приведенная информация охватывает основы структуры SQL-запросов. В *гл. 8* будут более подробно рассмотрены каждый из этих разделов и несколько дополнительных, которые используются реже, но встречаются в этой книге, а также порядок выполнения этих разделов. Теперь, когда у нас есть этот базис, мы можем перейти к одному из самых важных этапов анализа — профилированию данных.

## 2.3. Профилирование: распределения

Профилирование — это первое, что я делаю, начиная работать с новым набором данных. Я смотрю, как данные организованы в схемы и таблицы. Я проверяю названия таблиц, чтобы узнать, к каким сущностям они относятся, например к клиентам, сессиям или заказам. Я просматриваю имена столбцов в нескольких таблицах и начинаю строить ментальную модель того, как таблицы связаны друг с другом. Например, среди таблиц может быть таблица `order_detail` с позициями заказа, которая связана с таблицей `order` через `order_id`, а таблица `order` связана с таблицей `customer` через `customer_id`. Если есть словарь данных, я просматриваю его и сравниваю с теми данными, которые вижу в таблицах.

В таблицах обычно представлена деятельность компании или некоторая ее часть, поэтому я стараюсь понять, какая предметная область охвачена — электронная коммерция, маркетинг или движение товаров. Работать с данными будет проще, если знать, как они были сгенерированы. Профилирование может помочь с этим и подсказать, какие вопросы нужно задавать источнику данных — людям внутри или за пределами организации, ответственным за сбор или создание данных. Даже когда вы сами отвечаете за сбор данных, профилирование тоже будет полезным.

Еще одна деталь, которую я проверяю, — это то, как представлена история изменений, если она вообще как-то представлена. Наборы данных, являющиеся копиями производственных баз данных, могут, например, не содержать предыдущих значений адресов клиентов или статусов заказов, в то время как хорошо сконструированное хранилище данных может иметь ежедневные снимки (snapshots) меняющихся полей данных.

Профилирование данных связано с концепцией *анализа результатов наблюдений* (Exploratory Data Analysis, EDA), введенной Джоном Тьюки (John Tukey). В своей одноименной книге[4] Тьюки описывает, как анализировать наборы данных, вычисляя различные итоги и визуализируя результаты. Данный анализ включает в себя построение распределений данных, в том числе диаграммы «стебель-листья», диаграммы размаха и гистограммы.

Проверив несколько выборок данных, я начинаю анализ распределений данных. Распределения позволяют определить диапазон значений, которые присутствуют в

[4] *Тьюки Д. В.* Анализ результатов наблюдений. М.: Мир, 1981.

данных, как часто они встречаются, есть ли null-значения и встречаются ли отрицательные значения одновременно с положительными. Распределения могут быть получены для непрерывных или категориальных данных и называются *частотами*. В этом разделе мы рассмотрим, как создавать гистограммы, как биннинг может помочь нам определить распределение непрерывных данных и как использовать *n*-тили для получения более точных сведений о распределениях.

## Гистограммы и частоты

Один из лучших способов познакомиться с набором данных и его полями — это проверить частоту значений в каждом поле. Это также полезно, когда у вас возникают вопросы о том, возможны ли определенные значения, или когда вы замечаете неожиданное значение и хотите узнать, как часто оно попадается. Проверить частоту можно для любого типа данных, включая строки, числа, даты и логические значения. Запрос для получения частоты также является отличным способом обнаружения разреженных данных.

Сам запрос очень прост. Количество строк можно найти с помощью функции count(*), а профилируемое поле нужно указать в GROUP BY. Например, мы можем проверить частоту каждого вида фруктов fruit в вымышленной таблице fruit_inventory:

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

При использовании count стоит потратить минуту на то, чтобы подумать, могут ли в наборе данных быть какие-либо дублированные записи. Вы можете использовать count(*), если вам просто нужно количество записей, но используйте count distinct, чтобы узнать, сколько уникальных записей существует в наборе.

*Частотная диаграмма* — это способ визуализации того, сколько раз какое-то значение встречается в наборе данных. Профилируемое поле обычно откладывается по оси *x*, а количество наблюдений — по оси *y*. На рис. 2.1 показан пример такой диаграммы для фруктов из нашего запроса. Обратите внимание, что это категориальные данные без какой-либо внутренней сортировки. Частотные диаграммы также можно рисовать горизонтально, что удобно при длинных значениях.

*Гистограмма* — это способ визуализации распределения числовых значений в наборе данных, с которым хорошо знакомы те, у кого есть опыт работы со статистикой. Простейшая гистограмма может показывать распределение группы клиентов по возрасту. Представьте, что у нас есть таблица клиентов customers, которая содержит имена, дату регистрации, возраст и прочие атрибуты. Чтобы построить гис-

тограмму по возрасту, сгруппируйте с помощью `GROUP BY` по числовому полю `age` и посчитайте количество `customer_id` с помощью `count`:

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

Результаты нашего условного распределения по возрасту представлены на рис. 2.2.

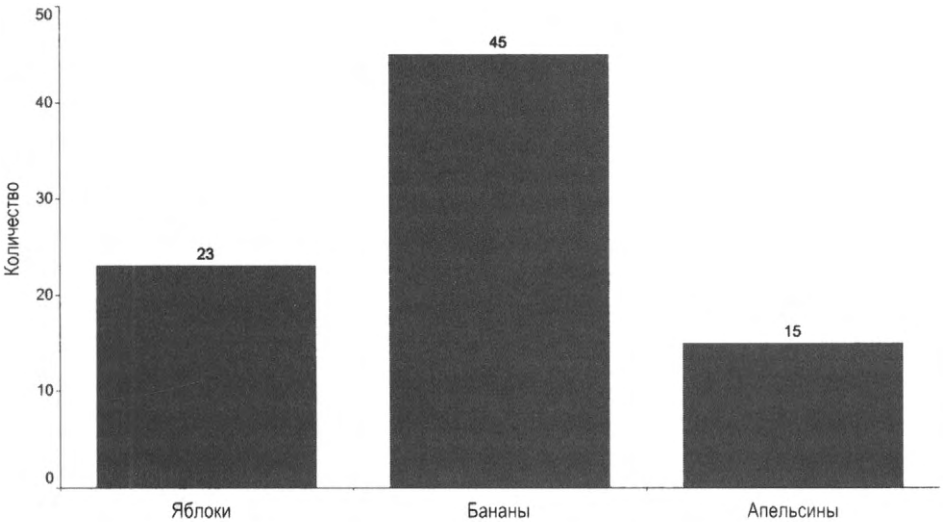


**Рис. 2.1.** Частотная диаграмма наличия фруктов

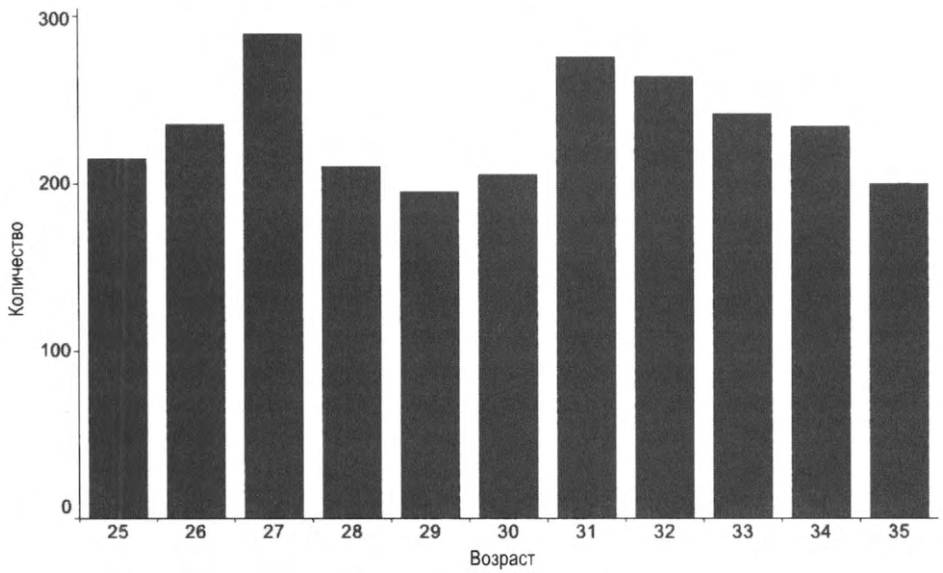


**Рис. 2.2.** Распределение клиентов по возрасту

Еще один пример, который я люблю приводить на собеседованиях, включает в себя агрегирование с последующим подсчетом частоты. Я даю кандидатам условную таблицу заказов `orders`, которая содержит дату, идентификатор клиента, идентификатор заказа и сумму, а затем прошу их написать SQL-запрос, который возвращает распределение клиентов по количеству заказов. Это не может быть решено с помощью простого запроса; для этого требуется промежуточный шаг с агрегированием, который можно выполнить с помощью подзапроса. Сначала в подзапросе нужно подсчитать количество заказов, размещенных каждым `customer_id`. Внешний запрос использует группировку по количеству заказов `num_orders` и подсчитывает количество клиентов:

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

Этот метод профилирования можно применять всякий раз, когда вам нужно увидеть, как часто определенные объекты или атрибуты встречаются в данных. В этих примерах использовалась функция `count`, но для создания гистограмм могут использоваться и другие агрегации (`sum`, `avg`, `min`, и `max`). Например, мы можем профилировать клиентов по сумме всех их заказов, по среднему объему заказа, по минимальной или максимальной (самой последней) дате заказа.

## Биннинг

*Биннинг* (от англ. *bin* — корзина, ящик) применяется при работе с непрерывными данными. Вместо того, чтобы подсчитывать количество наблюдений или записей для каждого дискретного значения, используются диапазоны значений, которые называются *ячейками*, или *интервалами*. Затем подсчитывается количество записей, попадающих в каждый интервал. Ячейки могут быть переменного или фиксированного размера, в зависимости от того, группируются ли данные по какому-то важному для организации принципу, имеют примерно одинаковую ширину или содержат приблизительно одинаковое количество записей. Ячейки можно создавать с помощью операторов `CASE`, округления или логарифмирования.

Оператор `CASE` позволяет использовать условную логику. Этот оператор очень гибкий, и мы будем обращаться к нему на протяжении всей книги при профилировании данных, очистке, анализе текста и т. д. Базовая структура оператора `CASE` выглядит так:

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

```
...
else return_value_default
end
```

Условие WHEN может быть равенством, неравенством или любым логическим выражением. Возвращаемое значение THEN может быть константой, выражением или полем в таблице. Можно использовать любое количество условий, но оператор прекратит выполнение и вернет результат при первом же случае, когда условие будет равно TRUE. Значение ELSE можно использовать в качестве значения по умолчанию, если ни одно условие не выполнено, оно также может быть константой, выражением или полем. Предложение ELSE является необязательным, и, если оно не добавлено и ни одно из условий не выполнилось, вернется значение null. Операторы CASE могут быть вложены друг в друга, когда в качестве возвращаемого значения используется другой оператор CASE.

Все возвращаемые значения, следующие за THEN, должны быть одного типа (строки, числа, логические значения и т. д.), иначе вы получите ошибку. Если вы столкнетесь с таким случаем, вам понадобится приведение к общему типу данных, например к строке.

Оператор CASE — это гибкий способ управления количеством ячеек, диапазоном значений, попадающих в каждую ячейку, и их именованием. Я нахожу его особенно полезным, когда есть длинный хвост очень маленьких или очень больших значений, которые я хочу сгруппировать вместе, чтобы не было пустых ячеек в какой-то части распределения. Некоторые диапазоны имеют бизнес-значение, которое необходимо применить и для данных. Многие компании B2B делят своих клиентов на категории «предприятие» и «малый и средний бизнес» (в зависимости от количества сотрудников или прибыли), потому что они имеют разные шаблоны поведения. Представьте, например, что мы рассматриваем предложения по скидкам на доставку и хотим узнать, скольких клиентов это может затронуть. Мы можем сгруппировать клиентов по сумме заказа order_amount в три ячейки, используя оператор 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
;
```

Такие ячейки произвольного размера очень удобны, но в некоторых случаях для анализа больше подходят ячейки фиксированного размера. Их можно получить несколькими способами, в том числе с помощью округления, логарифмирования и *n*-тилей. Для создания ячеек одинаковой ширины проще всего использовать округление. Округление снижает точность значений, и мы обычно понимаем округление как уменьшение количества знаков после запятой или полное их отбрасывание при округлении до ближайшего целого числа. Функция `round` имеет вид:

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

Количество знаков после запятой *number_of_decimal_places* может быть и отрицательным числом, что позволяет с помощью этой функции округлять до ближайших десятков, сотен, тысяч и т. д. В табл. 2.2 показаны примеры округлений с количеством знаков после запятой от −3 до 2.

**Таблица 2.2.** Округление числа 123 456.789 с различным количеством знаков после запятой

| Количество знаков | Функция | Результат |
|---|---|---|
| 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 |

Сгруппировать клиентов по ячейкам одинакового размера можно, например, так:

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

Логарифмирование — это еще один способ создания ячеек, особенно для наборов данных, в которых самые большие значения на несколько порядков больше, чем самые маленькие значения. Распределение благосостояния домохозяйств, количества посетителей веб-сайта по различным страницам или силы землетрясений — все это примеры таких случаев. Хотя логарифмы и не создают ячейки одинаковой ширины, они создают ячейки, размеры которых увеличиваются по определенному шаблону. Чтобы освежить вашу память, напомню, что десятичный логарифм — это показатель степени *exponent*, в которую нужно возвести 10, чтобы получить число *number*:

```
log(number) = exponent
```

В этом случае число 10 называется основанием логарифма, и обычно оно используется по умолчанию в базах данных. Но формально основанием логарифма может быть любое число. В табл. 2.3 показаны логарифмы для нескольких степеней числа 10.

**Таблица 2.3.** Результаты выполнения функции `log` по степеням 10

| Функция | Результат |
|---|---|
| `log(1)` | 0 |
| `log(10)` | 1 |
| `log(100)` | 2 |
| `log(1000)` | 3 |
| `log(10000)` | 4 |

В функции `log` аргументом может быть константа или поле:

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

Функцию `log` можно применить к любому положительному значению, а не только к кратному 10. Однако логарифмическая функция не работает, когда значения меньше или равны 0, тогда возвращается `null` или ошибка, в зависимости от базы данных.

## *n*-тили

Вы, скорее всего, уже знакомы с *медианой* — серединным значением набора данных. Это также значение 50-го процентиля. Половина всех значений в наборе больше медианы, а другая половина — меньше. Для *квартилей* мы вычисляем значения 25-го и 75-го процентилей. Четверть всех значений меньше 25-го процентиля, а три четверти — больше его. Три четверти всех значений меньше 75-го процентиля, а одна четверть — больше. *Децили* разбивают набор данных на 10 равных частей. Обобщая эту концепцию, можно сказать, что *n-тили* позволяют вычислить любой процентиль для набора данных: 27-й процентиль, 52-й процентиль и т. д.

### Оконные функции

Функции для вычисления *n*-тилей являются частью группы функций SQL, называемых *оконными* или аналитическими функциями. В отличие от большинства функций SQL, которые работают только с текущей строкой, оконные функции выполняют вычисления, охватывающие несколько строк. Оконные функции имеют специальный синтаксис, включающий в себя имя функции и раздел `OVER`,

определяющий группировку и сортировку строк, с которыми нужно работать. Общий формат оконной функции такой:

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

Функция *function* может быть любой из обычных агрегатных (count, sum, avg, min, max), а также могут использоваться некоторые специальные функции, такие как rank, first_value и ntile. В предложении PARTITION BY можно перечислить несколько полей или не указать ни одного. Если поле не указано, функция работает со всей таблицей, но, если перечислены поля, по ним делается группировка на секции, и функция будет выполняться только в пределах каждой секции. Например, мы можем использовать PARTITION BY customer_id для вычисления по всем записям для каждого клиента, сбрасывая вычисление заново для следующего клиента. Предложение ORDER BY определяет порядок сортировки строк для функции, когда это влияет на вычисление. Например, чтобы ранжировать клиентов с помощью rank, нам нужно указать поле, по которому их нужно упорядочить (например, по количеству заказов).

Все основные типы баз данных имеют оконные функции, за исключением версий MySQL до 8.0.2. Я буду использовать эти полезные функции на протяжении всей книги вместе с дополнительными пояснениями того, как они работают и как правильно задавать аргументы.

Многие базы данных имеют отдельную встроенную функцию median, но в основном все полагаются на более общую функцию для вычисления любых *n*-тилей. Эта функция является оконной, выполняющей вычисления по диапазону строк, чтобы вернуть значение для одной строки. Она принимает аргумент, указывающий количество ячеек для группировки данных, и может содержать необязательные предложения PARTITION BY и ORDER BY:

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

В качестве примера представьте, что у нас есть 12 транзакций со значениями order_amount, равными $19.99, $9.99, $59.99, $11.99, $23.49, $55.98, $12.99, $99.99, $14.99, $34.99, $4.99 и $89.99. Выполнение функции ntile для десяти интервалов отсортирует записи по order_amount и назначит интервал от 1 до 10 для каждой записи:

```
order_amount  ntile
------------  -----
4.99          1
9.99          1
11.99         2
12.99         2
14.99         3
19.99         4
23.49         5
34.99         6
```

```
55.98          7
59.99          8
89.99          9
9999           10
```

На практике это можно использовать для разбивки записей по интервалам: вычислить сначала `ntile` для каждой строки в подзапросе, а затем во внешнем запросе определить `min` и `max` для нахождения верхних и нижних границ каждого диапазона значений:

```
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
;
```

Схожим образом можно использовать функцию `percent_rank`. Вместо того чтобы возвращать интервалы, в которые попадают данные, `percent_rank` возвращает процентиль. Сама функция не имеет аргументов, но требует наличия раздела `OVER` и круглых скобок, может содержать необязательные `PARTITION BY` и `ORDER BY`:

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

Хотя функция `percent_rank` и не так удобна для биннинга, как `ntile`, ее можно использовать для создания непрерывного распределения или выходных данных для отчетов или дальнейшего анализа. И `ntile`, и `percent_rank` могут быть очень дорогими при вычислениях на больших наборах данных, поскольку они требуют сортировки всех строк. Помогает фильтрация таблицы, чтобы функция работала только с теми строками, которые вам нужны. В некоторых базах данных реализованы приблизительные версии функций, которые работают быстрее и обычно возвращают приемлемые результаты, когда не требуется абсолютной точности. Мы рассмотрим дополнительные варианты использования *n*-тилей в *разд. 6.3* при обсуждении поиска аномалий.

Во многих случаях нет какого-то единственно правильного или объективно лучшего способа построить распределение данных. У аналитиков есть широкая свобода действий при использовании описанных методов, чтобы разобраться в данных и представить их другим людям. Тем не менее, специалисты по данным должны руководствоваться здравым смыслом и этическими принципами каждый раз при обмене распределениями конфиденциальных данных.

## 2.4. Профилирование: качество данных

Качество данных критически важно, когда речь идет о хорошем анализе. Хотя это может казаться очевидным, но это был один из самых сложных уроков, которые я усвоила за годы работы с данными. Очень легко увлечься и сосредоточиться исключительно на обработке данных, составлении умных запросов и правильной визуализации, а затем заказчики могут отклонить все это и указать на единственное противоречие в данных. Обеспечение качества данных может быть одним из самых сложных и изнурительных этапов анализа. Фраза «мусор на входе — мусор на выходе» отражает лишь часть проблемы. Хорошие данные вместе с неправильными предположениями также могут привести к мусору на выходе.

Сравнить имеющиеся данные с реальностью или с тем, что принимается за таковую, было бы идеально, хотя это и не всегда возможно. Например, если вы работаете с копией производственной базы данных, то вы можете сравнить количество строк в обеих базах, чтобы убедиться, что все строки попали в копию базы данных. В других случаях вы можете знать сумму в долларах и количество продаж за конкретный месяц и, составив нужный запрос к базе данных, убедиться, что сумма продаж `sum` и количество записей `count` совпадают. Часто разница между результатами вашего запроса и ожидаемым значением появляется из-за того, правильно ли вы применили фильтры, например исключили ли вы отмененные заказы и тестовые учетные записи, учли ли `null`-значения и орфографические аномалии, правильно ли вы указали условия в `JOIN` для соединения таблиц.

Профилирование — это способ выявить проблемы с качеством данных на ранней стадии, до того, как они негативно повлияют на результаты и выводы, сделанные на основе этих данных. Профилирование выявляет `null`-значения, категориальные признаки, которые необходимо преобразовать, поля с несколькими значениями, которые нужно разделить, и необычные форматы даты и времени. Профилирование также может выявить пробелы и скачкообразные изменения в данных при помощи отслеживания изменений или простоев. Данные редко бывают идеальными, и очень часто только благодаря анализу выявляются проблемы с их качеством.

### Поиск дубликатов

*Дубликат* — это когда у вас есть две (или более) строки с одинаковой информацией. Дубликаты могут возникать по разным причинам. Возможно, при вводе данных была допущена ошибка, если какой-то шаг выполнялся вручную. Запись об операции могла сработать дважды. Шаг обработки мог выполниться несколько раз. Возможно, вы создали дубликаты случайно с помощью случайного `JOIN` «многие ко многим». Что бы ни было причиной, дубликаты могут серьезно испортить ваш анализ. Я помню случай в начале своей карьеры, когда я думала, что сделала замечательное открытие, пока менеджер не заметил, что на моих графиках продажи вдвое превышают фактические. Это всегда смущает, подрывает доверие, приходится пе-

ределывать заново, а иногда и кропотливо проверять код, чтобы найти проблему. Я научилась всегда проверять дубликаты по ходу работы.

К счастью, найти дубликаты в наших данных достаточно легко. Один из способов — проверить выборку, упорядочив все столбцы:

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

Этот запрос покажет, есть ли полное дублирование записей, например при получении совершенно нового набора данных, или когда вы подозреваете, что процесс генерирует дубликаты, или после возможного декартова соединения таблиц. Если в данных есть всего лишь несколько дубликатов, они могут не попасть в вашу выборку. Кроме того, просмотр всех данных в поисках дубликатов слишком утомителен. Более надежный способ поиска дубликатов состоит в том, чтобы выбрать столбцы с помощью `SELECT`, а затем подсчитать количество строк с помощью `count` (запрос может показаться знакомым из примера с гистограммой):

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

Этот запрос проверяет, имеются ли вообще какие-либо дубликаты. Если запрос возвращает 0, то все в порядке. Для более подробной информации вы можете вывести количество дубликатов (2, 3, 4 и т. д.):

```
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
;
```

Вместо подзапроса вы можете использовать оператор HAVING и только один основной запрос. Поскольку HAVING применяется после агрегации и GROUP BY, его можно использовать для фильтрации агрегированного значения:

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

Я предпочитаю использовать подзапросы, потому что считаю их очень удобными для отслеживания логики запросов. В *гл. 8* будет обсуждаться порядок вычислений и стратегии организации ваших SQL-запросов.

Для получения полной информации о том, какие записи дублируются, вы можете вывести все поля, чтобы использовать их для поиска ошибочных записей:

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

Обнаружение дубликатов — это только первый шаг, далее нужно решить, что с ними делать. Хорошо бы понять, почему в наборе данных возникли дубликаты, и, если это возможно, устранить проблему заранее. Можно ли улучшить процесс обработки данных так, чтобы уменьшить или исключить дублирование записей? Есть ли ошибка в процессе ETL? Или вы не учли отношение «один ко многим» в каком-то JOIN? Далее мы рассмотрим некоторые способы обработки и исключения дубликатов с помощью SQL.

## Исключение дубликатов с помощью *GROUP BY* и *DISTINCT*

Дубликаты иногда встречаются, и они не всегда являются результатом неверных данных. Представьте, например, что мы хотим получить список всех клиентов, которые успешно оформили покупку, чтобы мы могли отправить им купон на следующий заказ. Мы можем объединить таблицу клиентов customers с таблицей сделок transactions с помощью JOIN, что вернет только тех клиентов, которые встречаются в таблице transactions:

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

Однако это вернет по строке для каждого клиента и каждой его транзакции, а мы знаем, что есть по крайней мере несколько клиентов, которые совершали покупки более одного раза. Мы случайно создали дубликаты, но не потому, что у нас есть какая-то проблема с качеством данных, а потому, что мы не подумали о том, как избежать дублирования в запросе. К счастью, в SQL есть несколько способов, чтобы исправить это. И один из них — использовать ключевое слово `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
;
```

Другой способ — использовать `GROUP BY`. Хотя этот раздел обычно используется вместе с агрегацией, но он убирает дубликаты так же, как и `DISTINCT`. Я помню, как впервые увидела, что коллега использует `GROUP BY` без агрегации, — я даже не знала, что так можно. Я нахожу этот способ менее интуитивно понятным, чем `DISTINCT`, но результат получается тот же:

```
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
;
```

Еще один полезный способ — выполнить агрегацию, которая возвращает одну строку для каждой сущности. Хотя формально это не устранение дубликатов, результат будет аналогичным. Например, если у нас есть несколько транзакций, совершенных одним и тем же клиентом, а нам нужно вернуть одну запись для каждого клиента, мы можем найти первую (`min`) и/или самую последнюю (`max`) дату `transaction_date`:

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

Дублирующиеся данные, т. е. данные, содержащие несколько записей для каждой сущности, даже если они формально не являются полными дубликатами, — одна из наиболее частых причин получения неправильных результатов. Вы можете заподозрить, что у вас появилась проблема с дубликатами, когда вдруг количество клиентов или общий объем продаж, возвращенные запросом, в несколько раз превысили ожидаемое число. К счастью, есть несколько способов исправить это.

Еще одна распространенная проблема — отсутствующие данные, о которой мы поговорим далее.

# 2.5. Подготовка: очистка данных

Профилирование часто выявляет, какие исправления в данных могут сделать их более полезными для анализа. Некоторые из этих исправлений представляют собой CASE-преобразование, исправление null-значений и изменение типа данных.

## Очистка данных с помощью *CASE*

Операторы CASE можно использовать для выполнения различных задач по очистке, обогащению и обобщению данных. Иногда необработанные данные сами по себе точны, но для анализа было бы удобнее, если бы значения были унифицированы или сгруппированы по категориям. Структура оператора CASE была приведена ранее в *разд. 2.3,* посвященном биннингу.

Нестандартные значения возникают по разным причинам. Данные могли поступать из разных систем с немного различающимися справочниками, системный код мог быть изменен, варианты выбора могли быть представлены клиенту на разных языках, или клиент мог ввести произвольное значение сам, а не выбрать из списка.

Представьте, что у нас есть столбец, содержащий информацию о поле человека. Значения, указывающие на лицо женского пола, могут быть 'F', 'female' и 'femme'. Мы можем стандартизировать эти значения следующим образом:

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

Операторы CASE также можно использовать для категоризации или обогащения, чего нет в исходных данных. Например, многие организации используют показатель NPS (Net Promoter Score, индекс потребительской лояльности) для отслеживания настроений клиентов. В опросах NPS респондентов просят оценить по шкале от 0 до 10, насколько вероятно, что они порекомендуют компанию или продукт другу или коллеге. Оценки от 0 до 6 считаются недоброжелательными (detractor, критик), 7 и 8 — пассивными (passive, нейтрал), а 9 и 10 — способствующими продвижению (promoter, промоутер). Итоговый показатель рассчитывается путем вычитания процента критиков из процента промоутеров. Данные с результатами опроса обычно включают в себя необязательные текстовые комментарии и иногда дополняются информацией, известной о респонденте. Получив набор данных с опросом NPS, на первом шаге надо сгруппировать ответы по категориям «критик», «нейтрал» и «промоутер»:

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

```
      when likelihood <= 8 then 'Passive'
      else 'Promoter'
     end as response_type
FROM nps_responses
;
```

Обратите внимание, что типы данных проверяемого поля и возвращаемых данных могут различаться. В нашем случае мы сравниваем целые числа и возвращаем строку. Здесь также можно было перечислить нужные значения с помощью оператора IN. Оператор IN позволяет указать список значений вместо того, чтобы проверять равенство для каждого по отдельности. Это удобно, когда входные данные не являются непрерывными или когда значения в списке идут не по порядку:

```
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
```

Оператор CASE может проверять несколько столбцов и содержать сложную логику с AND/OR. Они также могут быть вложенными, хотя, как правило, этого можно избежать с помощью 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
```

### Альтернатива *CASE* при очистке данных

Очистка и обогащение данных с помощью оператора CASE работают хорошо, пока список возможных вариантов относительно короткий, вы можете найти всех их в данных, и этот список не будет меняться. Для более длинных списков и тех списков, которые часто меняются, лучше использовать таблицу-справочник. Таблица-справочник находится в базе данных и является либо статической, либо заполняется с помощью кода, который периодически проверяет наличие новых значений. В запросе необходимо будет выполнить JOIN с таблицей-справочником, чтобы получить очищенные данные. Таким образом, очищенные значения можно хранить вне вашего кода и использовать во множестве запросов, не беспокоясь о

сохранении согласованности между ними. Примером может быть таблица-справочник, в которой хранятся аббревиатуры штатов США и их полные названия. В своей работе я сперва начинаю с использования оператора CASE и создаю таблицу-справочник только тогда, когда список становится неуправляемым или когда становится понятно, что мне или моей команде придется многократно повторять эту очистку данных.
Конечно, стоит также выяснить, можно ли очистить данные заранее. Однажды я начала с оператора CASE примерно из 5 строк, который вскоре вырос до 10 строк, и в конце концов он стал занимать более 100 строк, после чего этот список стал неуправляемым и его было трудно поддерживать. Этот опыт оказался достаточно полезным, и я смогла убедить инженеров изменить код и добавить в набор данных более крупные категории.

Еще один полезный пример использования операторов CASE, — это создание флагов, указывающих на равенство определенному значению, без возврата фактического значения. Это может быть удобно при профилировании для оценки того, насколько широко распространен определенный атрибут. Еще одно применение флагов — подготовка набора данных для статистического анализа. В этом случае флаг также используется как фиктивное поле, принимающее значение 0 или 1, и указывает на наличие или отсутствие некоторого качественного показателя. Например, мы можем создать флаги is_female и is_promoter, используя операторы CASE, для полей gender и likelihood:

```
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 ...
;
```

Если вы работаете с набором данных, в котором есть несколько строк для каждой сущности, например с позициями заказа, вы можете «свернуть» данные с помощью агрегатной функции от оператора CASE, чтобы получить флаг со значениями 1 и 0 в качестве возвращаемого значения. Ранее мы уже говорили о том, что тип данных BOOLEAN часто используется для создания флагов (полей, указывающих на наличие или отсутствие какого-либо атрибута). Здесь значение 1 заменяет TRUE, а 0 — FALSE, чтобы можно было применить агрегирование max. Это работает следующим образом: для каждого клиента оператор CASE вернет 1 для каждой строки с типом фруктов 'apple'. Затем вычисляется max и возвращается наибольшее значение из этих строк для клиента. Если покупатель купил яблоко хотя бы один раз, флаг всегда будет равен 1, в противном случае — 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
;
```

Вы также можете добавить более сложные условия для флагов, например установить пороговое значение или количество чего-либо для маркировки значением 1. Например, для определения любителей яблок и апельсинов можно проверять количество купленных фруктов:

```
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
;
```

Операторы `CASE` предоставляют нам широкие возможности, и, как мы видели, их можно использовать для очистки и обогащения данных, установки флагов и добавления фиктивных полей в наборы данных. Далее мы рассмотрим некоторые специальные функции, связанные с операторами `CASE`, которые отдельно обрабатывают `null`-значения.

## Преобразование типов

Для каждого поля в базе данных определен тип данных, который мы рассмотрели в *разд. 2.1*. Когда данные добавляются в таблицу, значения, которые не соответствуют типу поля, отклоняются базой данных. Строковые значения нельзя вставить в целочисленные поля, а логические значения — в поля дата/время. В большинстве случаев мы можем использовать типы данных как они есть и применять строковые функции к строкам, функции даты/времени к датам и т. д. Однако иногда нам нужно переопределить тип данных поля и заставить его быть чем-то другим. Здесь и вступают в действие преобразования и приведение типов.

*Функции преобразования типов* позволяют менять тип данных какого-то значения в соответствующем формате с одного на другой. Есть несколько вариантов синтаксиса, которые в принципе эквивалентны. Один из способов — воспользоваться функцией `cast` в виде `cast(input as data_type)`, второй — использовать два двоеточия *in-*

*put::data_type*. Оба способа равнозначны, и с их помощью можно, например, преобразовать целое число 1234 в строку:

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

Преобразование целого числа в строку может пригодиться в операторе CASE при категоризации числовых значений с неограниченным верхним или нижним пределом. Например, следующий код, возвращающий целое число при значениях order_items, меньших или равных 3, и строку '4+' для больших значений, приведет к ошибке:

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

Преобразование целого числа к типу VARCHAR решает эту проблему:

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

Преобразования типов также могут пригодиться, когда значения, которые должны быть целыми числами, получаются из строки, а затем к ним надо применить агрегирование или математические функции. Представьте, что у нас есть данные о ценах, но значения содержат знак доллара ($), поэтому тип поля — VARCHAR. Мы можем удалить символ '$' из строки с помощью функции replace, которая будет рассматриваться подробнее в *разд. 5.5*:

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

replace
-------
19.99
```

Однако результат по-прежнему имеет тип VARCHAR, поэтому попытка применить агрегирование вернет ошибку. Чтобы это исправить, мы можем преобразовать результат выполнения функции в FLOAT двумя способами:

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

Дата и время могут быть представлены в огромном количестве разных форматов, и необходимо знать, как приводить их к нужному виду. Здесь я покажу несколько примеров преобразования типов, а в *разд. 3.1* будут более подробно рассмотрены возможные операции со значениями даты/времени. В качестве простого примера возьмем данные о транзакциях или событиях, поступающие в базу данных в виде TIMESTAMP (временная метка, т. е. дата и время), но мы хотим получить агрегированные значения, такие как количество транзакций за день. Простая группировка по полю дата/время вернет намного большее количество строк, чем необходимо. При-

ведение типа TIMESTAMP к типу DATE (только дата) уменьшит количество результатов и вернет то, что нам нужно:

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

Точно так же тип DATE может быть преобразован к типу TIMESTAMP, когда какая-то функция SQL требует в качестве аргумента значение TIMESTAMP. Иногда год, месяц и день хранятся в отдельных столбцах или получаются в виде отдельных значений после выделения из длинной строки. Далее их нужно собрать обратно в дату. Для этого воспользуемся оператором конкатенации || или функцией concat, а затем преобразуем результат к типу DATE:

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

Или, что то же самое:

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

Еще одним способом приведения строковых значений к дате является использование функции date. Например, мы можем собрать строковое значение, как указано выше, и преобразовать его в дату:

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

Функции to_*datatype* могут принимать на вход как значение, так и строку формата и, таким образом, дают вам бóльший контроль над преобразованием данных. В табл. 2.4 перечислены эти функции и их применение. Они особенно удобны при преобразовании в форматы даты/времени и обратно, поскольку позволяют указать порядок следования частей даты и времени.

**Таблица 2.4.** Функции to_datatype

| Функция | Описание |
|---|---|
| to_char | Преобразует другие типы данных в строку. |
| to_number | Преобразует другие типы в числовой тип. |
| to_date | Преобразует другие типы в дату с указанными частями даты. |
| to_timestamp | Преобразует другие типы в дату с указанными частями даты и времени. |

Иногда база данных автоматически преобразует тип данных. Это называется *неявное приведение типов*. Например, числовые значения INT и FLOAT можно одновременно использовать в математических функциях и агрегировании без явного преобразования типа данных. Значения CHAR и VARCHAR тоже, как правило, можно совмещать в выражениях. Некоторые базы данных будут неявно приводить значения BOOLEAN к 0 и 1, где 0 — это FALSE, а 1 — TRUE, но другие базы данных могут потребо-

вать, чтобы вы явно указали преобразование логических значений. Некоторые базы данных более требовательны в отношении одновременного использования значений DATE и TIMESTAMP в запросах и функциях. Просмотрите документацию или выполните несколько простых запросов, чтобы узнать, как база данных, с которой вы работаете, выполняет преобразование типов данных — явно и неявно. Обычно добиться того, чего вы хотите, очень легко, хотя изредка вам все-таки придется проявить творческий подход при использовании функций в ваших запросах.

## Работа с *null*-значениями: функции *coalesce*, *nullif*, *nvl*

Значение null было одним из самых странных понятий, к которому мне пришлось привыкать, когда я начала работать с данными. Это не то, с чем мы сталкиваемся в повседневной жизни, где мы привыкли иметь дело с конкретным количеством вещей. Значение null имеет особый смысл в базах данных и было введено Эдгаром Коддом, основателем теории реляционных баз данных, чтобы можно было обозначать отсутствие информации. Если кто-то спросит, сколько у меня парашютов, я отвечу «ноль». Но пока этот вопрос никто не задал, у меня null парашютов (неизвестно сколько).

Отсутствующие значения в полях базы данных могут означать, что для этих полей не собрано никаких данных или данные непригодны. Когда в таблицу добавляется новый столбец, он заполняется null-значениями для ранее созданных записей, если не было указано какое-то значение по умолчанию. Когда две таблицы соединяются с помощью OUTER JOIN, значения null проставляются во всех полях, для которых нет соответствующей записи во второй таблице.

Значения null могут быть проблемой для определенных агрегаций и группировок, разные типы баз данных обрабатывают их по-разному. Представьте, например, что у меня есть пять записей со значениями 5, 10, 15, 20 и null. Их сумма равна 50, а среднее значение может быть равно 10 или 12.5 в зависимости от того, учитывается ли значение null в знаменателе. Весь вопрос также может считаться некорректным, поскольку одно из значений равно null. Для большинства функций баз данных null на входе вернет null на выходе. Равенства и неравенства со значением null также вернут null. Ваши запросы могут возвращать неожиданные и странные результаты, если вы не будете следить за отсутствующими значениями.

Таблицы определяются так, что в полях могут быть или разрешены null-значения, или запрещены null-значения, или задано значение по умолчанию, которое будет применено, если поле оставлено пустым. На практике это означает, что вы не можете всегда полагаться на то, что при отсутствии реальных данных в поле будет null, потому что оно может заполняться значением по умолчанию, например нулем. Однажды у меня был долгий спор с разработчиком, когда выяснилось, что отсутствующие в исходной системе даты заполняются по умолчанию значением '1970-01-01' в нашей базе данных. Я настаивала на том, чтобы вместо этого даты оставались значения null, чтобы отразить тот факт, что они неизвестны или неприемлемы. Разработчик парировал, что я могу предварительно отфильтровывать эти даты или са-

ма менять их на null с помощью оператора CASE. В конце концов я его переубедила, сказав, что в один прекрасный день придет другой специалист, который тоже не будет знать этого нюанса с датами по умолчанию, выполнит свой запрос и получит загадочную группу клиентов, обращавшихся в компанию за несколько лет до ее основания.

Значения null часто неудобны или не подходят для анализа данных. Они также могут сбить с толку целевую аудиторию, для которой вы делаете ваш анализ. Деловые люди не обязательно понимают, как интерпретировать null-значения, или могут предположить, что они указывают на проблему с качеством данных.

### Пустые строки

Еще одна концепция, связанная с null-значениями, но немного отличающаяся от них, — это *пустая строка* (''), в которой нет значения, но поле технически не равно null. Одной из причин использования пустой строки является то, что про поле известно, что оно должно быть незаполненным, в отличие от null, которое означает, что значение отсутствует или неизвестно. Например, в базе данных может быть поле name_suffix для хранения такого дополнения к фамилии, как 'Jr' («младший»). У многих людей нет никакого дополнения к фамилии, поэтому для значения name_suffix подходит пустая строка. Пустую строку также можно использовать как значение по умолчанию или как способ обойти ограничение NOT NULL при вставки новой записи. В запросе пустая строка обозначается с помощью двух одинарных кавычек:

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

Профилирование частот значений должно показать, содержат ли ваши данные null-значения, пустые строки или и то, и другое.

Есть несколько способов заменить null альтернативными значениями: использовать оператор CASE или специальные функции coalesce и nullif. Ранее мы уже говорили о том, что оператор CASE проверяет условие и возвращает значение. С его помощью можно также проверять наличие null-значения и, если оно найдено, заменить его другим значением:

```
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
```

С помощью функции coalesce можно более компактно добиться того же. Она принимает два аргумента или более и возвращает первый из аргументов, который не равен null:

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

В некоторых базах данных определена функция `nvl`, похожая на `coalesce`, но принимающая только два аргумента.

Функция `nullif` сравнивает два аргумента и, если они не равны, возвращает первый из них, или, если они равны, возвращается `null`. Запуск этого кода:

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

вернет 6, тогда как в следующем случае результат будет `null`:

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

Функция `nullif(column_a,column_b)` эквивалентна такому оператору `CASE`:

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

Эта функция может быть полезна для обратного преобразования значений в `null`, когда вы знаете, какое значение по умолчанию было добавлено в базу данных. Так, в моем примере с датой по умолчанию я могла бы изменить значения обратно на `null`, используя:

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

Значения `null` могут вызывать сложности при фильтрации данных с помощью `WHERE`. Выбрать `null`-значения довольно просто:

```
WHERE my_field is null
```

Представьте, что поле `my_field` содержит `null`-значения, а также несколько названий фруктов. И мне нужно вернуть все строки, которые не являются яблоками. Кажется, это должно сработать:

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

Однако некоторые базы данных исключат из выборки не только строки со значением `'apple'`, но и все строки со значением `null` в `my_field`. Чтобы исправить это, в SQL-запрос, кроме исключения `'apple'`, нужно явно включить проверку на `null`-значения с помощью условия `OR`:

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

Не забывайте про значения `null` при работе с данными. Независимо от того, почему они возникают, нам часто приходится учитывать их при профилировании и при очистке данных. К счастью, в SQL есть несколько способов их обработки, а также несколько полезных функций, которые позволяют вместо `null` подставлять альтернативные значения. Далее мы рассмотрим проблему отсутствующих данных, которая может привести к появлению `null`-значений, но может иметь и более широкие последствия, и поэтому заслуживает отдельного раздела.

## Отсутствующие данные

Данные могут отсутствовать по разным причинам, каждая из которых по-своему влияет на то, как вы будете это обрабатывать. Поле могло быть не обязательным для системы или процесса, собирающего эти данные, как в случае с полем «откуда вы узнали о нас?», которое можно не заполнять при оформлении заказа электронной торговли. Требование заполнения этого поля может вызвать недовольство клиентов и уменьшить количество заказов. В другой ситуации, когда данные являются обязательными, они могут быть не собраны из-за ошибки в коде или из-за человеческой ошибки, например в медицинском опроснике интервьюер пропустил вторую страницу с вопросами. Кроме того, изменение способа сбора данных может привести к тому, что в записях до и после внедрения этого изменения будут отсутствовать некоторые значения. Трекер, отслеживающий взаимодействие с мобильным приложением, может добавить дополнительное поле, в котором будет записываться действие пользователя, например нажатие или прокрутка, или трекер может удалить какое-то поле из-за изменения своей функциональности. Данные могут потеряться, если таблица ссылается на значение в другой таблице, а та строка или вся таблица были удалены или еще не успели загрузиться в хранилище данных. И, наконец, данные могут быть доступны, но не на таком уровне детализации, который необходим вам для анализа. Примером этого может быть бизнес с платной годовой подпиской, а мы хотим проанализировать прибыль по месяцам.

Кроме профилирования данных с помощью гистограмм и частотного анализа, мы можем обнаружить отсутствие данных, просто сравнивая записи в двух таблицах. Например, можно ожидать, что для каждого клиента из таблицы `transactions` есть запись в таблице `customers`. Чтобы проверить это, напишите запрос с использованием `LEFT JOIN` и добавьте условие `WHERE`, чтобы найти клиентов, которых нет во второй таблице:

```
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
;
```

Отсутствие данных может быть важным само по себе, поэтому не думайте, что это всегда нужно исправлять или заполнять. Отсутствующие данные могут помочь выявить недостатки в базовой структуре системы или в процессе сбора данных.

Все записи с отсутствующими полями можно отфильтровать, но чаще всего мы хотим сохранить их и внести некоторые коррективы на основе того, что мы знаем об ожидаемых или типичных значениях. У нас есть несколько способов, называемых методами *импутации* (восстановления), для заполнения недостающих данных. К ним относится заполнение средним или медианным значением набора данных или заполнение предыдущим значением. Важно вести документацию об отсутствующих данных и о том, как они были исправлены, поскольку это может повлиять на последующую интерпретацию и использование данных. Восстановленные зна-

чения могут привести к специфичным проблемам, например, когда эти данные используются в машинном обучении.

Распространенным способом восстановления является заполнение отсутствующих данных постоянным значением. Это может быть удобно, когда некоторые значения известны заранее, даже если они не сохранены в базе данных. Представьте, например, что в приложении возникла ошибка, из-за которой цена товара с названием `'abc'` не была заполнена, но мы знаем, что цена составляет 20 долларов. Для обработки этой ситуации может быть использован оператор `CASE`:

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

Другой способ — заполнить значением, полученным из других полей с помощью математических функций или оператора `CASE`. Например, у нас есть поле `net_sales` (чистая продажа) каждой транзакции. Из-за ошибки в некоторых строках это поле не заполнено, но заполнены поля `gross_sales` (валовая продажа) и `discount` (скидка). Мы можем рассчитать `net_sales`, вычтя `discount` из `gross_sales`:

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

Отсутствующие данные также можно заполнить значениями из других строк набора данных. Перенос значения из предыдущей строки называется *прямым заполнением* (fill forward), а использование значения из последующей строки называется *обратным заполнением* (fill backward). Это можно сделать с помощью оконных функций `lag` и `lead` соответственно. Например, в нашей таблице транзакций есть поле `product_price`, в котором хранится цена купленного товара без скидки. Иногда это поле не заполняется, но мы можем предположить, что цена такая же, как и у последнего клиента, купившего этот же товар. Мы можем взять предыдущее значение, используя функцию `lag` с группировкой `PARTITION BY product`, чтобы гарантировать, что цена выбирается для этого же товара, и с сортировкой `ORDER BY` по дате, чтобы найти самую последнюю транзакцию из всех предыдущих:

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

Функцию `lead` можно использовать для заполнения `product_price` из последующей транзакции. Кроме того, мы могли бы найти среднюю цену товара и использовать ее для заполнения отсутствующих значений. Заполнение предыдущими, последующими или средними значениями должно опираться на предположения о типовых значениях и о том, что эти записи целесообразно включать в анализ. Всегда полезно проверить результаты, убедиться, что они правдоподобны, и оставить примечание о том, что вы интерполировали отсутствующие данные.

Для данных, которые представлены с недостаточной степенью детализации, нам часто приходится создавать дополнительные строки в наборе данных. Представьте, например, что у нас есть таблица `customer_subscriptions` (подписки) с полями `subscription_date` и `annual_amount` (годовая сумма). Мы можем разбить эту годовую сумму по месяцам, просто разделив на 12, эффективно преобразовав ARR (annual

recurring revenue, годовой регулярный доход) в MRR (monthly recurring revenue, ежемесячный регулярный доход):

```
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
;
```

То же самое можно сделать, если период подписки составляет два, три года или пять лет, хотя запрос станет более трудоемким. И этот способ не поможет, если нам нужны фактические даты месяцев. Теоретически мы могли бы написать такой запрос:

```
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
;
```

Однако если данные содержат оплаты пользователей в разное время, использование фиксированных названий месяцев будет неправильным. Можно применить операторы `CASE` вместе с фиксированными названиями месяцев, но, опять же, написание такого запроса будет утомительным и, вероятно, с ошибками, т. к. вы добавите более запутанную логику. Вместо этого для создания новых строк лучше использовать более элегантное решение — `JOIN` к размерной таблице дат.

*Размерная таблица дат* (date dimension) — это статическая таблица, которая имеет по строке на каждый день с необязательными дополнительными полями, такими как день недели, полное название месяца, квартал и финансовый год. Даты в этой таблице должны уходить достаточно далеко в прошлое и достаточно далеко в будущее, чтобы охватить все возможные временные периоды. Поскольку в году всего 365 или 366 дней, таблица, охватывающая даже 100 лет, не займет много места. На рис. 2.3 показан фрагмент подобной таблицы. Пример SQL-кода для создания размерной таблицы дат находится в репозитории GitHub[5] для этой книги.

[5] **https://github.com/cathytanimura/sql_book/blob/master/Chapter 2: Preparing Data for Analysis/create_date_dimension.sql**

| 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 |

**Рис. 2.3.** Фрагмент размерной таблицы дат с дополнительными полями

Если вы работаете с базой данных Postgres, вы можете использовать функцию `generate_series` для генерации последовательности дат, чтобы заполнить размерную таблицу дат или даже применять вместо нее в запросах, когда создание такой таблицы невозможно. Функция имеет вид:

```
generate_series(start, stop, step_interval)
```

В этой функции *start* — это первая дата в вашей последовательности дат, *stop* — последняя дата, а *step_interval* — это временной интервал между значениями. Интервал *step_interval* может быть любым, но для размерной таблицы дат нужно указать один день:

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

Функция `generate_series` требует, чтобы хотя бы один из аргументов имел тип `TIMESTAMP`, поэтому первый аргумент `'2000-01-01'` приводится к `TIMESTAMP`. Затем мы можем написать запрос, который возвращает по строке на каждый день независимо от того, сделал ли клиент заказ в этот день. Это удобно, когда нам нужно проверить, что клиенты учитываются правильно, или когда мы хотим отдельно проанализировать те дни, в которых не было никаких заказов:

```
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
;
```

Возвращаясь к нашему примеру с годовой подпиской, для создания строк для каждого месяца мы можем использовать JOIN с размерной таблицей дат, указав в условии соединения, что дата должна находиться в промежутке между subscription_date и 11-ю месяцами позже (всего 12 месяцев). Кроме того, условие a.date = a.first_day_of_month ограничивает вывод только одной строки для каждого месяца, в противном случае вернулось бы по строке на каждый день:

```
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'
WHERE a.date = a.first_day_of_month
;
```

Данные могут отсутствовать по многим причинам, и эти причины важно знать, чтобы принять решение о том, как это обрабатывать. Есть несколько способов поиска и заполнения отсутствующих данных. Можно использовать операторы CASE для установки значений по умолчанию или высчитывать пропущенные значения по значениям других полей в той же строке, или заполнять пробелы на основе предыдущих или последующих значений в этом же столбце.

Очистка данных является важным этапом процесса подготовки данных к анализу. Это может быть необходимо по разным причинам. Иногда с помощью очистки данных можно исправить плохое качество данных, например, когда в необработанных данных есть противоречивые или отсутствующие значения, а в других случаях очистка данных нужна, чтобы сделать дальнейший анализ более простым или более достоверным. Гибкость SQL позволяет выполнять эти задачи различными способами.

После очистки данных следующим шагом, как правило, идет структурирование набора данных.

## 2.6. Подготовка: структурирование данных

*Структурирование данных* относится к изменению способа организации данных в строки и столбцы. Каждая таблица в базе данных имеет определенную форму. Результаты каждого запроса тоже имеют свою форму. Структурирование данных может показаться довольно абстрактным понятием, но если вы работаете с большим

количеством данных, вы почувствуете его важность. Этому навыку можно научиться, его можно практиковать и развивать.

Одним из наиболее важных вопросов при структурировании данных является определение степени детализации данных, которая вам нужна. Точно так же, как камни могут быть разных размеров, от гигантских валунов до песчинок и даже до микроскопической пыли, данные могут иметь разный уровень детализации. Например, если население страны — это валун, то население города — это маленький булыжник, а население какого-то дома — песчинка. Данные с более мелким уровнем детализации могут включать в себя дни рождения или переезды с места на место.

*Выравнивание данных* — еще одно важное понятие при структурировании данных. Оно выражается в уменьшении количества строк, представляющих каждую сущность, в том числе до одной строки. Объединение нескольких таблиц вместе для создания единого набора выходных данных — один из способов выравнивания данных. Другой способ — это агрегация.

В этом разделе мы сначала поговорим о выборе структуры данных. Затем мы разберем некоторые способы сворачивания и разворачивания данных. Примеры структурирования данных для конкретных анализов мы рассмотрим в других главах. А в *гл. 8* более подробно обсуждается организация сложных SQL-запросов при создании наборов данных для последующего анализа.

## Зависимость от конечной цели: для BI, визуализации, статистики или машинного обучения

Решение о том, как структурировать данные с помощью SQL, во многом зависит от того, что вы планируете делать с ними впоследствии. Общая идея заключается в том, чтобы набор данных содержал как можно меньше строк, но при этом сохранял бы необходимую степень детализации. Это позволит эффективнее использовать вычислительную мощность базы данных, сократит время, необходимое для копирования данных из базы куда-то еще, и уменьшит объем работы, который необходимо будет выполнить в сторонних приложениях. Например, ваш результирующий набор данных может использоваться в инструменте BI для создания отчетов и информационных панелей, в электронной таблице для изучения бизнес-пользователями, в инструменте для статистической обработки, таком как R, или в модели машинного обучения на Python, или вы можете использовать данные непосредственно для построения графиков, генерируемых с помощью соответствующих приложений.

При подготовке данных для инструмента BI важно понимать, как они будут использоваться. Возможно, наборы данных должны быть очень подробными, чтобы конечные пользователи могли проводить дополнительные исследования и делать срезы. Или же наборы данных должны быть небольшими и агрегированными, а также включать в себя специфичные вычисления, чтобы обеспечить быструю загрузку и время отклика для информационных панелей. Важно понимать, как работает этот инструмент, будет ли он лучше работать с небольшими наборами данных или он был спроектирован для выполнения собственных агрегаций для больших

наборов данных. Не существует какого-то универсального ответа на все эти вопросы. Чем больше вы узнаете о том, как в дальнейшем будут использоваться ваши данные, тем лучше вы будете понимать, как нужно структурировать ваши данных.

Небольшие агрегированные и очень специфичные наборы данных, как правило, лучше всего подходят для визуализации независимо от того, созданы ли они в каком-то коммерческом приложении или с использованием языка программирования, такого как R, Python или JavaScript. Подумайте об уровне агрегирования и срезах — элементах, по которым конечным пользователям нужно будет выполнять фильтрацию. Иногда в наборах данных нужны отдельные строки для каждого среза, а также дополнительный срез «Итого». Возможно, вам потребуется объединить с помощью `UNION` два запроса — один будет с нужным уровнем детализации, а другой — для среза «Итого».

При создании выходных данных для инструментов статистики или для моделей машинного обучения важно знать основную изучаемую сущность, требуемый уровень агрегирования и необходимые атрибуты или характеристики. Например, модели могут потребоваться записи по каждому клиенту с несколькими атрибутами или записи по каждой транзакции с ее атрибутами и атрибутами клиента. Как правило, результирующие данные для моделирования должны соответствовать понятию «аккуратные данные» (tidy data), предложенному Хэдли Уикхэмом[6]. Аккуратные данные обладают следующими свойствами:

1. Каждая переменная образует отдельный столбец.
2. Каждое наблюдение записывается в отдельную строку.
3. Каждое значение представляет собой отдельную ячейку.

Далее мы рассмотрим, как данные, которые хранятся в вашей базе, с помощью SQL свернуть или развернуть в структуру, необходимую для анализа.

## Сворачивание с помощью оператора *CASE*

*Сводная таблица* (pivot table) — это способ суммирования данных с одновременным созданием строк по значениям одного атрибута, а столбцов — по значениям другого атрибута. На пересечении каждой строки и каждого столбца вычисляется сводная статистика, например, с помощью агрегатных функций `sum`, `count` или `avg`. Сводные таблицы часто являются хорошим способом представления данных для бизнес-аудитории, поскольку они позволяют свернуть данные в более компактную и понятную форму. Такие таблицы широко известны благодаря Microsoft Excel, в котором реализован быстрый и удобный интерфейс для их создания.

Сводные таблицы или сводные выходные данные можно создавать в SQL с помощью оператора `CASE` вместе с одной или несколькими функциями агрегирования. Мы уже несколько раз встречались с оператором `CASE`, и изменение структуры дан-

---

[6] *Hadley Wickham*, Tidy Data. Journal of Statistical Software 59, no. 10 (2014): 1–23, **https://doi.org/10.18637/jss.v059.i10**.

ных — это еще один удачный способ его применения. Представьте, что у нас есть таблица orders с записями о каждой покупке, сделанной клиентами. Для выравнивания данных просуммируем по order_amount и сделаем GROUP BY по customer_id:

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

Пусть в таблице orders есть также поле product, которое содержит тип приобретенного товара, и дата заказа order_date. В сводной таблице мы создадим дополнительные столбцы для каждого значения атрибута — типа товара, а в строках будут даты заказов. Для этого выполним группировку GROUP BY по order_date и просуммируем sum результаты выполнения оператора CASE, который возвращает order_amount всякий раз, когда строка соответствует определенному типу продукта:

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

Обратите внимание, что в операторах `CASE` указано значение по умолчанию `else 0`, чтобы при агрегировании избежать `null`-значений. Но если вы используете `count` или `count distinct`, вам не нужно добавлять предложение `ELSE`, т. к. это приведет к раздуванию результатов. Это связано с тем, что база данных не будет подсчитывать значения `null`, но учтет каждое непустое значение, даже ноль.

Создавать сводные таблицы с помощью операторов `CASE` очень удобно, и это открывает новые возможности для хранения разреженных данных. Длинные и узкие таблицы лучше подходят для разреженных данных, чем широкие таблицы, поскольку добавление столбцов в таблицу может быть дорогостоящей операцией. Например, вместо того, чтобы хранить различные атрибуты клиентов в отдельных столбцах, таблица может содержать несколько строк для каждого клиента, по одному атрибуту в каждой строке, с полями `attribute_name` и `attribute_value` для названия атрибута и его значения. При необходимости такие данные можно свернуть, чтобы получить одну запись о клиенте со всеми атрибутами. Этот дизайн эффективен, когда у вас есть очень много разреженных атрибутов (когда только несколько клиентов имеют множество заполненных атрибутов).

Построение сводной таблицы с помощью агрегирования и оператора `CASE` работает хорошо, когда вы сворачиваете данные по конечному числу элементов. Если вы писали на других языках программирования, то это похоже на то, когда вместо цикла используют длинный построчный код. Такой код дает вам больше контроля, например, если вы хотите вычислять разные показатели в разных столбцах, но он может оказаться очень трудоемким. Этот способ построения сводной таблицы не будет работать, если постоянно добавляются новые значения атрибута или его значения быстро меняются, поскольку код SQL нужно будет постоянно обновлять. В таких случаях более эффективным может оказаться перенос вычислений на другой этап вашего анализа, например в инструмент BI или на язык для статистической обработки данных.

## Разворачивание с помощью оператора *UNION*

Иногда у нас возникает обратная задача, и нам нужно перенести компактные данные, хранящиеся в столбцах, в строки. Эта операция называется *разворачиванием* (unpivot). Наборы данных, для которых может потребоваться разворот, — это наборы данных в виде сводной таблицы. Рассмотрим пример с населением стран Северной Америки с 10-летним интервалом[7], начиная с 1980 г., показанным на рис. 2.4.

Чтобы превратить этот набор данных в таблицу с отдельными строками для каждой страны в определенный год, мы можем использовать оператор `UNION`. Этот оператор позволяет объединить наборы строк из нескольких запросов в один набор результатов. У него есть две формы: `UNION` и `UNION ALL`. Количество столбцов в каждом запросе должно совпадать. Типы данных в столбцах должны совпадать или быть со-

[7] US Census Bureau, "International Data Base (IDB)," last updated December 2020, **https://www.census.gov/data-tools/demo/idb**.

вместимыми (целые числа и числа с плавающей запятой можно сложить в один столбец, а целые числа и строки — нельзя). Имена столбцов в результирующем наборе берутся из первого запроса. В остальных запросах необязательно использовать псевдонимы полей, но они могут упростить чтение кода:

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

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

**Рис. 2.4.** Население стран по годам (в тысячах)

В этом примере, чтобы заполнить столбец `year`, мы явно прописываем в коде константу для каждого года, которому соответствует количество населения. Такие же-

стко прописанные константы могут быть любого типа, это зависит от ситуации. Вам может также потребоваться явное преобразование типов для указанных значений, например при вводе даты:

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

В чем разница между UNION и UNION ALL? Оба оператора могут использоваться для такого объединения данных, но они немного отличаются. Оператор UNION удаляет дублирующиеся строки из набора результатов, тогда как UNION ALL сохраняет все записи, независимо от того, повторяются они или нет. При этом UNION ALL выполняется быстрее, т. к. базе данных не нужно выполнять проход по данным для поиска дубликатов. Это также гарантирует, что каждая запись окажется в результирующем наборе. Я чаще использую UNION ALL и только в редких случаях — UNION, когда у меня есть причина подозревать дублирование данных.

Объединение данных с помощью UNION также может пригодиться при сборе данных из разных источников. Представьте, например, что у нас есть таблица population с ежегодными данными о населении стран и таблица gdp с годовым валовым внутренним продуктом (ВВП, gross domestic product, GDP). Один из вариантов объединения — выполнить JOIN для этих таблиц и получить результирующий набор с двумя столбцами: один для населения и другой для ВВП:

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

Другой вариант объединения — UNION ALL, чтобы в итоге мы получили «развернутый» набор данных:

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

Какой из вариантов использовать, во многом зависит от того, в каком виде вам нужны данные для вашего анализа. Второй вариант объединения может быть полезен, когда у вас есть несколько разных метрик в разных таблицах, и ни одна из них не содержит полного списка сущностей (в данном случае стран). Это альтернативный подход, который можно использовать вместо FULL OUTER JOIN.

## Операторы *PIVOT* и *UNPIVOT*

Из-за того, что сворачивание и разворачивание таблиц очень широко применяется на практике, в некоторых типах баз данных реализованы специальные операторы,

позволяющие выполнить это с помощью намного меньшего количества кода. В Microsoft SQL Server и Snowflake есть оператор PIVOT, который используется как дополнительное выражение после предложения FROM:

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

Здесь *aggregation* — это любая функция агрегирования, например sum или avg; *value_column* — это поле для агрегирования, и для каждого перечисленного значения *label_column* будет создан столбец. Мы могли бы переписать пример сворачивания таблицы orders, в котором использовали оператор CASE, следующим образом:

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

Хотя этот синтаксис более компактен, чем с оператором CASE, нужные столбцы все равно приходится перечислять в коде. В результате PIVOT не решает проблему с новыми или часто меняющимися значениями поля, которые преобразуются в столбцы. В Postgres в модуле tablefunc есть функция crosstab, выполняющая то же самое.

В Microsoft SQL Server и Snowflake есть также оператор UNPIVOT, который работает в обратную сторону и преобразует столбцы в строки:

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

Например, таблицу country_populations из примера ранее можно развернуть следующим образом:

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

Этот синтаксис тоже более компактен, чем вариант с UNION или UNION ALL, который мы видели ранее, но список столбцов все равно должен быть перечислен в запросе.

В Postgres есть функция unnest, которую можно использовать для разворачивания таблицы и которая работает с типом данных массив. Массив — это коллекция эле-

ментов, которые в Postgres перечисляются в квадратных скобках. Функцию можно использовать в операторе SELECT, и она выглядит так:

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

Возвращаясь к нашему примеру с населением стран, запрос с функцией unnest вернет тот же результат, что и запрос с повторяющимся оператором UNION ALL:

```
SELECT
country
,unnest(array['1980', '1990', '2000', '2010']) as year
,unnest(array[year_1980,year_1990,year_2000,year_2010]) as population
FROM country_populations
;

country  year  population
-------  ----  --------------
Canada   1980  24593
Canada   1990  27791
Canada   2000  31100
...      ...   ...
```

Наборы данных могут поступать к вам в различном формате и виде, и они не всегда соответствуют той структуре, которая необходима для нашей задачи. Сворачивание или разворачивание данных можно выполнить несколькими способами: либо с помощью оператора CASE, либо с помощью оператора UNION, либо с помощью особых функций или операторов, специфичных для каждой базы данных. Умение манипулировать вашими данными, чтобы придать им нужную форму, даст вам больше гибкости при выполнении анализа и для представления результатов.

## 2.7. Заключение

Может показаться, что подготовка данных для анализа совершенно неважна, раз она выполняется до того, как мы приступаем к настоящему анализу. Но этот этап является базовым для понимания данных, и я всегда уверена, что время будет потрачено не зря. Знание различных типов данных, которые могут вам встретиться, имеет существенное значение, и я рекомендую вам уделить время на то, чтобы познакомиться с каждой таблицей, с которой вы будете работать. Профилирование помогает нам понять, что находится в наборе данных, и проверить их качество. Я часто возвращаюсь к профилированию данных в своих аналитических проектах, чтобы узнать о них больше и проверить результаты моих запросов по мере усложнения задач. Качество данных, вероятно, всегда будет иметь критическое значение, поэтому мы рассмотрели некоторые способы очистки и улучшения наборов данных. Наконец, очень важно знать, как менять структуру данных для создания правильного результирующего набора. Мы увидим, что эти темы будут постоянно всплывать при проведении различных видов анализов на протяжении всей книги. В следующей главе мы начнем наше знакомство с методами анализа временных рядов.

# ГЛАВА 3
# Анализ временных рядов

Теперь, когда вы познакомились с SQL и базами данных, а также с основными этапами подготовки данных для анализа, пришло время перейти к конкретному типу анализа, который можно выполнить с помощью SQL. В мире существует бесконечное количество наборов данных и, соответственно, бесконечное количество способов их анализировать. В этой и последующих главах я разбила типы анализов по темам, что, я надеюсь, поможет вам быстрее развить свои навыки в анализе и в SQL. Главы лучше изучать последовательно, одну за другой, т. к. многие из методов, которые будут обсуждаться, опираются на методы, приведенные в предыдущих главах. Временные ряды настолько широко распространены и важны, что я начну именно с них.

Анализ временных рядов — один из наиболее распространенных типов анализа, выполняемых с помощью SQL. *Временной ряд* — это последовательность измерений или точек данных, записанных во временном порядке, часто через равные промежутки времени. В повседневной жизни есть много примеров таких временных рядов, например максимальная дневная температура, фондовый индекс S&P 500 на закрытии или количество ежедневных шагов, записанных вашим фитнес-трекером. Анализ временных рядов используется в самых разных отраслях и дисциплинах, от статистики и инженерии до прогнозов погоды и бизнес-планирования. Анализ временных рядов — это способ понять и количественно оценить, как все меняется с течением времени.

Основной целью такого анализа является прогнозирование. Поскольку время движется только вперед, будущие значения могут быть получены как функция от прошлых значений, но не наоборот. Однако важно отметить, что прошлое не может точно предсказать будущее. Любое колебание рыночных условий, популярных тенденций, внедрение новых продуктов и другие крупные изменения затрудняют прогнозирование. Тем не менее, просмотр исторических данных помогает оценить ситуацию, а разработка диапазона вероятных результатов полезна для планирования. Пока я пишу эту книгу, мир находится в эпицентре глобальной пандемии COVID-19, подобной которой не было уже 100 лет, — это дольше, чем история всех организаций, кроме самых долгоживущих. То есть многие организации, работающие сегодня, не проходили через такие испытания ранее, но они существовали во время других экономических кризисов, таких как крах доткомов и теракты 11 сентября в 2001 г., а также глобальный финансовый кризис 2007–2008 гг. При

тщательном анализе и понимании контекста мы можем извлечь полезную информацию из данных за эти периоды.

В этой главе мы сначала рассмотрим полезные инструкции SQL для анализа временных рядов: синтаксис и функции для работы с датами и временными метками. Затем я расскажу, где найти набор данных о розничных продажах, который мы будем использовать в качестве примеров в остальных разделах этой главы. После этого рассмотрим методы анализа тенденций, а затем я расскажу о расчете скользящих временных окон. Далее следуют расчеты временных периодов для анализа данных с сезонной компонентой. И, наконец, мы рассмотрим некоторые дополнительные приемы, полезные при анализе временных рядов.

## 3.1. Работа с *Date*, *Datetime* и *Time*

Дата и время могут быть представлены в самых разных форматах в зависимости от источника данных. Очень часто нам нужно преобразовать необработанные данные в привычный формат для вывода или выполнить вычисления, чтобы получить новые даты или какие-то части дат. Например, набор данных может содержать временные метки транзакций, а целью анализа является определение тенденций ежемесячных продаж. В других случаях нам может понадобиться узнать, сколько дней или месяцев прошло после определенного события. К счастью, в SQL есть много функций и способов форматирования, которые позволяют преобразовать почти любые необработанные входные данные в почти любые выходные данные, годные для анализа.

В этом разделе я сначала покажу вам, как преобразовывать часовые пояса, а затем подробно расскажу о форматировании дат и временных меток. Далее мы рассмотрим арифметические операции с датами и временем, в том числе те, которые оперируют интервалами. Интервал — это отдельный тип данных, который содержит промежуток времени, например количество месяцев, дней или часов. Хотя таблицы в базе данных могут хранить данные с типом интервал, на практике я редко видела, чтобы он применялся, поэтому я буду говорить об использовании интервалов только в функциях даты и времени. В конце раздела я расскажу о некоторых особенностях объединения или иного комбинирования данных из разных источников.

### Преобразование часовых поясов

Знание часового пояса, используемого в наборе данных, может предотвратить недоразумения и ошибки в процессе анализа. Часовые пояса делят мир на области протяженностью с севера на юг, в которых установлено одно и то же время. Часовые пояса позволяют в разных частях света иметь одинаковое местное время суток — так, например, солнце находится над головой в 12 часов дня, где бы вы ни оказались. Пояса имеют неправильные границы не только по географическим, но и по политическим причинам. В большинстве случаев разница между соседними поясами составляет один час, но некоторые смещены всего на 30 или 45 минут, по-

этому в мире насчитывается более 30 часовых поясов. Во многих странах, удаленных от экватора, введен переход на летнее время, но есть исключения, например, в США и Австралии, где некоторые штаты переходят на летнее время, а некоторые — нет. Каждый часовой пояс имеет стандартную аббревиатуру, например PST (Pacific Standard Time) для стандартного тихоокеанского времени и PDT (Pacific Daylight Time) для тихоокеанского летнего времени.

Многие базы данных настроены на работу с UTC (Coordinated Universal Time) — *всемирным координированным временем* — это глобальный стандарт, используемый для регулирования часов и записи событий. Оно заменило устаревшее GMT (Greenwich Mean Time) — *среднее время по Гринвичу*, которое вы все еще можете встретить в данных, если они поступают из старой базы. В UTC нет перехода на летнее время, поэтому оно остается неизменным в течение всего года. Это очень удобно для анализа. Я помню, как однажды менеджер по продукту запаниковал и попросил меня выяснить, почему продажи в одно воскресенье так сильно упали по сравнению с предыдущим воскресеньем. Я потратила много времени на написание запросов и поиск возможных причин, пока в конце концов не выяснила, что наши данные были записаны по тихоокеанскому времени (Pacific Time, PT). Переход на летнее время был рано утром в воскресенье, часы базы данных перевели на 1 час вперед, и в сутках оказалось всего 23 часа вместо 24, и поэтому продажи за день упали. Через полгода у нас был обратный переход с 25-часовым рабочим днем, когда продажи оказались необычно высоки.

Часто временные метки в базе данных записаны без часового пояса, и тогда вам нужно проконсультироваться с источником данных или разработчиками, чтобы выяснить, как ваши данные были сохранены. Формат UTC стал самым распространенным в наборах данных, с которыми я работаю, но, конечно, бывают и исключения.

Одним из недостатков UTC, да и любой машинной записи времени, является то, что мы теряем информацию о местном времени человека, выполняющего действия, которые записываются в базу данных. Например, мне хочется узнать, когда люди используют мое мобильное приложение чаще всего: в рабочее время или вечерами и по выходным. Если все мои клиенты находятся в одном часовом поясе, это несложно выяснить. Но если клиенты находятся в нескольких часовых поясах или в разных странах, то необходимо выполнять преобразование каждого записанного времени в местный часовой пояс.

Все местные часовые пояса имеют смещение относительно UTC. Например, смещение для PDT равно UTC–7 часов, а смещение для PST составляет UTC–8 часов. Временные метки в базах данных хранятся в формате `YYYY-MM-DD hh:mi:ss` (год-месяц-день часы:минуты:секунды). Временные метки с часовым поясом содержат дополнительную информацию о смещении UTC, выраженную в виде положительного или отрицательного числа. Преобразование из одного часового пояса в другой может быть выполнено с помощью выражения `"at time zone"`, за которым следует

аббревиатура целевого часового пояса. Например, мы можем преобразовать метку времени в формате UTC (со смещением –0) в PST:

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

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

Имя целевого часового пояса может быть константой или полем таблицы, что позволяет выполнять динамическое преобразование для набора данных. В некоторых базах есть функции `convert_timezone` или `convert_tz`, которые делают то же самое. При этом один из аргументов — часовой пояс результата, а второй —временная метка с часовым поясом, которую надо преобразовать:

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

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

Проверьте в документации к вашей базе данных имена часовых поясов и порядок следования аргументов в этих функциях. Многие базы данных содержат список часовых поясов и их аббревиатуры в системной таблице. В табл. 3.1 приведены имена системных таблиц для некоторых типов баз данных. Обращаться к таким таблицам можно с помощью обычного запроса `SELECT * FROM table_name`. В Википедии также есть список аббревиатур стандартных часовых поясов и их смещения относительно UTC[1].

**Таблица 3.1.** Системные таблицы с часовыми поясами в разных базах данных

| Тип базы данных | Имя системной таблицы |
|---|---|
| Postgres | `pg_timezone_names` |
| MySQL | `mysql.time_zone_names` |
| SQL Server | `sys.time_zone_info` |
| Redshift | `pg_timezone_names` |

Часовые пояса являются неотъемлемой частью работы с временными метками. С помощью функций преобразования часовых поясов можно перевести время с часовым поясом, в котором были записаны данные, в любой другой часовой пояс мира. Далее я покажу вам различные способы обработки дат и временных меток в SQL.

[1] **https://en.wikipedia.org/wiki/List_of_time_zone_abbreviations**

## Форматирование дат и временных меток

Даты и временные метки являются ключевыми значениями при анализе временных рядов. Из-за большого разнообразия вариантов записи даты и времени в исходных данных вам обязательно когда-нибудь потребуется преобразовать формат даты. В этом разделе я расскажу о нескольких наиболее частых преобразованиях и о том, как их выполнить с помощью SQL: как изменить тип данных, извлечь часть даты или времени, собрать дату или время из частей. Я начну с некоторых полезных функций, которые возвращают текущую дату и/или время.

Получение текущей даты или времени — обычная задача анализа, например, чтобы включить их в результаты выборки или использовать в арифметических вычислениях с датами, описанных в следующем разделе. Текущие дата и время называются *системным временем*, и хотя их легко получить с помощью SQL, существуют некоторые синтаксические различия в разных базах данных.

Для получения текущей даты в некоторых базах есть функция `current_date` (используется без круглых скобок):

```
SELECT current_date;
```

Есть много функций для получения текущих даты и времени. Изучите документацию к вашей базе данных или просто проверьте в SQL-запросе, какая функция вернет значение. Некоторые функции нужно указывать со скобками, но они не принимают никаких аргументов:

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

Кроме того, есть функции, которые возвращают только текущее время. Опять же, обратитесь к документации или сами проверьте, какие функции можно использовать в вашей базе данных:

```
current_time
localtime
timeofday()
```

В SQL есть ряд функций для изменения формата даты и времени. Чтобы отсечь лишнюю детализацию временной метки, используйте функцию `date_trunc`. Первый аргумент — это строковое значение, обозначающее единицу времени, до которой усекается временная метка из второго аргумента. Результатом является временная метка. Например:

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


date_trunc
-------------------
2020-10-01 00:00:00
```

Стандартные значения, которые можно использовать в первом аргументе, перечислены в табл. 3.2. Они покрывают диапазон от микросекунд до тысячелетий, обеспечивая достаточную гибкость. Базы данных, которые не поддерживают `date_trunc`, например MySQL, имеют альтернативную функцию `date_format`, которую можно использовать схожим образом:

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


date_trunc
-------------------
2020-10-01 00:00:00
```

**Таблица 3.2.** Стандартные значения для первого аргумента функции `date_trunc`

| Строковое значение | Единица времени |
|---|---|
| microsecond | микросекунда |
| millisecond | миллисекунда |
| second | секунда |
| minute | минута |
| hour | час |
| day | день |
| week | неделя |
| month | месяц |
| quarter | квартал |
| year | год |
| decade | десятилетие |
| century | столетие |
| millennium | тысячелетие |

Вместо того чтобы работать с полными датами или временными метками, иногда для анализа требуется часть даты или времени. Например, мы можем захотеть сгруппировать продажи по месяцам, дням недели или часам дня.

В SQL есть несколько функций для возврата только необходимой части даты или времени. Обычно в качестве аргумента можно передать и дату, и временную метку, за исключением случаев, когда запрос должен вернуть часть времени. В таких случаях, конечно, требуется временная метка.

Функция `date_part` принимает строковое значение для указания требуемой части и само значение даты или временной метки. Возвращаемое значение — вещественное число `FLOAT`, и при необходимости вы можете привести его к целочисленному типу:

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

Другая функция, работающая аналогичным образом, — `extract`, которая также принимает часть, которую надо вернуть, и значение даты или временной метки и возвращает значение `FLOAT`:

```
SELECT extract('day' from current_timestamp);
SELECT extract('month' from current_timestamp);
SELECT extract('hour' from current_timestamp);
```

Функции `date_part` и `extract` можно использовать и для интервалов, но обратите внимание, что запрашиваемая часть должна точно оответствовать единицам измерения интервала. Например, запрос дней для интервала, указанного в днях, возвращает нужное значение:

```
SELECT date_part('day',interval '30 days');

date_part
---------
30.0
```

Однако запрос дней для интервала, указанного в месяцах, вернет 0.0:

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

date_part
---------
0.0
```

Полный список частей даты/времени можно найти в документации к вашей базе данных или в интернете, но наиболее распространенными являются `'day'`, `'month'` и `'year'` для даты, `'second'`, `'minute'` и `'hour'` для времени.

Чтобы вернуть текстовые значения для частей даты, используйте функцию `to_char`, которая принимает в качестве аргументов значение даты или временной метки и требуемый формат:

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

Если вы когда-нибудь встретите значения, хранящиеся как Unix-время (количество секунд, прошедших с 1 января 1970 года 00:00:00 UTC), вы можете преобразовать их в обычные временные метки с помощью функции `to_timestamp`.

Иногда для анализа требуется собрать дату из частей, получаемых из разных источников. Например, это может понадобиться, когда значения года, месяца и дня хранятся в разных столбцах таблицы. Это также бывает необходимо, когда части даты достаются из текста (тема, которую мы рассмотрим более подробно в *гл. 5*).

Простой способ создать временную метку из отдельной даты и отдельного времени — соединить их вместе при помощи знака плюс (+):

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

timestamp
-------------------
2020-09-01 03:00:00
```

В разных базах данных функция для получения даты из частей может называться по-разному — `make_date`, `makedate`, `date_from_parts` или `datefromparts` — они все эквивалентны. Функция принимает в качестве аргументов год, месяц, день и возвращает значение типа даты:

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

make_date
----------
2020-09-01
```

Аргументы должны быть целочисленными и могут быть константами или полями таблицы. Еще один способ собрать дату или временную метку — соединить значения с помощью функции `concat`, а затем преобразовать результат к дате, используя обычный синтаксис или функцию `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);

concat
----------
2020-09-01
```

Как вы видите, в SQL есть разные способы форматирования и преобразования дат и временных меток, а также получения системного времени. В следующем разделе я буду их использовать при выполнении вычислений с датами.

## Арифметические действия с датами

SQL позволяет нам выполнять различные арифметические операции над датами. Это может показаться странным, поскольку, строго говоря, дата не является числовым типом данных. Но эта концепция должна быть вам знакома, если вы когда-нибудь пытались вычислить по календарю, какая дата будет через четыре недели. Арифметика с датами пригодится для множества аналитических задач. Например, мы будем использовать ее, чтобы определить возраст клиента или как давно он зарегистрирован, сколько времени прошло между двумя событиями или сколько событий произошло за определенный промежуток времени.

Арифметика с датами использует два типа данных: сами даты и интервалы. Здесь не обойтись без интервалов, потому что составные части даты и времени не ведут себя как целые числа. Одна десятая от 100 равна 10; одна десятая от года составляет 36,5 дней. Половина от 100 — это 50; половина дня — это 12 часов. Интервалы позволяют нам гибко пользоваться разными единицами времени. Интервалы бывают двух типов: год-месяц и день-время. Мы начнем с нескольких операций, которые возвращают целочисленные значения, а затем рассмотрим функции, которые работают с интервалами или возвращают их.

Для начала давайте найдем количество дней между двумя датами. Есть несколько способов сделать это в SQL. Первый способ заключается в использовании арифметического оператора вычитания — знак минус (–):

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

days
----
30
```

Эта операция возвращает количество дней между этими двумя датами. Обратите внимание, что результат равен 30 дням, а не 31-му. Результат включает только одну из конечных дат. Вычитание дат в обратном порядке также работает и возвращает интервал в –30 дней:

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

days
----
-30
```

Найти разницу между двумя датами можно также с помощью функции `datediff`. Postgres не поддерживает ее, но она есть во многих других базах данных, включая

Microsoft SQL Server, Redshift и Snowflake. Эта функция очень удобна в случаях, когда результатом будет интервал, отличный от количества дней. Функция принимает три аргумента: единица времени, в которой будет выражен результат, начальная дата или временная метка, конечная дата или временная метка:

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

Таким образом, наш предыдущий пример будет выглядеть так:

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

days
----
30
```

Мы также можем найти количество полных месяцев между двумя датами, и база данных выполнит правильные вычисления, даже если продолжительность месяцев в течение года разная:

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

months
------
5
```

В Postgres это можно сделать с помощью функции `age`, которая вычисляет интервал между двумя датами:

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

age
--------------
5 mons 29 days
```

После этого мы можем найти количество месяцев в интервале с помощью функции `date_part`:

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

months
------
5.0
```

Вычитание дат, чтобы определить интервал времени между ними, является очень удобной операцией. Сложение дат работает иначе. Чтобы выполнять сложение, нужно использовать интервалы или специальные функции.

Например, мы можем добавить к дате семь дней, прибавив интервал '7 days':

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

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

Некоторые базы данных не требуют такого синтаксиса интервала и автоматически преобразуют указанное число в дни, хотя рекомендуется все-таки использовать полную запись интервала как для совместимости между базами данных, так и для читабельности кода:

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

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

Если вы хотите прибавить другую единицу времени, используйте запись интервала с месяцами, годами, часами или другой частью даты и времени. Обратите внимание, что такой же подход можно использовать для вычитания интервала из даты, используя оператор (–) вместо (+). Во многих (но не во всех) базах данных есть функция date_add или dateadd, которая принимает на вход интервал и начальную дату и выполняет сложение:

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

new_date
----------
2020-07-01
```

Обратитесь к документации по вашей базе данных или просто поэкспериментируйте с запросами, чтобы выяснить синтаксис и доступные вам функции.

Любая из этих записей может быть использована в разделе WHERE в дополнение к оператору SELECT. Например, мы можем отфильтровать записи, которые были внесены более трех месяцев назад:

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

Эти операции также можно использовать в условиях JOIN, но обратите внимание, что производительность выполнения такого запроса будет ниже, если условие JOIN содержит вычисления, а не простое сравнение двух дат.

Арифметические операции с датами широко используются при анализе с помощью SQL как для нахождения временного интервала между датами или временными метками, так и для вычисления новых дат на основе интервала от известной даты. Далее мы рассмотрим аналогичные операции со временем.

## Арифметические действия со временем

Арифметические операции со временем менее распространены в анализе, но могут быть полезны в ряде случаев. Например, мы можем захотеть узнать, сколько времени требуется сотруднику службы поддержки, чтобы ответить на телефонный звонок в колл-центр или на электронное письмо с просьбой о помощи. Всякий раз, когда время, прошедшее между двумя событиями, меньше суток или когда округление результата до количества дней не дает достаточно информации, приходится выполнять вычисления со временем. Арифметика со временем работает аналогично арифметике с датами с использованием интервалов. Мы можем добавлять временные интервалы ко времени:

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

new_time
--------
08:00:00
```

Мы можем вычитать интервалы из времени:

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

new_time
--------
02:00:00
```

Можно также вычитать времена, в результате чего получится интервал:

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

time_diff
---------
02:00:00
```

Времена, в отличие от дат, можно умножать на число:

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

time_multiplied
---------------
10:00:00
```

Интервалы также можно умножать на число, в результате чего получим время или интервал:

```
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
```

В этих примерах используются константы, но вы можете указывать имена полей или выражения в запросах SQL, чтобы сделать такие вычисления динамичными. Далее я расскажу о некоторых особенностях при работе с датами, которые следует учитывать при объединении наборов данных из разных исходных систем.

## Объединение данных из разных источников

Объединение данных из разных источников — одна из наиболее веских причин использования хранилища данных. Но разные исходные системы могут записывать дату и время в разных форматах или в разных часовых поясах, или даже немного отставать друг от друга из-за проблем с системным временем на сервере. Даже таблицы из одного и того же источника данных могут иметь различия, хотя это встречается очень редко. Согласование и стандартизация дат и временных меток — важный шаг перед выполнением анализа.

Даты и временные метки в разных форматах можно стандартизировать с помощью SQL. Выполнение соединения `JOIN` по датам или добавление полей с датами с помощью `UNION` обычно требует, чтобы даты и временные метки были в одном и том же формате. Ранее в этой главе я показала способы форматирования дат и временных меток, которые хорошо подходят для решения этих проблем. Не забывайте про часовые пояса при объединении данных из разных источников. Например, внутренняя база данных может использовать время UTC, а данные от третьей стороны могут быть в местном часовом поясе. Я видела данные, полученные от SaaS-сервиса, которые были записаны по местному времени для разных часовых поясов. Обратите внимание, что сами значения временных меток не обязательно будут содержать часовой пояс. Возможно, вам придется обратиться к документации поставщика и преобразовать данные в UTC, если остальные даты тоже хранятся в этом формате. Другой вариант — сохранить часовой пояс в отдельном поле, чтобы значение временной метки можно было преобразовать при необходимости.

Еще один момент, на который следует обратить внимание при работе с данными из разных источников, — это временные метки, которые немного рассинхронизированы. Это может произойти, когда для одного источника данных записывается время клиентского устройства, например ноутбука или мобильного телефона, а для другого источника — время сервера. Однажды я столкнулась с тем, что результаты ряда экспериментов были рассчитаны неверно, потому что время на клиентском устройстве, фиксировавшем действия пользователя, отличалось на несколько минут от времени сервера, записывавшего данные о терапевтической группе, которую он посещал. Данные клиентов, по-видимому, поступали раньше, чем выполнялась обработка данных группы, поэтому некоторые события случайно исключались из рассмотрения. Исправить эту ошибку достаточно просто: при выборке данных вместо того, чтобы сравнивать время действий клиентов со временем группы, включите в результаты и действия, произошедшие в пределах короткого интервала или временного окна около времени группы. Это можно сделать с помощью оператора `BETWEEN` и арифметических операций с датами, как было показано в предыдущем разделе.

При работе с данными из мобильных приложений обратите особое внимание на то, что именно обозначают временные метки — когда действие произошло на устройстве или когда информация об этом событии поступила в базу данных. Разница во времени может быть незначительной, а может быть и несколько дней, в зависимости от того, разрешено ли использование мобильного приложения в автономном режиме (offline) и как оно выполняет отправку данных при низком уровне сигнала. Данные из мобильных приложений могут поступать в базу данных с опозданием или даже через несколько дней после того, как они были получены на устройстве. Кроме того, даты и временные метки могут быть повреждены при передаче данных, и в результате вы можете получить невероятно далекое прошлое или далекое будущее.

Теперь, когда мы разобрались, как форматировать даты и временные метки, преобразовывать часовые пояса, выполнять арифметические операции с датами и работать с наборами данных из разных источников, мы можем перейти к некоторым примерам временных рядов. Сначала я расскажу о наборе данных, который мы будем использовать в следующих разделах этой главы.

## 3.2. Набор данных о розничных продажах

Далее в примерах этой главы используется набор данных о розничных продажах в США из ежемесячного отчета Monthly Retail Trade Report — файл **Retail and Food Services Sales Excel (1992 - Present)**, доступный на веб-сайте Бюро переписи населения **Census.gov**[2]. Данные из этого отчета используются в качестве экономического индикатора для понимания трендов потребительских расходов в США. Так как

[2] **https://www.census.gov/retail/index.html#mrts**

данные о валовом внутреннем продукте (ВВП) публикуются ежеквартально, а данные о розничных продажах публикуются каждый месяц, то они также используются для прогнозирования ВВП. Поэтому когда данные о розничных продажах публикуются, об этом обычно пишет деловая пресса.

Данные охватывают период с 1992 по 2020 г. и включают в себя как общий объем продаж, так и детализацию по категориям розничных продаж. Данные содержат неизмененные цифры (not adjusted) и цифры, скорректированные с учетом сезонных колебаний (adjusted). В этой главе мы будем использовать неизмененные цифры, поскольку одной из наших целей является анализ сезонности. Данные о продажах указаны в миллионах долларов США. Исходный формат файла — Excel-файл с вкладками для каждого года, каждый месяц занимает отдельный столбец. В репозитории GitHub[3] для этой книги выложены данные в формате CSV, который легко импортировать в базу данных, а также код, специально написанный для импорта файла в Postgres. На рис. 3.1 показана выборка из таблицы `retail_sales`.

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

**Рис. 3.1.** Фрагмент данных о розничных продажах в США

## 3.3. Анализ трендов данных

Работая с временными рядами, мы часто хотим найти тренды в данных. Тренд — это просто направление, в котором движутся данные. Он может быть восходящим (увеличиваться с течением времени) или нисходящим (уменьшаться с течением времени). Тренд может оставаться более или менее горизонтальным или в данных

[3] **https://github.com/cathytanimura/sql_book/tree/master/Chapter 3: Time Series Analysis**

может быть так много шума или постоянных скачков вверх/вниз, что тренд вообще трудно определить. В этом разделе мы рассмотрим несколько способов определения трендов временного ряда, начиная с простых трендов, сравнения составляющих временного ряда, процентных отношений для сравнения частей с целым и заканчивая приведением к базовому периоду, чтобы увидеть процентное изменение по сравнению с базовым годом.

## Простые тренды

Определение тренда может быть одним из шагов профилирования данных, а может быть и конечным результатом анализа. Набор данных представляет собой последовательность дат или временных меток и числовых значений. При построении графика временного ряда даты или временные метки откладываются по оси *x*, а числовые значения — по оси *y*. Например, мы можем проверить динамику общих (total) продаж в розничной торговле и общественном питании в США:

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

Результаты представлены в виде графика на рис. 3.2.

Эти данные явно имеют определенную закономерность, но в них также есть и некоторый шум. Мы можем преобразовать данные и агрегировать их по годам, что поможет нам лучше понять ситуацию. Сначала мы воспользуемся функцией `date_part`, чтобы вернуть только год из поля `sales_month`, а затем просуммируем продажи `sales`. Результаты снова отфильтруем по `kind_of_business` с тем же условием, чтобы работать только с общими продажами:

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

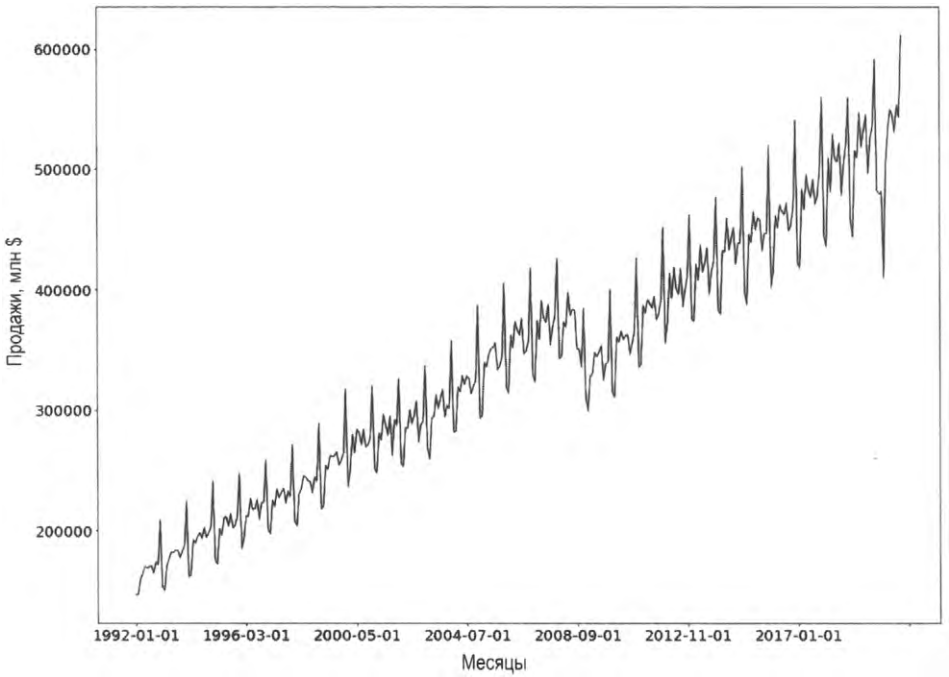


**Рис. 3.2.** Динамика ежемесячных продаж в розничной торговле и общественном питании

Судя по графику, показанному на рис. 3.3, мы теперь имеем более плавный временной ряд, который увеличивается со временем, что и можно было ожидать, поскольку объемы продаж не корректировались на величину инфляции. Продажи всей розничной торговли и общественного питания упали в 2009 г. из-за мирового финансового кризиса. После роста в течение всех 2010-х гг. продажи в 2020 г. не изменились по сравнению с 2019 г. из-за пандемии COVID-19.

Графики временных рядов с разным уровнем агрегирования — еженедельный, ежемесячный или годовой — являются хорошим способом определения трендов. Этот шаг можно использовать для простого профилирования данных, но он также может быть включен в окончательный результат, в зависимости от целей анализа. Далее мы перейдем к использованию SQL для сравнения временных рядов разных категорий розничных продаж.

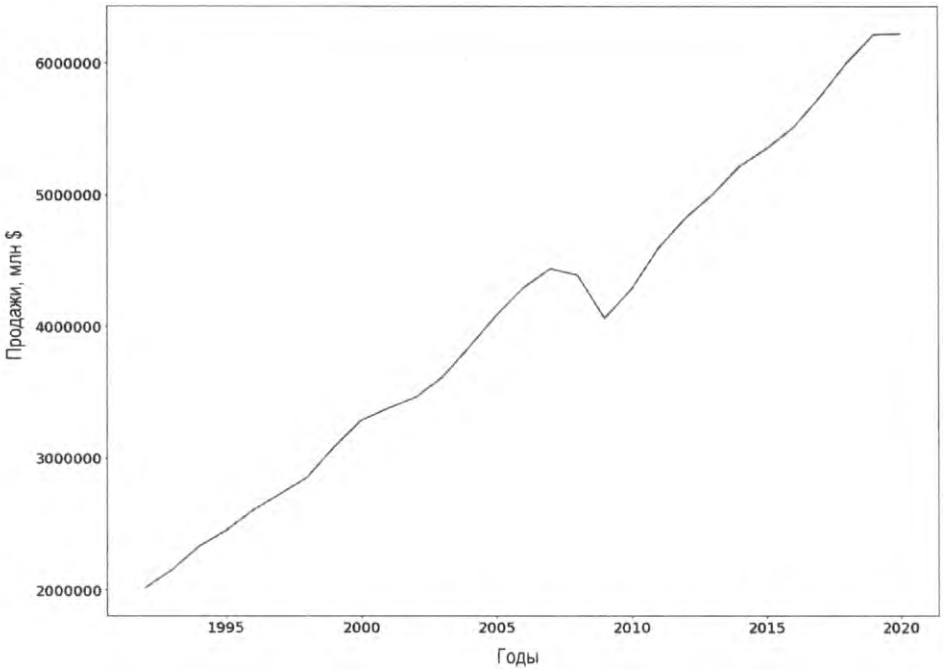


**Рис. 3.3.** Динамика годового объема продаж в розничной торговле и общественном питании

## Сравнение временных рядов

Наборы данных часто содержат не просто один временной ряд, а несколько частей или составляющих итоговых значений в одном и том же временном диапазоне. Сравнение этих составляющих может выявить интересные закономерности. В нашем наборе данных о розничных продажах есть не только общие объемы продаж, но и ряд категорий. Давайте сравним ежегодную динамику продаж для нескольких категорий, связанных с досугом: книжные магазины, магазины спортивных товаров и магазины товаров для хобби, игрушек и игр. Для этого добавим `kind_of_business` в операторе `SELECT` и, поскольку это еще одно поле, а не агрегация, то его нужно добавить и в раздел `GROUP BY`:

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

```
sales_year  kind_of_business             sales
----------  --------------------------  -----
1992.0      Book stores                 8327
1992.0      Hobby, toy, and game stores 11251
1992.0      Sporting goods stores       15583
...         ...                         ...
```

Результаты представлены в виде графика на рис. 3.4. Розничные продажи в магазинах спортивных товаров вначале были самыми высокими среди трех категорий и росли намного быстрее в течение наблюдаемого периода времени, а к концу этого периода стали значительно выше. В 2017 г. продажи спортивных товаров начали снижаться, но резко выросли в 2020 г. Продажи товаров для хобби, игрушек и игр в наблюдаемый период оставались относительно неизменными, с небольшим падением в середине 2000-х гг. и еще одним небольшим спадом перед началом кризиса 2017 г. с последующим восстановлением в 2020 г. Продажи в книжных магазинах росли до середины 2000-х гг. и с тех пор постоянно снижаются. На все эти категории повлиял рост количества интернет-магазинов, но сроки и масштабы влияния, по-видимому, различаются.

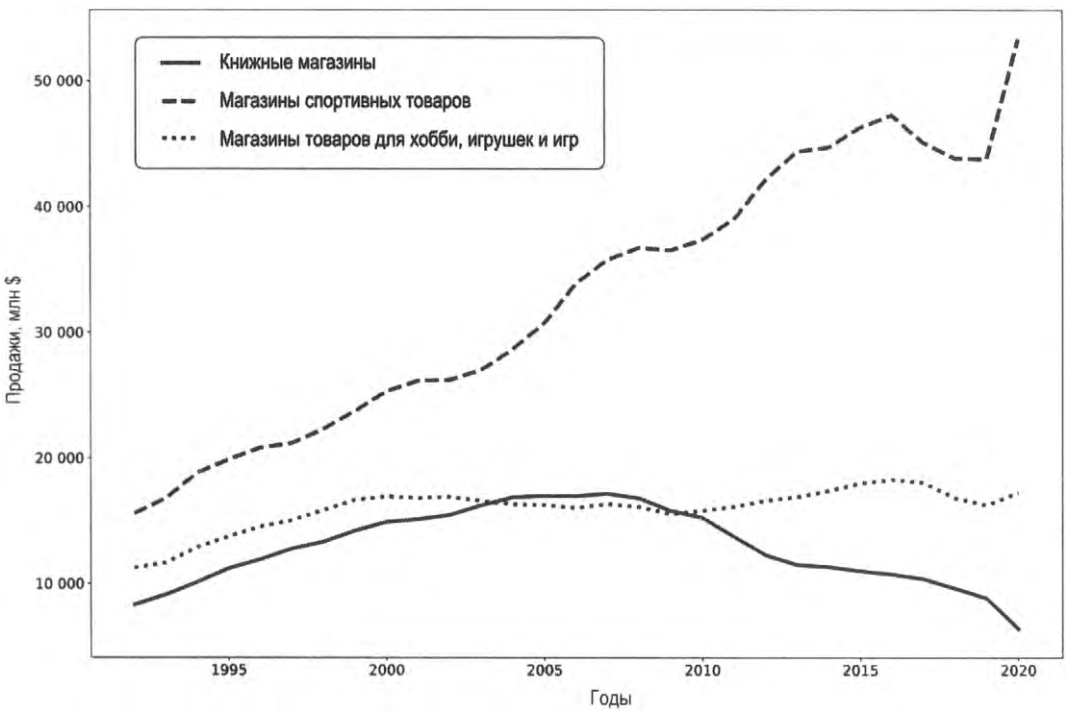


**Рис. 3.4.** Динамика ежегодных розничных продаж книжных магазинов, магазинов спортивных товаров и магазинов хобби, игрушек и игр

Помимо анализа простых трендов, мы можем выполнить более сложные сравнения составляющих временного ряда. В следующих примерах давайте рассмотрим про-

дажи в магазинах женской и мужской одежды. Обратите внимание, что поскольку названия этих категорий содержат символ апострофа ('), который также используется для обозначения начала и конца строк, нам нужно экранировать его с помощью дополнительного апострофа. Это указывает на то, что апостроф является частью строки. Хотя мы могли бы добавить шаг при загрузке данных, который бы удалял апострофы в названиях категорий, я оставила их, чтобы обратить ваше внимание, что такая корректировка кода очень часто требуется для реальных данных. Для начала давайте проанализируем продажи в каждой категории по месяцам:

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

Результаты представлены в виде графика на рис. 3.5. Объем розничных продаж в магазинах женской одежды намного выше, чем в магазинах мужской одежды. Продажи в обеих категориях подвержены сезонности, но мы рассмотрим это подробнее в *разд. 3.5*. В 2020 г. в обоих типах магазинов произошло сильное падение из-за закрытия магазинов и сокращения количества покупок в пандемию COVID-19.

Ежемесячные данные имеют любопытные закономерности, но зашумлены, поэтому далее мы будем рассматривать агрегацию по годам. Мы уже использовали похожий запрос ранее при агрегации общих продаж и продаж для категорий досуга:

```
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
;
```

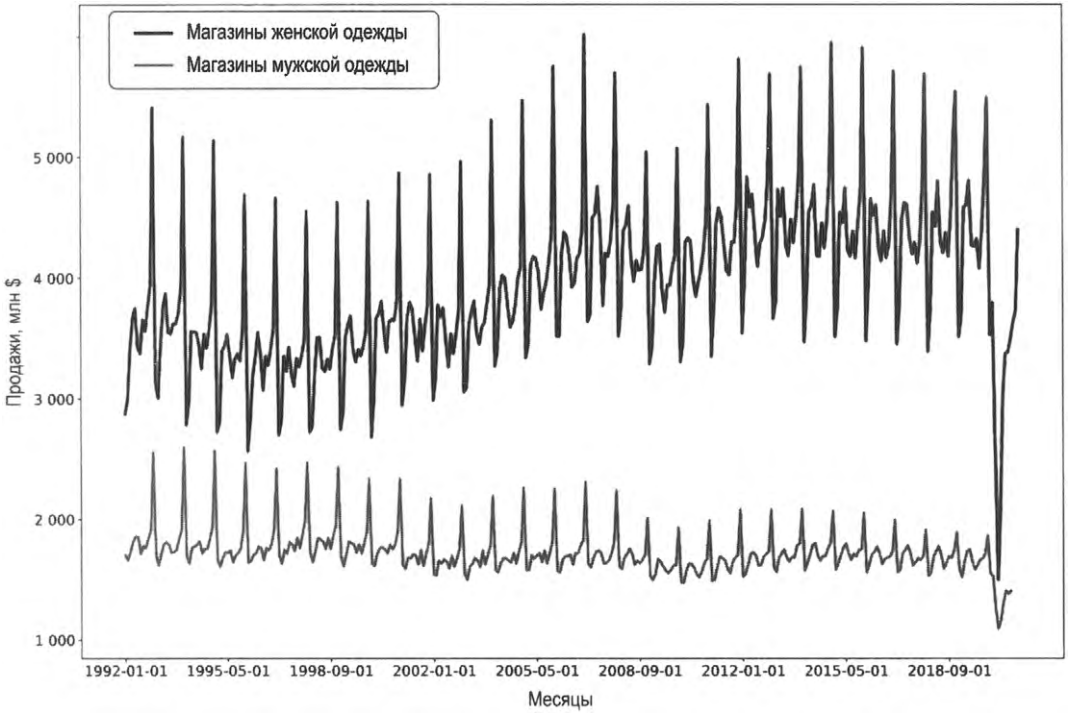


**Рис. 3.5.** Ежемесячная динамика продаж в магазинах женской и мужской одежды

Является ли разница между продажами в магазинах женской и мужской одежды всегда одинаковой? По годовой динамике, показанной на рис. 3.6, видно, что разрыв между продажами одежды для мужчин и женщин не был постоянным, а увеличивался в первой половине 2000-х гг. Продажи женской одежды заметно упали во время мирового финансового кризиса 2008–2009 г. и продажи в обеих категориях сильно упали во время пандемии в 2020 г.

Однако нам не нужно опираться только на визуальную оценку. Для большей точности мы можем вычислить разницу между этими двумя категориями, отношение и процентную разницу между ними. Для этого сначала организуем данные так, чтобы для каждого месяца была одна строка со столбцами для каждой категории. Соединим агрегатные функции с операторами CASE:

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

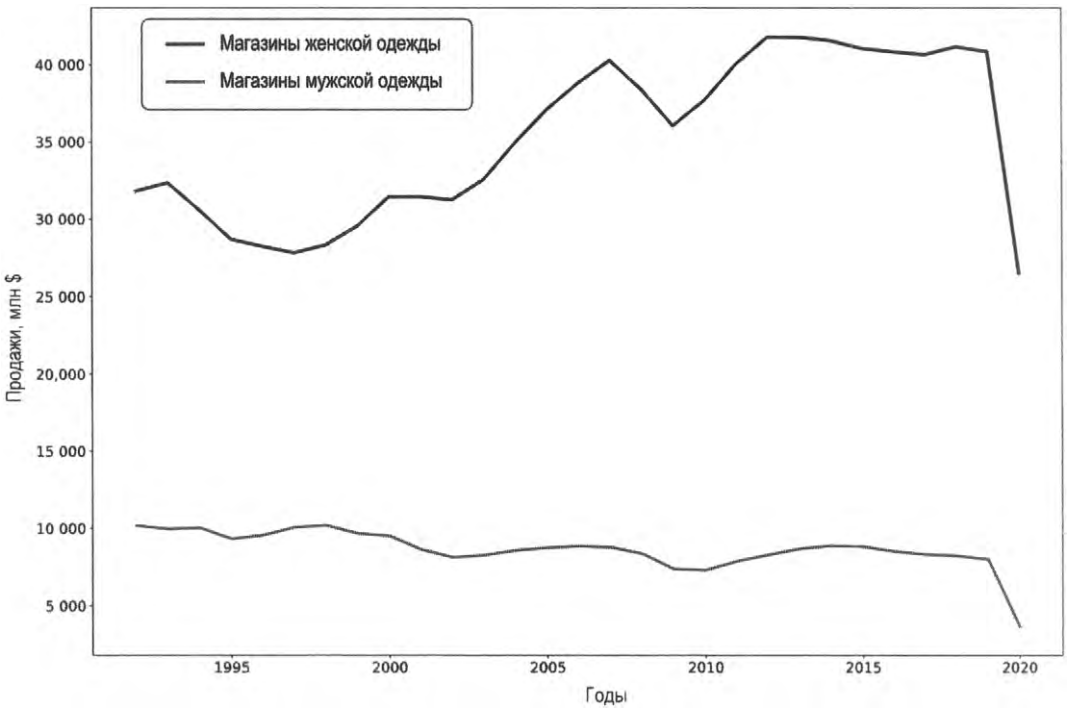


**Рис. 3.6.** Динамика продаж в магазинах женской и мужской одежды по годам

С помощью этого вычисления мы можем найти разницу, отношение и процентную разницу между временными рядами из набора данных. Разницу можно получить, просто вычитая одно значение из другого с помощью оператора (–). В зависимости от целей анализа может пригодиться вычитание одной категории из другой или наоборот. Я приведу обе разницы, хотя они эквивалентны, за исключением знака. Об-

ратите внимание, что в WHERE добавлен фильтр для исключения 2020 года из рассмотрения, поскольку в некоторых месяцах этого года есть null-значения[4]:

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

Здесь можно было и не использовать подзапрос, поскольку агрегирующие функции можно складывать и вычитать. Код с подзапросом чаще всего более понятен, но из-за этого он становится длиннее. В зависимости от того, насколько длинной или сложной является остальная часть вашего SQL-запроса, вы можете выделить промежуточные вычисления в подзапрос или просто использовать их в основном запросе. Приведу пример без подзапроса, где из продаж женской одежды вычитаются продажи мужской одежды:

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

[4] Данные за октябрь и ноябрь 2020 г. по продажам в магазинах мужской одежды были скрыты (supressed) издателем из-за опасений по поводу качества данных. Вероятно, сбор данных усложнился из-за закрытия магазинов во время пандемии.

```
 then sales end)
  -
 sum(case when kind_of_business = 'Men''s clothing stores'
 then sales end)
 as womens_minus_mens
FROM retail_sales
WHERE kind_of_business in ('Men''s clothing stores'
 ,'Women''s clothing stores')
and sales_month <= '2019-12-01'
GROUP BY 1
;

sales_year  womens_minus_mens
----------  -----------------
1992.0      21636
1993.0      22388
1994.0      20553
...         ...
```

Из графика на рис. 3.7 видно, что разрыв сокращался с 1992 по 1997 г., затем начался продолжительный рост примерно до 2011 г. (с коротким спадом в 2007 г.), а после этого оставался более или менее ровным до 2019 г.

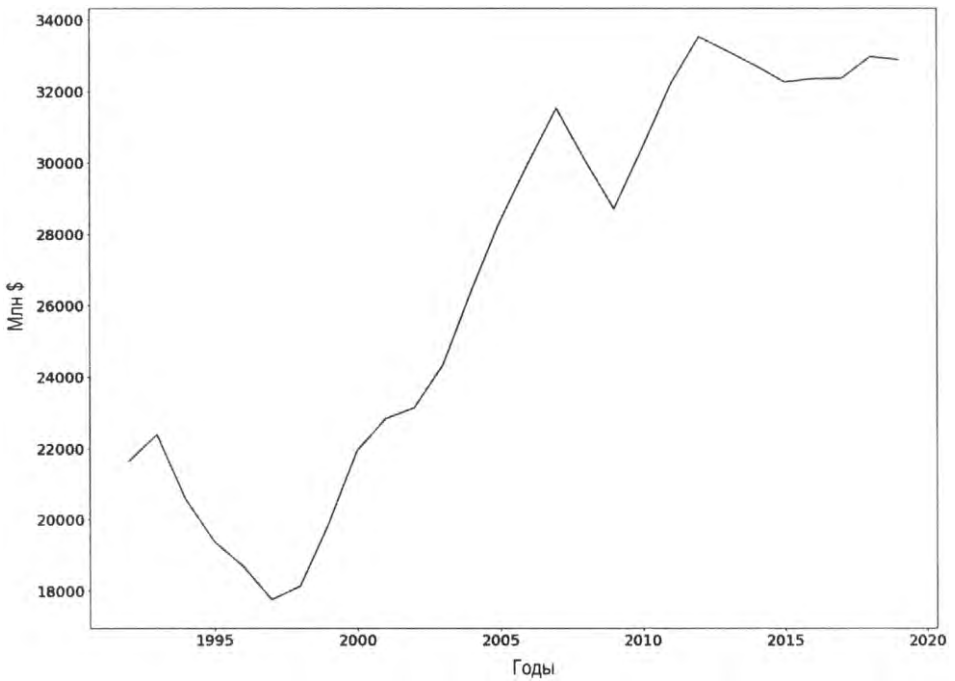


**Рис. 3.7.** Разница между продажами в магазинах женской и мужской одежды по годам

Продолжим наше исследование и посмотрим на отношение этих категорий. Мы будем использовать продажи в магазинах мужской одежды в качестве базового уровня или знаменателя, но обратите внимание, что здесь легко можно было бы взять в качестве базового уровня продажи в магазинах женской одежды:

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

sales_year  womens_times_of_mens
----------  --------------------
1992.0      3.1255526083112290
1993.0      3.2473398915880345
1994.0      3.0487440191387560
...         ...
```

При делении чисел SQL возвращает много десятичных цифр. Вам следует выполнить округление перед представлением результатов анализа. Используйте такой уровень точности (количество знаков после запятой), которого будет достаточно для наглядности результатов.

График на рис. 3.8 показывает, что тенденции отношения аналогичны тенденциям разницы, но, несмотря на то, что в 2009 г. было падение в разнице, на самом деле отношение увеличилось.

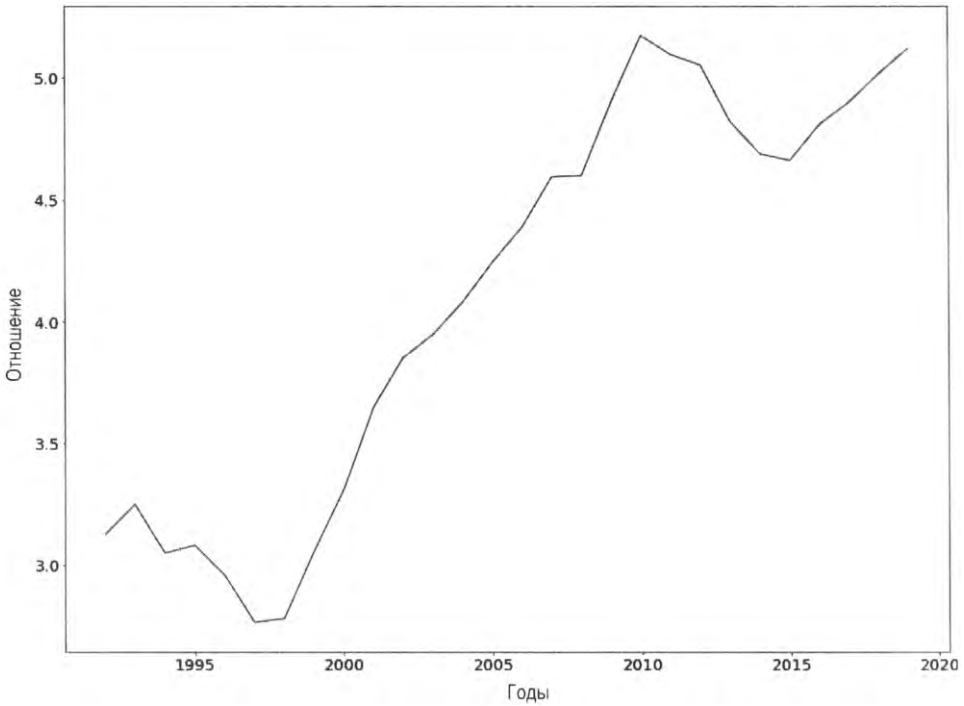


**Рис. 3.8.** Отношение продаж в магазинах женской и мужской одежды по годам

Далее мы можем рассчитать процентную разницу между продажами в магазинах женской и мужской одежды:

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

Хотя единицы измерения этих значений отличаются от единиц измерения в предыдущем примере, форма этого графика такая же, как и у графика отношения. Выбор способа сравнения зависит от вашей целевой аудитории и предметной области, в которой вы работаете. Все наши выводы верны: в 2009 г. продажи в магазинах женской одежды были на 28,7 млрд долларов выше, чем продажи в магазинах мужской одежды; в 2009 г. продажи в магазинах женской одежды в 4.9 раз превышали продажи в магазинах мужской одежды; в 2009 г. продажи в женских магазинах были на 390% выше, чем в мужских. Какую версию выбрать, зависит от истории, которую вы хотите рассказать с помощью анализа.

Преобразования, которые мы сделали в этом разделе, позволяют нам анализировать временные ряды и сравнивать родственные категории. Далее мы продолжим сравнивать временные ряды, которые являются частями целого.

## Вычисление процента от целого

При работе с временными рядами, которые состоят из многих составляющих или частей, которые складываются в единое целое, бывает полезно проанализировать вклад каждой части в целое и выяснить, как он менялся с течением времени. Если набор данные не содержит временного ряда с итоговыми значениями, нужно сначала вычислить общую сумму, чтобы потом определить процент от нее для каждой составляющей. Этого можно добиться с помощью self-JOIN или оконной функции, которая, как мы видели в *разд. 2.3*, представляет собой особый вид функции SQL, способный ссылаться на любую строку таблицы.

Сначала я расскажу как работает self-JOIN. Это «самосоединение» — любое соединение таблицы с самой собой. Если каждому указанному экземпляру таблицы в запросе присвоен свой псевдоним, база данных будет рассматривать их как отдельные таблицы. Например, чтобы найти проценты продаж мужской и женской одежды от их совокупного объема, мы можем соединить `retail_sales` с псевдонимом `a` и `retail_sales` с псевдонимом `b` с помощью `JOIN` по полю `sales_month`. При этом в `SELECT` мы выбираем название категории (`kind_of_business`) и объем продаж `sales` из таблицы `a`. Затем из таблицы `b` мы находим `sum` для продаж в обеих категориях и обозначаем результат как `total_sales`. Обратите внимание, что `JOIN` таблиц по полю `sales_month` возвращает частичное декартово соединение, т. е. каждой строке из таблицы `a` соответствуют две строки из таблицы `b`. Однако группировка по полям `a.sales_month`, `a.kind_of_business` и `a.sales` и агрегирование по `b.sales` возвращает

именно те результаты, которые необходимы. Во внешнем запросе процент от целого для каждой строки рассчитывается путем деления sales на 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
...          ...                     ...
```

Здесь тоже необязательно использовать подзапрос, но он делает код немного проще для понимания. Второй способ для расчета для каждой категории процента от общего объема продаж — использовать оконную функцию sum и указать в PARTITION BY поле sales_month. Напомним, что оператор PARTITION BY указывает на то, по которому полю делается группировка, чтобы выполнять вычисления в пределах каждой группы. Оператор ORDER BY здесь не требуется, поскольку порядок вычисления не имеет значения. Кроме того, в запросе не нужен оператор GROUP BY, т. к. оконные функции просматривают несколько строк, но не сокращают количество строк в результирующем наборе:

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

График с этими данными, показанный на рис. 3.9, позволяет выявить некоторые интересные тенденции. Во-первых, начиная с конца 1990-х гг. доля продаж женской одежды в общем объеме продаж одежды увеличивалась. Во-вторых, в начале ряда очевидна сезонная закономерность, когда доля продаж мужской одежды резко увеличивается в декабре и январе. В первом десятилетии XXI в. появляются два сезонных пика, летом и зимой, но к концу 2010-х сезонные закономерности затухают почти до случайности. Далее в *разд. 3.5* мы более подробно рассмотрим анализ сезонности.

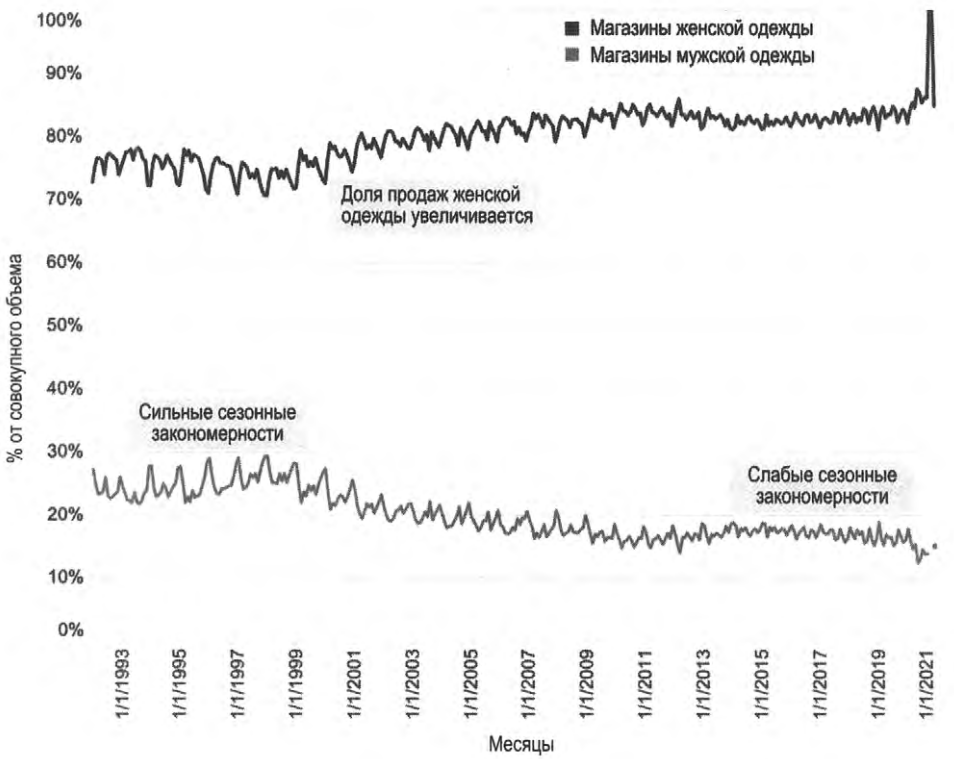


**Рис. 3.9.** Проценты продаж в магазинах мужской и женской одежды от их совокупного объема по месяцам

Другой процент, который нам может понадобиться, — это доля в продажах за более длительный временной период, например процент продаж каждого месяца от

годового объема. Опять же можно использовать либо self-JOIN, либо оконную функцию. Приведу сначала пример с self-JOIN в подзапросе:

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

То же самое с помощью оконных функций:

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

Результаты для 2019 года показаны на рис. 3.10. Эти два временных ряда очень похожи, но доля продаж в мужских магазинах в январе была выше, чем в женских магазинах. В мужских магазинах в июле наблюдался летний спад, в то время как в женских магазинах соответствующий спад произошел в сентябре.

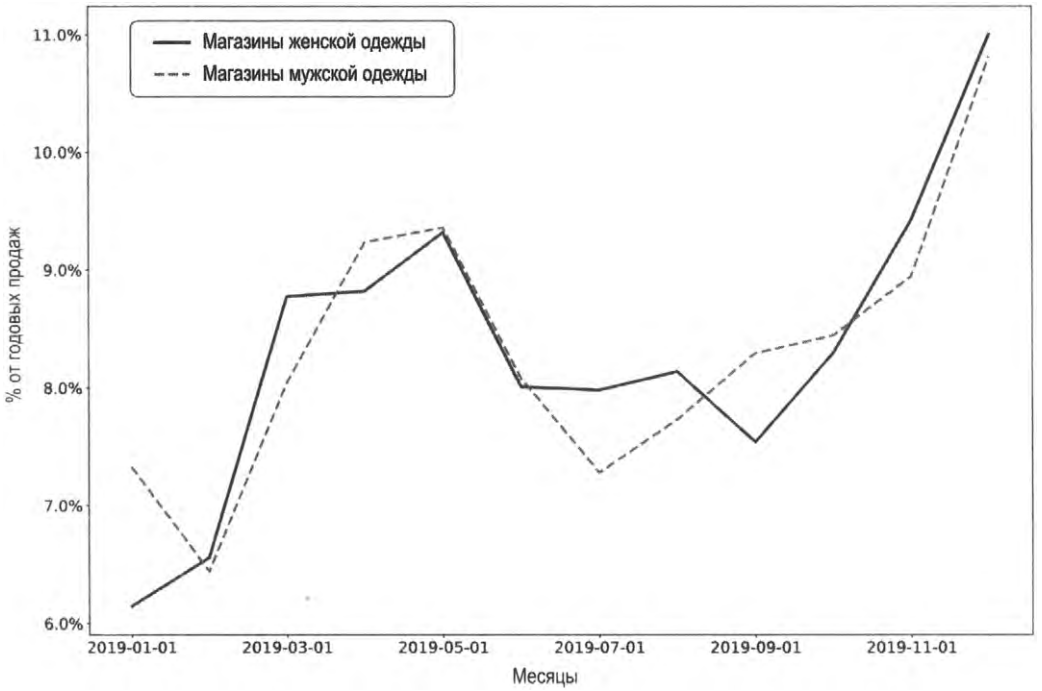


**Рис. 3.10.** Процент продаж женской и мужской одежды от годового объема за 2019 год

Теперь, когда я показала, как использовать SQL для вычисления процента от общей суммы и какие типы анализа можно выполнить, давайте рассмотрим приведение к базовому периоду и вычисление процентного изменения с течением времени.

## Приведение к базовому периоду

Значения временного ряда обычно меняются со временем. Продажи увеличиваются с ростом популярности и доступности продукта, а время отклика веб-страницы уменьшается благодаря усилиям разработчиков по оптимизации кода. Приведение данных — это способ оценить изменения временного ряда по отношению к базово-

му периоду (начальной точке). Этот способ широко используются как в экономике, так и в бизнесе. С его помощью рассчитывается один из самых известных индексов — индекс потребительских цен (ИПЦ, или consumer price index, CPI), отслеживающий изменение цен на товары, которые покупает рядовой потребитель, и использующийся для оценки инфляции, принятия решения о повышении заработной платы и во многих других случаях. ИПЦ представляет собой сложный статистический показатель, использующий различные данные и удельные веса, но основная идея проста. Для расчета индекса нужно выбрать базовый период и для каждого следующего периода вычислить процентное изменение значений по сравнению с этим базовым периодом.

Приведение временных рядов с помощью SQL может быть выполнено с помощью агрегирования и оконных функций или self-JOIN. В качестве примера давайте выполним приведение объемов продаж в магазинах женской одежды к первому году в наборе — к 1992-му. Первым шагом является агрегирование продаж `sales` по `sales_year` в подзапросе, как мы делали ранее. В основном запросе используется оконная функция `first_value`, которая возвращает первое значение, соответствующее группировке `PARTITION BY` и сортировке `ORDER BY`. В этом примере мы можем опустить предложение `PARTITION BY`, потому что хотим получить значение продаж из первой строки всего набора данных, возвращаемого подзапросом:

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

С помощью этого запроса мы можем убедиться, что значение базового периода установлено правильно и соответствует 1992 году. Затем для каждой строчки найдем процентное изменение по сравнению с этим базовым годом:

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

Процентное изменение может быть как положительным, так и отрицательным, что и получилось для этого временного ряда. Оконную функцию `first_value` можно заменить на `last_value` в этом запросе. Однако индексирование по последнему временному периоду встречается гораздо реже, поскольку вопросы, ставящиеся перед анализом, чаще всего касаются изменений относительно начальной точки, а не относительно конечной точки; но, тем не менее, вариации возможны. Кроме того, для приведения к первому или к последнему периоду можно просто изменить порядок сортировки с помощью `ASC` и `DESC`:

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

Оконные функции обеспечивают большую гибкость. Но приведение может быть выполнено и без них с помощью нескольких последовательных self-JOIN, хотя это требует больше строк кода:

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

Обратите внимание на необычный оператор `JOIN` с условием соединения `on 1 = 1` между таблицей `aa` и подзапросом `bb`. Поскольку подзапрос `bb` возвращает только одну запись для базового периода и мы хотим, чтобы значение `bb.index_sales` добавилось в каждой строке результирующего набора, то мы не можем выполнить `JOIN` по году или по другому полю, чтобы не ограничить результат. Поэтому мы можем использовать здесь любое выражение, которое всегда будет равно `TRUE`, чтобы создать нужное декартово соединение. Можно использовать любое условие, возвращающее `TRUE`, например `on 2 = 2` или `on 'apples' = 'apples'`.

При выполнении операций деления, таких как `sales / index_sales` в последнем примере, будьте аккуратны с возможными нолями в знаменателе. Базы данных возвращают ошибку, когда сталкиваются с делением на ноль, что может раздражать. Даже если вы считаете, что ноль в поле знаменателя маловероятен, считается хорошей практикой предотвращать такие ошибки, указав с помощью оператора `CASE` значение по умолчанию, если знаменатель будет равен нулю. В примерах этого раздела нет нолей в знаменателях, поэтому я буду опускать этот дополнительный код для краткости.

В завершение этого раздела давайте посмотрим на график приведенных временных рядов для продаж в магазинах мужской и женской одежды, показанный на рис. 3.11. Код SQL выглядит так:

```
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
;
```

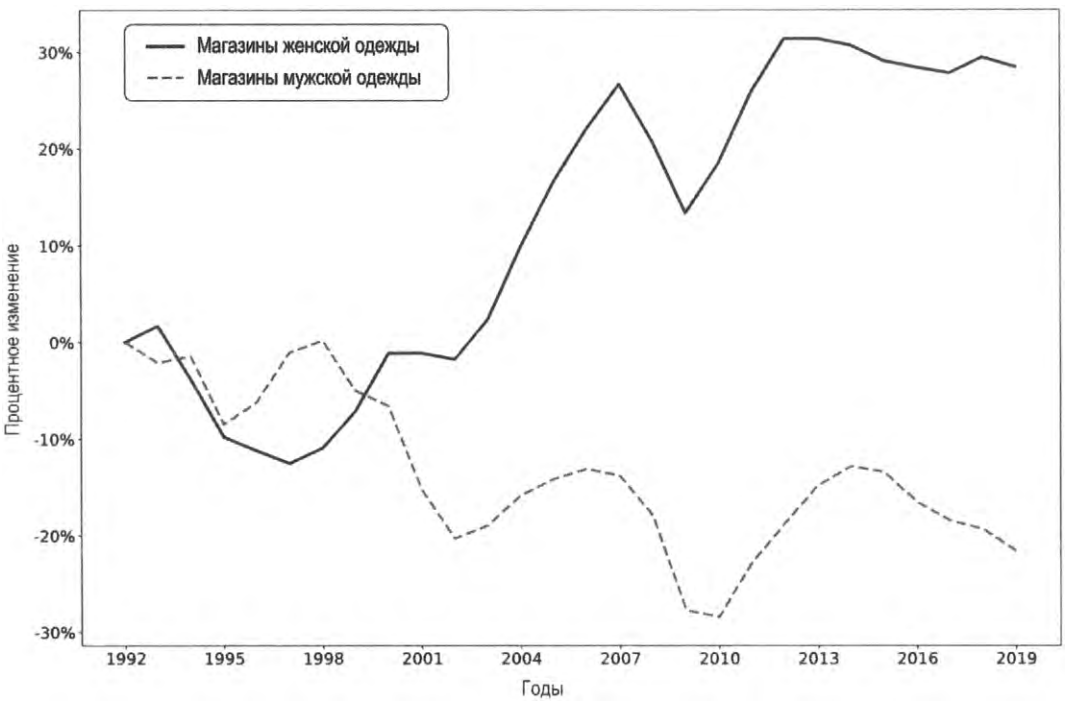


**Рис. 3.11.** Продажи в магазинах мужской и женской одежды, приведенные к 1992 году

Из этого графика видно, что 1992 год был своего рода высшей точкой продаж в магазинах мужской одежды. После него продажи упали, затем ненадолго вернулись к тому же уровню в 1998 г. и с тех пор продолжают снижаться. Это интересно, поскольку сам набор данных не скорректирован с учетом инфляции — постоянного роста цен с течением времени. Продажи в магазинах женской одежды сначала сни-

зились по сравнению с 1992 г., но к 2003 г. они вернулись к уровню 1992 г. С тех пор они постоянно росли, за исключением спада во время финансового кризиса, который повлиял на продажи в 2009—2010 гг. Одним из объяснений этих тенденций является то, что мужчины просто сократили расходы на одежду, возможно потому, что стали меньше следить за модой по сравнению с женщинами. Возможно, мужская одежда просто стала дешевле, поскольку глобальные цепочки поставок снизили затраты. Еще одним объяснением может быть то, что мужчины стали покупать одежду не в розничных магазинах одежды, отнесенных к категории "Men's clothing stores", а в магазинах других категорий, таких как магазины спортивных товаров или интернет-магазины.

Приведение данных временных рядов к базовому периоду — это эффективный метод анализа, позволяющий нам находить в наборе данных много ценных сведений. SQL хорошо подходит для решения этой задачи, и я показала, как выполнить приведение временных рядов с использованием оконных функций и без них. Далее мы разберем, как анализировать данные, используя скользящие временные окна, чтобы найти закономерности в зашумленных временных рядах.

## 3.4. Скользящие временные окна

Данные временных рядов очень часто зашумлены, что является проблемой для одной из наших основных целей — поиска закономерностей. Мы видели, как агрегирование данных, например переход от ежемесячных значений к годовым, может сгладить результаты и упростить их интерпретацию. Другой метод сглаживания данных — *скользящие временные окна*, также известный как скользящие вычисления, в которых учитываются несколько временных периодов. Метод скользящих средних, вероятно, наиболее широко известен, но благодаря SQL для анализа можно использовать любую агрегатную функцию. Скользящие временные окна применяются в самых разных видах анализа, в том числе для фондовых рынков, макроэкономических трендов и оценки аудитории. Некоторые вычисления используются настолько часто, что имеют собственные аббревиатуры: прошлые 12 месяцев (last twelve months, LTM), предыдущие 12 месяцев (trailing twelve months, TTM) и с начала года (year-to-date, YTD).

На рис. 3.12 показан пример скользящего временного окна и расчета накопительных сумм относительно октября месяца.

Есть несколько важных моментов при расчете любых скользящих временных окон. Во-первых, это размер окна, т. е. количество временных периодов, которые необходимо включить в расчет. Широкие окна с большим количеством временных периодов имеют бóльший эффект сглаживания, но есть риск потери чувствительности к важным краткосрочным изменениям данных. Более узкие окна с меньшим количеством периодов обеспечивают меньшее сглаживание, и, следовательно, они более чувствительны к краткосрочным изменениям, но при этом существует риск слишком слабого подавления шума.

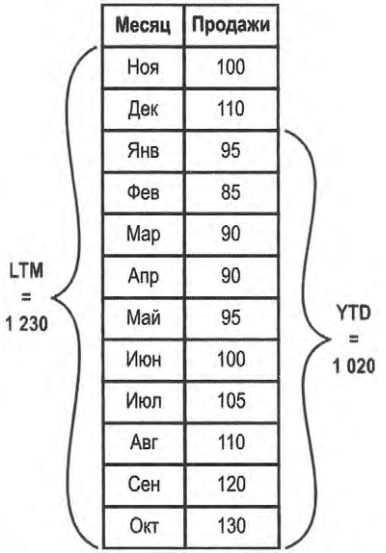


**Рис. 3.12.** Пример расчета скользящих сумм продаж LTM и YTD

Вторым важным моментом является используемая агрегатная функция. Как отмечалось ранее, метод скользящих средних, вероятно, наиболее распространен. С помощью SQL можно вычислить скользящие суммы, количество строк (`count`), минимумы и максимумы. Скользящий счетчик `count` полезен при оценке количества пользователей (см. следующую врезку). Скользящие минимумы и максимумы могут помочь при оценке крайних значений, необходимых для планирования анализа.

Еще один важный момент при расчете временных рядов — это выбор сегментирования или группировки данных, которые будут включены в окно. Анализ может потребовать сброса окна с началом каждого года. Или возникнет необходимость в разных скользящих функциях для каждой категории или группы пользователей. В *гл. 4* мы рассмотрим когортный анализ, в котором показатель удержания и накопительный итог сравниваются между группами с течением времени. Разбиение на категории можно выполнять с помощью группировки, а также с помощью `PARTITION BY` в оконных функциях.

Учитывая эти три момента, давайте перейдем к коду SQL и расчетам скользящих временных окон на примере того же набора данных о розничных продажах в США.

**Метрики «активные пользователи»: DAU, WAU и MAU**

Многие клиентские и некоторые B2B-приложения типа SaaS, чтобы оценить размер своей аудитории, отслеживают количество пользователей — ежедневные активные пользователи (daily active users, DAU), еженедельные активные пользователи (weekly active users, WAU) и ежемесячные активные пользователи (monthly active users, MAU). Поскольку все эти метрики используют скользящие временные окна, их можно рассчитать на каждый день. Меня часто спрашивают,

какой показатель лучше всего использовать, и я всегда отвечаю: «Все зависит от ваших обстоятельств».
DAU помогает компаниям с планированием пропускной способности, например с оценкой ожидаемой нагрузки на серверы. Однако, в зависимости от особенностей сервиса, вам могут потребоваться более подробные данные, например почасовая или даже поминутная информация об активных пользователях в пиковые нагрузки.

MAU обычно используется для оценки относительных размеров приложений и сервисов. Этот показатель удобен для оценки довольно стабильной или растущей группы пользователей, которая имеет устойчивые шаблоны поведения, не обязательно ежедневные, например более активное использование товаров для отдыха в выходные дни или более активное использование товаров, относящихся к работе или учебе, в будние дни. MAU не очень хорошо подходит для обнаружения скрытого оттока пользователей, которые перестают заходить в приложение. Поскольку окно для вычисления MAU, как правило, равно 30 дням, то пользователь может отсутствовать в приложении в течение 29 дней, прежде чем это отразится в снижении показателя MAU.

WAU, вычисляемый за период в 7 дней, может быть золотой серединой между DAU и MAU. WAU более чувствителен к краткосрочным изменениям, предупреждая об оттоках пользователей быстрее, чем MAU, и сглаживает разницу в днях недели, проявляющуюся в DAU. Недостатком WAU является то, что он чувствителен к краткосрочным изменениям, которые могут быть вызваны такими событиями, как длинные государственные праздники.

## Расчет скользящих временных окон

Теперь, когда мы знаем, что такое скользящие временные окна, для чего они нужны и из чего они состоят, давайте приступим к их расчету с использованием набора данных о розничных продажах в США. Мы начнем с более простого случая, когда набор данных содержит все записи для каждого периода, который попадает в скользящее временное окно, а в следующем разделе мы рассмотрим, что делать, когда некоторых периодов не хватает.

Существуют два основных метода расчета скользящего временного окна: с помощью самосоединения self-JOIN, которое можно использовать в любой базе данных, и с помощью оконных функций, которые в некоторых базах данных не реализованы. В обоих случаях для вычисления скользящего окна нам нужны: дата и определенное количество точек данных, которые соответствуют размеру окна и на которых мы будем вычислять среднее значение или другую агрегатную функцию.

Поскольку наши данные имеют месячный уровень детализации, то в этом примере мы будем использовать окно в 12 месяцев. Затем мы вычислим среднее значение, чтобы получить 12-месячное скользящее среднее. Для начала давайте подумаем, что войдет в расчет. В этом запросе таблица `retail_sales` с псевдонимом `a` — это наша «якорная» таблица, из которой мы берем даты. Для начала давайте рассмот-

рим только один месяц — декабрь 2019 г. Из таблицы `retail_sales` с псевдонимом `b` выберем данные о продажах за предшествующие 12 месяцев, которые войдут в скользящую среднюю. Это достигается с помощью условия `JOIN b.sales_month between a.sales_month - interval '11 months' and a.sales_month`, который вернет нужное декартово соединение:

```
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
```

Обратите внимание, что результаты в `sales_month` и `sales` одинаковые для каждого месяца временного окна.

Помните, что обе даты, указанные в операторе `BETWEEN`, включаются в результат. Распространенной ошибкой является использование числа 12 вместо 11 при указании диапазона. Если вы сомневаетесь, проверьте промежуточные результаты запроса для одного окна, как сделала я, чтобы убедиться, что в скользящее окно попадает нужное количество записей.

Следующим шагом является применение агрегации — в данном случае функции avg, т. к. нам нужно вычислить скользящее среднее. Я добавлю вывод количества записей, возвращаемых таблицей b, для подтверждения, что усреднение выполняется на 12 точках данных, и это тоже является полезной проверкой качества вычислений. Для таблицы a также добавлен фильтр на sales_month, потому что наш набор данных начинается с 1992 г., и временные окна для точек этого года, за исключением декабря, будут иметь менее 12 записей:

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

sales_month  sales  moving_avg  records_count
-----------  -----  ----------  -------------
1993-01-01   2123   2672.08     12
1993-02-01   2005   2673.25     12
1993-03-01   2442   2676.50     12
...          ...    ...         ...
```

Результаты представлены на рис. 3.13. В то время как график ежемесячных продаж сильно зашумлен, сглаженная кривая скользящего среднего помогает обнаружить такие изменения, как рост продаж с 2003 по 2007 г. и последующее падение до 2011 г. Обратите внимание, что резкое падение продаж в начале 2020 г. привело к снижению скользящего среднего даже с учетом того, что к концу года уровень продаж восстановился.

Добавлять фильтр kind_of_business = 'Women''s clothing stores' для каждой таблицы необязательно. Поскольку в запросе используется INNER JOIN, фильтрация по этому полю для одной таблицы будет автоматически фильтровать по этому же полю другую таблицу. Однако добавление такой фильтрации для обеих таблиц часто ускоряет выполнение запросов, особенно когда таблицы большие.

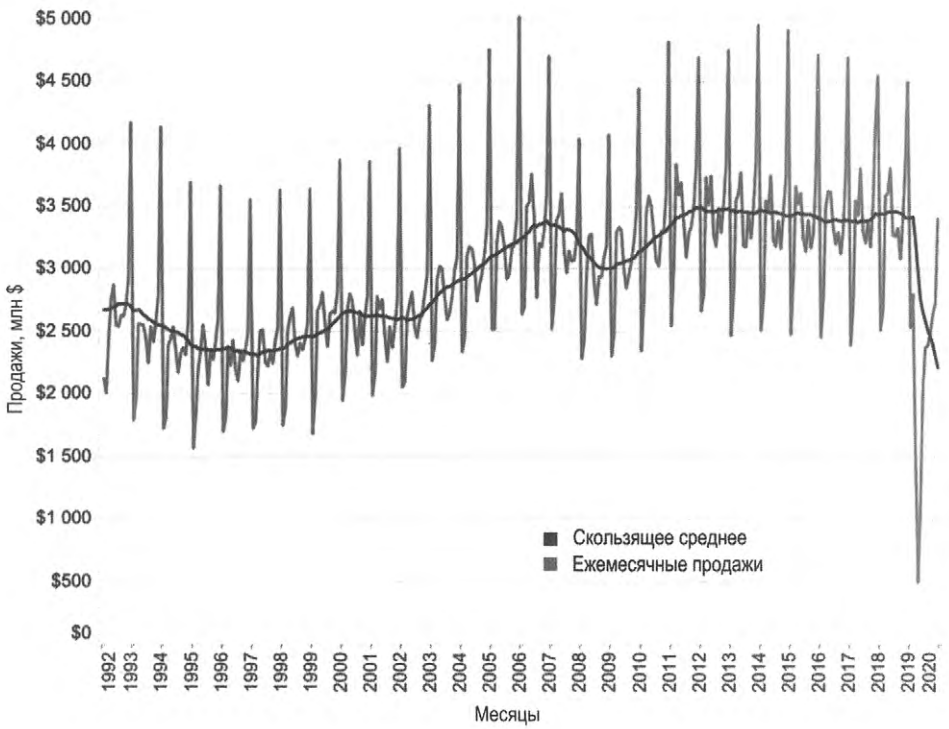


**Рис. 3.13.** Ежемесячные продажи и скользящее среднее за 12 месяцев для магазинов женской одежды

Оконные функции — это еще один способ расчета скользящих временных окон. Чтобы задать скользящее окно в оконных функциях, нужно использовать еще одно необязательное выражение — *рамки окна* (*frame clause*). Это выражение позволяет указать, какие записи следует включить в окно. По умолчанию при вычислении используются все записи в секции, и во многих случаях это работает отлично. Однако более детальное управление записями необходимо для таких случаев, как вычисления со скользящим окном. Синтаксис простой, но при первом знакомстве может сбить с толку. Рамки окна могут быть указаны так:

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

В фигурных скобках приведены три варианта для типа рамки: `RANGE` (диапазон), `ROWS` (строки) и `GROUPS` (группы). С их помощью вы можете указать, какие записи следует включать в результат относительно текущей строки. Записи всегда выбираются из секции `PARTITION BY` и следуют указанному порядку `ORDER BY`. По умолчанию используется сортировка по возрастанию (`ASC`), но ее можно изменить на убывающую (`DESC`). Тип рамки `ROWS` является самым простым и позволяет указать точное количество строк, которые должны быть возвращены. Тип `RANGE` включает в результат записи, которые находятся в пределах некоторой границы значений относительно текущей строки. Тип `GROUPS` можно использовать при наличии нескольких записей с

одним и тем же порядком ORDER BY, например когда набор данных содержит несколько строк о продажах за месяц, по одной для каждого клиента.

Вместо *frame_start* и *frame_end* могут быть использованы следующие выражения:

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

Ключевое слово PRECEDING (предшествующий) означает строки перед текущей строкой в соответствии с сортировкой ORDER BY. Выражение CURRENT ROW (текущая строка) соответствует своему названию. А FOLLOWING (следующее) означает строки, следующие за текущей строкой в соответствии с сортировкой ORDER BY. Ключевое слово UNBOUNDED означает включение всех записей из секции, находящихся соответственно до или после текущей строки. Вместо *offset* нужно указать количество записей, часто это просто целочисленная константа, хотя также может быть поле или выражение, возвращающее целое число. Выражение для рамки окна также может иметь еще один необязательный параметр frame_exclusion, но нам он не понадобится. На рис. 3.14 показан пример строк, которые могут быть выбраны для границ рамки окна.

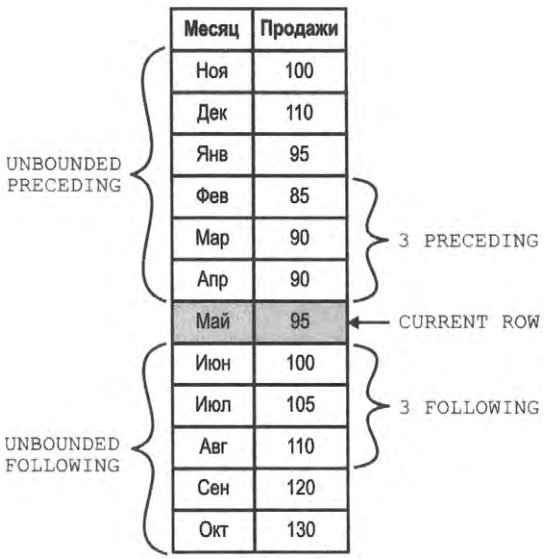


**Рис. 3.14.** Варианты границ рамки окна и строки, которые они включают

Оконные функции имеют множество параметров, с помощью которых можно управлять вычислениями, что делает их невероятно мощными и удобными для выполнения сложных вычислений при относительно простом синтаксисе. Возвращаясь к нашему примеру с розничными продажами, расчет скользящего среднего, ко-

торый мы сделали с помощью self-JOIN, может быть выполнен с помощью оконных функций с меньшим количеством строк кода:

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

В этом запросе в оконных функциях продажи упорядочиваются по месяцам (по возрастанию), чтобы гарантировать, что записи расположены в хронологическом порядке. Рамки окна заданы как `rows between 11 preceding and current row`, поскольку я знаю, что у меня есть запись для каждого месяца, и мне нужно, чтобы 11 предыдущих месяцев и текущий месяц были включены в расчеты среднего значения и общего количества. Запрос возвращает все месяцы, даже те, у которых нет 11 предыдущих записей, и можно отфильтровать их, упаковав этот запрос в подзапрос, а во внешнем запросе отфильтровать результат по месяцу или по количеству записей.

Хотя вычисление скользящих средних по предыдущим периодам широко применяется во многих бизнес-задачах, оконные функции SQL достаточно гибки и позволяют включать в расчеты также и будущие временные периоды. Поэтому эти функции можно использовать для любого набора, в котором данные как-то упорядочены, а не только при анализе временных рядов.

Вычисление скользящих средних или других агрегированных значений может быть легко выполнено с помощью self-JOIN или оконных функций, когда в наборе данных существуют записи для каждого временного периода в скользящем окне. Меж-

ду этими двумя способами могут быть различия в производительности, но это зависит от типа базы данных и размера набора данных. К сожалению, трудно предсказать, какой из способов будет эффективнее, или дать общий совет, какой из них использовать. Стоит попробовать оба варианта и проверить, как долго выполняется ваш запрос; затем выберите тот, который работает быстрее, и применяйте его по умолчанию. Теперь, когда мы разобрались, как работать со скользящими временными окнами, я покажу, как их использовать с разреженными наборами данных.

## Скользящие окна на разреженных данных

Наборы данных в реальном мире не всегда содержат записи для каждого временного периода, попадающего в окно. Интерес к товару может быть сезонным или непостоянным по своей природе. Например, клиенты могут совершать покупки на вебсайте через нерегулярные промежутки времени, или конкретный товар может то поступать, то исчезать со склада. Это приводит к разреженным данным.

В предыдущем разделе я показала, как вычислить скользящее окно с помощью самосоединения и диапазона дат в условии `JOIN`. Можно решить, что такой запрос вернет записи для расчета 12-месячного временного окна, независимо от того, все ли месяцы есть в наборе данных или нет. Но с этим подходом возникает проблема, когда нет «якорной» записи для самой точки (для месяца, дня или года). Представьте, например, что я хочу рассчитать 12-месячные скользящие значения для продаж каждой модели обуви, представленной в моем магазине. Пусть на начало декабря 2019 г. некоторые модели исчезли со склада, поэтому записей об их продажах в этом месяце нет. Использование self-JOIN или оконной функции вернет данные о продажах обуви в декабре, но в результатах будут отсутствовать строки с моделями, которых не было в наличии. К счастью, у нас есть способ решить эту проблему — использовать размерную таблицу дат.

*Размерная таблица дат* — статическая таблица, содержащая по строке на каждую календарную дату и с которой мы уже встречались в *разд. 2.5*. С помощью такой таблицы мы можем гарантировать, что запрос вернет значение для каждой интересующей нас даты независимо от того, есть ли запись на эту дату в исходном наборе данных. Поскольку данные о розничных продажах `retail_sales` содержат строки за все месяцы, я смоделировала разреженный набор данных, добавив подзапрос для фильтрации по `sales_month` и оставив только январь и июль (1-й и 7-й месяцы). Давайте посмотрим на результаты `JOIN` с `date_dim`, но до выполнения агрегации, чтобы проверить данные перед вычислением:

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

Обратите внимание, что, кроме января, запрос возвращает результаты и для февраля и марта, даже если наш подзапрос не содержит записи за эти месяцы. Это возможно, потому что размерная таблица дат `date_dim` содержит записи для всех месяцев. Условие `a.date = a.first_day_of_month` ограничивает вывод только одной строки для каждого месяца. В остальном конструкция этого запроса очень похожа на self-JOIN из предыдущего раздела, в котором использовалось условие `JOIN on b.sales_month between a.date - interval '11 months' and a.date`. Теперь, когда мы понимаем, что вернет запрос, мы можем применить агрегацию `avg` для получения скользящего среднего:

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

Как мы видим, результирующий набор включает строку для каждого месяца. Однако сейчас скользящее среднее остается постоянным до тех пор, пока во временное окно не попадет следующая точка данных (в данном случае следующий январь или июль). Сейчас каждое скользящее среднее состоит только из двух точек данных. В реальной жизни количество исходных точек данных, вероятно, будет варьироваться. Чтобы в результаты еще добавить значение продаж для текущего месяца, можно использовать агрегацию с оператором CASE, например:

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

Условие внутри оператора CASE можно изменить, чтобы вернуть любую из исходных записей, которая требуется для анализа, используя операторы равенства, неравенства или вычисления с датами. Если в вашей базе данных нет размерной таблицы дат, то вместо нее можно использовать другой способ. В подзапросе выберите необходимые даты с помощью DISTINCT и присоедините к ним таблицу с продажами так же, как в предыдущих примерах:

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

В этом примере для подзапроса я использовала ту же исходную таблицу retail_sales, потому что знаю, что она содержит все нужные мне месяцы. Однако на практике можно использовать любую таблицу базы данных, содержащую необходимые даты, независимо от того, связана она с таблицей, для которой вы хотите вычислить скользящее значение, или нет.

Расчет агрегации со скользящими временными окнами на разреженных или отсутствующих данных может быть выполнен в SQL с применением разных соединений. Далее мы рассмотрим, как вычислить накопительные суммы, которые часто используются в анализе.

## Расчет накопительного итога

При расчетах скользящих окон, таких как скользящее среднее, обычно используются окна фиксированного размера, например 12 месяцев, как мы видели в предыдущем разделе. Еще одним часто используемым видом расчетов является *накопительный итог*, например с начала года (year-to-date, YTD), с начала квартала (quarter-to-date, QTD) или с начала месяца (month-to-date, MTD). Вместо окна фиксированного размера при таких расчетах выбирается общая начальная точка временного окна, при этом размер окна будет увеличиваться с каждой следующей строкой.

Самый простой способ вычислить накопительный итог — использовать оконную функцию. В следующем примере функция sum используется для определения общего объема продаж с начала года (YTD) по состоянию на каждый месяц. Для другого анализа вам может потребоваться вычислить ежемесячное среднее или максимальное значение с начала года, которое можно получить, просто заменив функцию sum на avg или max соответственно. Начальная точка временного окна сбрасывается в соответствии с PARTITION BY в нашем случае в начале каждого года продаж. Порядок сортировки ORDER BY обычно содержит поле даты при анализе временных рядов. Отсутствие ORDER BY может привести к неправильным результатам из-за сортировки данных в исходной таблице, поэтому рекомендуется добавить его, даже если вы считаете, что данные уже отсортированы по дате:

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

Для каждого месяца `sales_month` запрос возвращает продажи `sales` за этот месяц и накопительный итог `sales_ytd`. Суммирование начинается в январе 1992 г., а затем сбрасывается в январе 1993 г. и так для каждого года из набора данных. Результаты за период с 2016 по 2020 г. представлены на рис. 3.15. Первые четыре года имеют схожие тенденции на протяжении года, но 2020 год, конечно, выглядит совсем иначе.

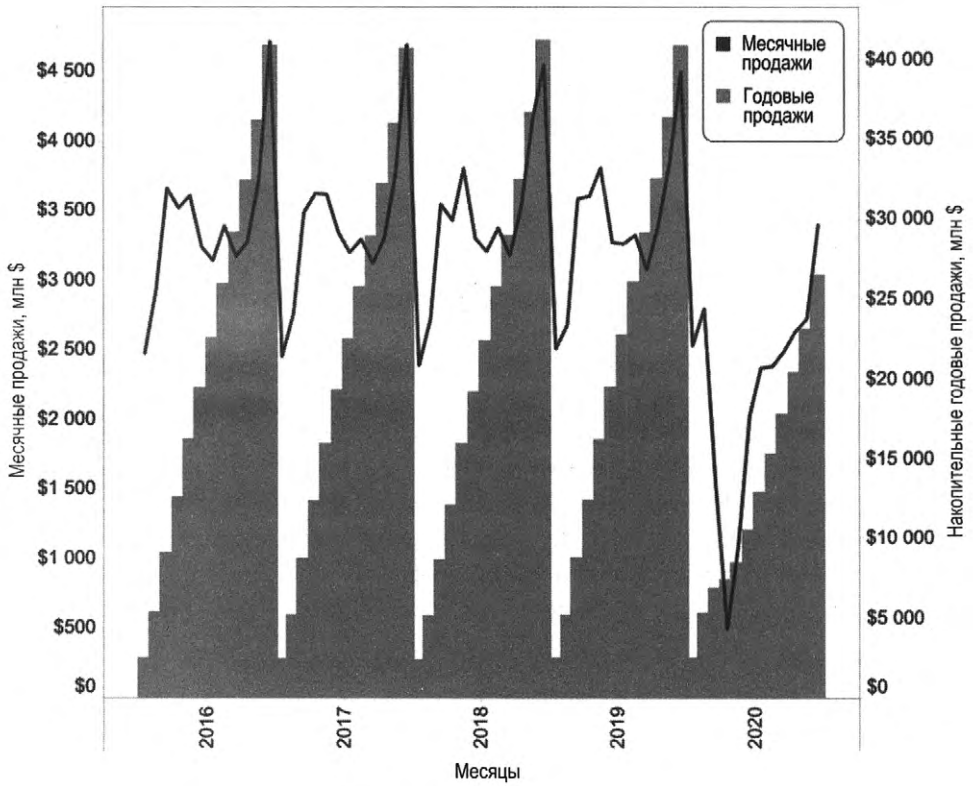


**Рис. 3.15.** Ежемесячные продажи и накопительные годовые продажи в магазинах женской одежды

Эти же результаты можно получить без оконных функций, используя self-JOIN, который вернет декартово соединение. В примере ниже две таблицы с псевдонимами соединяются с помощью `JOIN` по году, чтобы гарантировать, что агрегирование выполняется для одного и того же года, сбрасываясь каждый год. В условии опера-

тора JOIN также указано, что b.sales_months должно быть меньше или равно a.sales_month. В январе 1992 г. этому условию будет соответствовать только строка за январь 1992 г. из таблицы b; в феврале 1992 г. — и за январь, и за февраль и т. д. Запрос будет выглядеть так:

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

Для оконных функций приходится писать меньше кода, и, если вы разберетесь с их синтаксисом, то легко понять, что именно они вычисляют. Как правило, в SQL существует несколько способов решения задач, и скользящие временные окна являются хорошим примером. Я считаю, что полезно знать разные способы, потому что время от времени я сталкиваюсь со сложной проблемой, которая лучше решается с помощью подхода, менее эффективного в обычных случаях. Теперь, когда мы рассмотрели скользящие временные окна, перейдем к нашей последней теме в анализе временных рядов — сезонности.

## 3.5. Анализ с учетом сезонности

*Сезонность* — это любая закономерность, которая повторяется через определенные промежутки времени. В отличие от разных шумов в данных, сезонность можно предсказать. Само слово «сезонность» напоминает о четырех временах года — весне, лете, осени и зиме — и в некоторых наборах данных можно найти эту закономерность. Поведенческие шаблоны покупателей меняются в зависимости от време-

ни года, начиная с одежды и продуктов питания и заканчивая деньгами, потраченными на отдых и путешествия. Сезон зимних праздничных покупок может быть решающим для многих розничных магазинов. Но сезонность может быть и в других масштабах времени — в годах или минутах. Президентские выборы в Соединенных Штатах Америки проводятся каждые четыре года, что приводит к четкой сезонности их освещения в СМИ. Цикличность дней недели является обычным явлением, т. е. работа и учеба преобладают с понедельника по пятницу, а домашние дела и досуг приходятся на выходные дни. Время суток — это еще один вид сезонности, с которым сталкиваются, например, рестораны: наплыв посетителей во время обеда и ужина и более низкий уровень продаж между ними.

Чтобы узнать, существует ли сезонность во временном ряду и в каком масштабе, лучше отобразить временной ряд в виде графика, а затем визуально оценить закономерности. Попробуйте агрегировать данные на разных уровнях: по часам, дням, неделям или месяцам. Вы также должны воспользоваться сведениями о самом наборе данных. Можно ли догадаться о существовании закономерностей, опираясь на то, что вы знаете о сущностях или процессах, которые они представляют? Проконсультируйтесь со специалистами в предметной области, если у вас есть такая возможность.

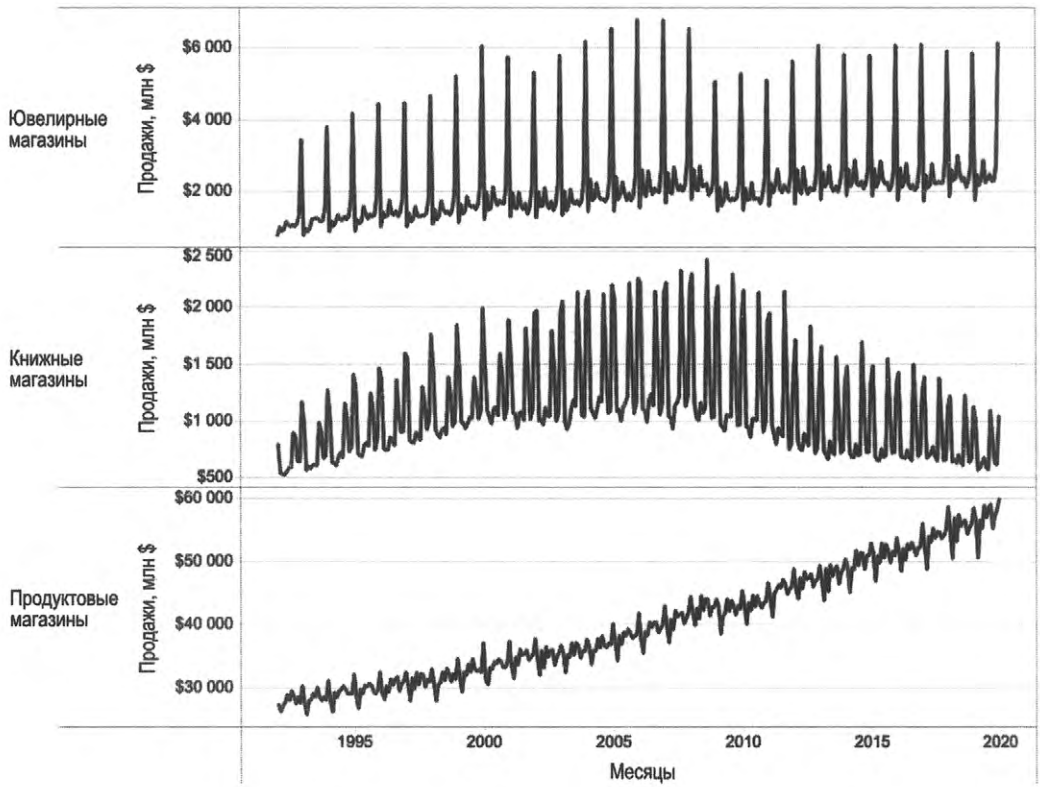


**Рис. 3.16.** Примеры сезонности продаж в розничных магазинах

Давайте взглянем на некоторые сезонные закономерности в наборе данных о розничных продажах, показанных на рис. 3.16. Для ювелирных магазинов характерна ярко выраженная сезонность с ежегодными пиками в декабре, связанными с новогодними подарками. Каждый год в книжных магазинах бывает два пика продаж: один пик приходится на август, что соответствует началу учебного года в США; второй пик начинается в декабре и продолжается в январе, включая период новогодних подарков и возвращение к учебе в новом семестре. Третий пример — это продуктовые магазины, в которых сезонность по месяцам гораздо менее выражена, чем для двух других типов магазинов (хотя продуктовые магазины, вероятно, имеют сезонность на уровне дня недели и времени суток). В этом нет ничего удивительного: людям нужно питаться круглый год. Продажи в продуктовых магазинах немного увеличиваются в декабре из-за праздников и снижаются в феврале, поскольку в этом месяце просто меньше всего дней.

Сезонность бывает разных видов, но, несмотря на это, есть общие подходы к ее анализу. Один из способов справиться с сезонностью — сгладить ее, используя агрегирование или скользящие окна, как мы видели ранее. Еще один способ обработки сезонных данных — сравнение аналогичных временных периодов и анализ различий. Далее я покажу несколько способов, как это сделать.

## Сравнение периодов: YoY и MoM

Сравнение временных периодов между собой можно выполнять разными способами. Первый заключается в сравнении временного периода с предыдущим в ряду — это настолько распространенная практика в анализе, что существуют обозначения для самых часто используемых сравнений. В зависимости от уровня агрегирования сравнение может выполняться по годам (год к году, year-over-year, YoY), по месяцам (месяц к месяцу, month-over-month, MoM), по дням (день к дню, day-over-day, DoD) и т. д.

Для этих вычислений мы будем использовать функцию `lag` — еще одну оконную функцию. Функция `lag` возвращает предыдущее значение из ряда и имеет следующие аргументы:

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

Здесь *return_value* — это любое поле из набора данных, может быть любого типа. Необязательный параметр *offset* указывает, на сколько строк назад в пределах секции нужно сместиться, чтобы взять значение *return_value*. По умолчанию этот параметр равен 1, но можно использовать любое целочисленное значение. Вы также можете указать значение по умолчанию *default*, которое будет использоваться, если нет записи, из которой можно взять значение. Как и другие оконные функции, `lag` также вычисляется в пределах секции `PARTITION BY` с сортировкой `ORDER BY`. Если `PARTITION BY` не указан, `lag` работает со всем набором данных, и аналогично, если не указан `ORDER BY`, используется сортировка базы данных. Но, тем не менее, считается правильным добавлять хотя бы предложение `ORDER BY` для управления выводом.

Оконная функция `lead` работает так же, как функция `lag`, за исключением того, что она возвращает последующее значение, определяемое смещением *offset*. Изменение порядка сортировки в `ORDER BY` с возрастающего (`ASC`) на убывающий (`DESC`) приводит к тому, что оконная функция `lag` становится эквивалентной функции `lead`. Кроме того, в функции `lag` в качестве параметра *offset* можно указать отрицательное целое число для возврата значений из последующих строк.

Давайте применим это к нашему набору данных о розничных продажах, чтобы рассчитать рост в месячном и годовом исчислении (MoM и YoY). В этом разделе мы сосредоточимся на продажах в книжных магазинах, т. к. я настоящий книжный червь. Для начала давайте проверим, что возвращает функция `lag`, применив ее к месяцу и объему продаж:

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

Для каждой строки возвращается предыдущий месяц `sales_month`, а также продажи `sales` за тот месяц, и мы можем убедиться в этом, проверив первые несколько строк результирующего набора. В первой строке значения `prev_month` и `prev_month_sales` равны `null`, поскольку для этой строки нет более ранней записи. Имея представление о значениях, возвращаемых функцией `lag`, мы можем вычислить процентное изменение по сравнению с предыдущим месяцем:

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

Продажи упали на 31,77% с января по февраль, по крайней мере, частично из-за сезонного спада после каникул и возвращения к учебе на новый семестр. С февраля по март продажи снизились всего на 0,74%.

Сравнение в годовом исчислении YoY рассчитывается аналогично, но сначала нужно агрегировать продажи за год. Поскольку мы рассматриваем только один вид розничных магазинов, я уберу поле `kind_of_business` из результатов:

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

Продажи выросли на 9.37% с 1992 по 1993 г. и почти на 11% с 1993 по 1994 г. Эти сравнения по периодам полезны, но они не очень помогают нам с анализом сезонности в наборе данных. Например, на рис. 3.17 представлен процентный рост в месячном исчислении MoM, и он содержит столько же сезонных колебаний, сколько и исходный временной ряд.

Чтобы решить эту проблему, давайте рассмотрим в следующем разделе, как использовать SQL для сравнения текущих значений со значениями за этот же месяц в предыдущем году.

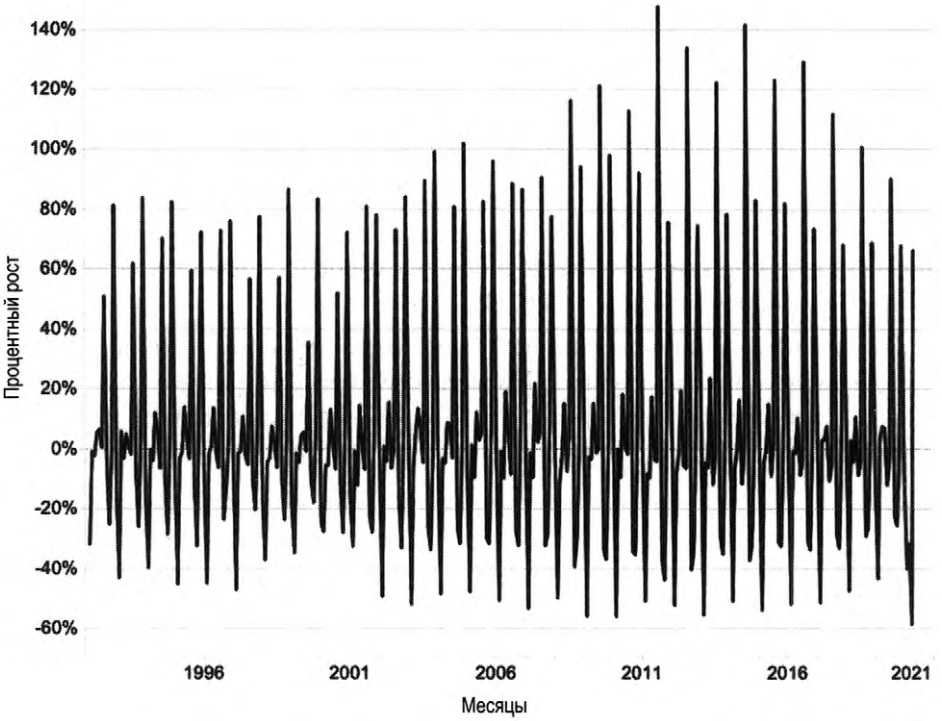


**Рис. 3.17.** Процентный рост продаж в месячном исчислении MoM в книжных магазинах

## Сравнение периодов: этот же месяц в прошлом году

Сравнение данных за какой-то месяц с данными за аналогичный месяц в предыдущем году может быть удобным способом контроля сезонности. Этот же способ можно применить для сравнения дня недели с таким же днем на предыдущей неделе или квартала с аналогичным кварталом в прошлом году, или любого другого временного периода, который имеет смысл для вашего набора данных.

Чтобы выполнить такое сравнение, мы можем использовать функцию `lag` вместе с хитрым `PARTITION BY`, указав в этом предложении единицу времени, с которой мы хотим сравнить текущее значение. В данном случае мы будем сравнивать месячные продажи с продажами за такой же месяц в предыдущем году. Например, продажи за январь будут сравниваться с продажами за январь предыдущего года, продажи за февраль — с продажами за февраль предыдущего года и т. д.

Сначала давайте вспомним, что функция `date_part` возвращает число при использовании с аргументом `'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
...          ...
```

Мы будем использовать `date_part` в предложении `PARTITION BY`, чтобы оконная функция `lag` искала месяц с таким же номером из предыдущего года.

Это пример того, как в оконных функциях можно использовать вычисления, а не только поля базы данных, что делает эти функции еще более универсальными. Так как я считаю полезным проверять промежуточные результаты для оттачивания навыков, то сначала мы убедимся, что функция `lag` с `partition by date_part('month', sales_month)` возвращает предполагаемые значения:

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

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

Первая функция `lag` возвращает тот же месяц за предыдущий год — значение `prev_year_month`. В строке результатов с `sales_month`, равным `1993-01-01`, значение

prev_year_month равно 1992-01-01, как и должно быть, а prev_year_sales, равное 790, соответствует продажам sales за 1992-01-01. Обратите внимание, что prev_year_month и prev_year_sales равны null для 1992 года, поскольку в наборе данных нет записей за предыдущий год.

Теперь, когда мы уверены, что функция lag возвращает правильные значения, мы можем рассчитать показатели сравнения, такие как разница и процентное изменение:

```
SELECT sales_month, sales
,sales - lag(sales) over (partition by date_part('month',sales_month)
                          order by sales_month
                          ) as difference
,(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  difference     pct_diff
-----------  -----  -------------  --------
1992-01-01   790    (null)          (null)
1993-01-01   998    208             26.32
1994-01-01   1053   55             5.51
...          ...    ...            ...
```

Теперь мы можем представить результаты графически (рис. 3.18). Легко увидеть месяцы, когда был необычно высокий рост, например январь 2002 г., или необычно высокая убыль, например декабрь 2011 г.

Еще одним полезным инструментом анализа является сводный график, на котором общие названия временных периодов (в данном случае месяцев) отложены по горизонтальной оси и построены линейные графики для каждого временного ряда (в данном случае для каждого года), один под другим.

Для этого мы напишем запрос, который возвращает по строке для каждого номера или названия месяца и по столбцу для каждого года, который мы хотим отобразить. Чтобы получить месяц, мы можем использовать функцию date_part или функцию to_char, в зависимости от того, какое значение нам нужно для обозначения месяца — числовое или текстовое. Затем мы выполним сведение данных по годам, используя агрегатную функцию.

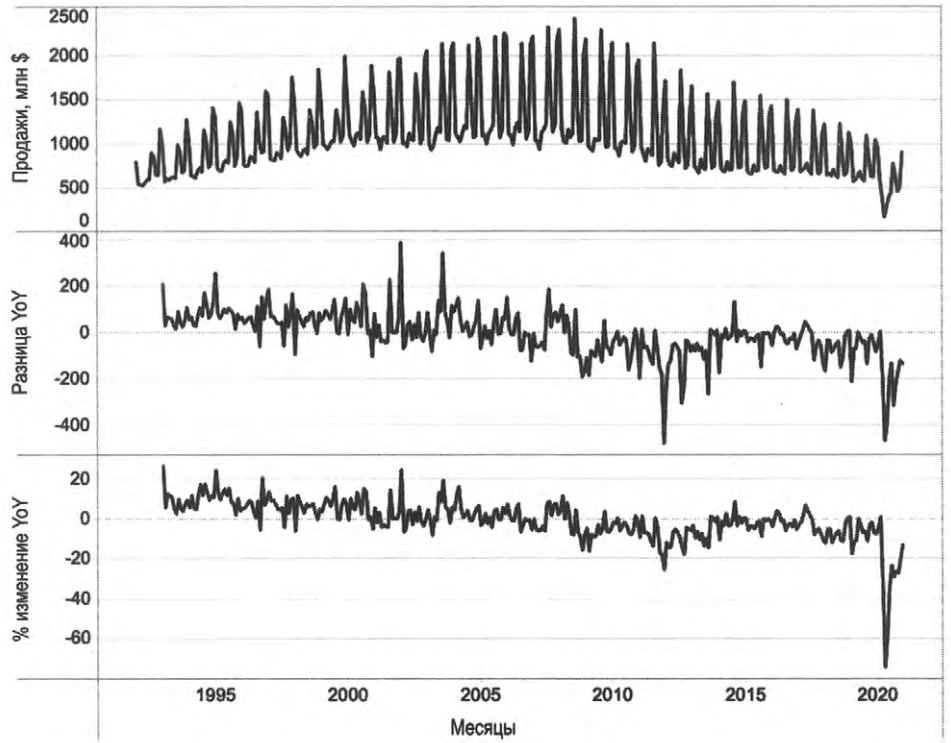


**Рис. 3.18.** Розничные продажи в книжных магазинах: разница YoY и процентное изменение YoY

В этом примере используется функция `max`, но, в зависимости от анализа, это может быть `sum`, `count` или любое другое агрегирование. Мы возьмем данные только с 1992 по 1994 г.:

```
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 |

Выстраивая данные таким образом, мы можем сразу увидеть некоторые тенденции. Продажи за декабрь являются самыми высокими месячными продажами в каждом году. Продажи в каждом месяце 1994 г. были выше, чем продажи в 1992 и 1993 гг. Всплеск продаж в августе и сентябре заметен, но сильнее выражен в 1994 г.

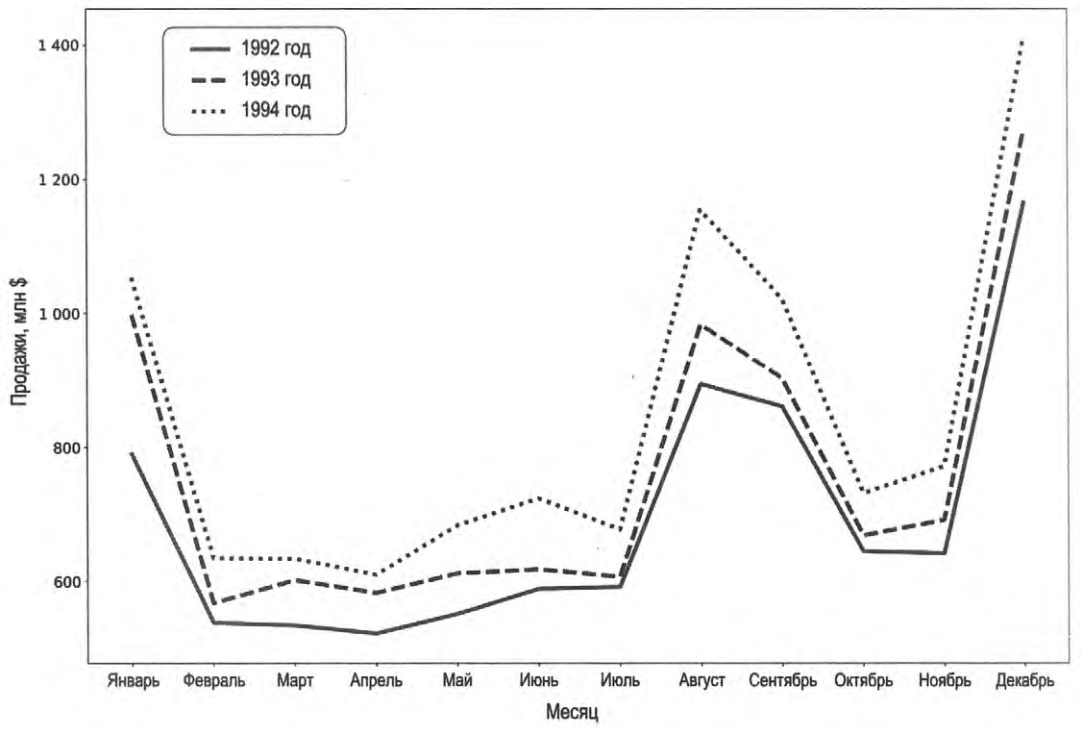


**Рис. 3.19.** Розничные продажи в книжных магазинах за 1992–1994 гг., выстроенные по месяцам

По графику данных (рис. 3.19) тренды определить намного проще. Продажи увеличивались из года в год для каждого месяца, хотя в некоторые месяцы рост был больше, чем в другие. Имея на руках эти данные и график, мы можем приступить к созданию отчета о продажах в книжных магазинах, который пригодится при планировании инвентаризации или при составлении расписания маркетинговых акций, или станет частью более широкой отчетности о розничных продажах в США.

В SQL есть ряд способов, позволяющих избавиться от сезонного шума при сравнении данных во временных рядах. В этом разделе мы разобрали, как сравнивать текущие значения с предыдущими сопоставимыми периодами, используя оконную функцию `lag`, и как сводить данные с помощью `date_part`, `to_char` и агрегирования. Далее я расскажу, как сравнивать несколько предыдущих периодов, чтобы дополнительно контролировать зашумленные данные временных рядов.

## Сравнение с несколькими предыдущими периодами

Сравнение данных с предыдущими сопоставимыми периодами является хорошим способом уменьшить шум, возникающий из-за сезонности. Иногда сравнения с одним предыдущим периодом недостаточно, особенно если на этот период повлияли необычные события. Сравнивать понедельник с предыдущим понедельником сложно, если один из них был нерабочим днем. Месяц в предыдущем году мог быть необычным из-за экономических событий, неблагоприятных погодных условий или отключения сайта, что изменило привычное поведение покупателей. Сравнение текущих значений со средним значением за несколько предыдущих периодов может сгладить эти отклонения. При этом мы будем применять те же возможности SQL, которые уже использовали при расчете скользящих временных окон и сравнении результатов за предыдущий период.

Для начала воспользуемся функцией `lag`, как и в предыдущем разделе, но здесь мы применим необязательный параметр *offset*. Напомним, что если смещение не указано, функция возвращает непосредственно предшествующее значение в соответствии с `PARTITION BY` и `ORDER BY`. Если смещение равно 2, то возвращается значение из 2-й предшествующей строки, если равно 3, — из 3-й предшествующей строки и т. д.

В этом примере мы сравним продажи текущего месяца с продажами такого же месяца за три предыдущих года. Как обычно, сначала проверим возвращаемые значения, чтобы убедиться, что SQL работает должным образом:

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

Значение `null` возвращается тогда, когда нет запрашиваемой предшествующей записи. Остальные значения отображаются правильно и соответствуют нужному месяцу в предыдущих годах. Теперь мы можем вычислить любой сравнительный показатель, который требуется в анализе, например процент от скользящего среднего за три предыдущих периода:

```
SELECT sales_month, sales
,sales * 100 / ((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
...          ...    ...
```

Из результатов видно, что в середине 1990-х гг. продажи книг росли по сравнению со скользящим средним за три предыдущие года, но картина изменилась в конце 2010-х, когда каждый год продажи сокращались в процентном соотношении от скользящего среднего за три года.

Вы могли заметить, что этот пример напоминает тот, который мы видели ранее при расчете скользящих временных окон. В качестве альтернативы предыдущему запросу мы можем использовать оконную функцию `avg` с заданными рамками окна. В `PARTITION BY` будем использовать все ту же функцию `date_part`, и `ORDER BY` будет тот же. Рамки окна добавлены как `rows between 3 preceding and 1 preceding` для включения в расчеты только трех предыдущих строк, исключая текущую строку:

```
SELECT sales_month, sales
,sales * 100 /
 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
...          ...    ...
```

Результаты совпадают с результатами предыдущего запроса, подтверждая эквивалентность кода.

Если вы посмотрите внимательнее, то заметите, что значения десятичных разрядов в результатах запроса с `lag` немного отличаются от результатов запроса с оконной функцией `avg`. Это связано с тем, как база данных обрабатывает десятичное округление в промежуточных вычислениях. Для многих анализов эта разница не будет иметь значения, но будьте внимательны, если вы работаете с финансовыми или другими строго контролируемыми данными.

Анализ данных с учетом сезонности часто включает в себя попытки уменьшить шум, чтобы сделать четкие выводы об основных трендах. Сравнение точек данных с несколькими предыдущими временными периодами позволяет получить более плавную динамику для оценки того, что на самом деле происходит в текущий период. Необходимо, чтобы данные включали достаточно информации для проведения таких сравнений, а когда у вас есть довольно длинные временные ряды, то это может привести к очень ценным выводам.

## 3.6. Заключение

Анализ временных рядов — это очень мощный способ анализа наборов данных. Мы увидели, как подготовить наши данные к анализу с помощью операций над датами и временными метками. Мы поговорили о размерной таблице дат и увидели, как ее применять для расчета скользящих временных окон. Мы рассмотрели сравнение периодов и способы анализа данных с учетом сезонности. В следующей главе мы углубимся в смежную тему, связанную с анализом временных рядов, — когортный анализ.

# ГЛАВА 4
# Когортный анализ

В *гл. 3* мы рассмотрели анализ временных рядов. Зная его техники, мы можем теперь обратиться к смежному типу анализа, используемого во многих сферах бизнеса, — когортному анализу.

Я помню, как впервые познакомилась с когортным анализом. Моим первым местом работы в качестве аналитика данных был небольшой стартап. Я просматривала результаты анализа продаж, который делала для генерального директора, и он предложил мне разбить клиентскую базу на группы (когорты), чтобы увидеть, как меняется их поведение со временем. Я решила, что это какая-то модная идея из его бизнес-школы и, скорее всего, бесполезная, но так как она исходила от генерального директора, то, конечно, мне пришлось этим заняться. Как оказалось, это была не просто причуда. Разбивка аудитории на когорты и отслеживание их во времени — это эффективный способ анализа ваших данных и предотвращения различных ошибок. Когортный анализ может дать представление о том, чем группы людей отличаются друг от друга и как они меняются с течением времени.

В этой главе мы сначала разберем, что такое когорты и что входит в некоторые типы когортного анализа. После беглого знакомства с набором данных о законодателях, который мы будем использовать в примерах, мы узнаем, как сделать анализ удержания и решать разные задачи, такие как формирование когорт и обработка разреженных данных. Далее мы рассмотрим, как вычислять выживаемость, возвращаемость и накопительный итог, для которых структура кода SQL аналогична анализу удержания. Наконец, мы рассмотрим, как сочетать когортный анализ с анализом поперечных срезов, чтобы изучить состав аудитории с течением времени.

## 4.1. Составляющие когортного анализа

Прежде чем перейти к коду, я расскажу, что такое когорты, на какие вопросы можно ответить с помощью этого типа анализа, и из чего состоит любой когортный анализ.

*Когорта* — это группа лиц, которые обладают некоторыми общими характеристиками в то время, когда мы начинаем за ними наблюдение. В когорты чаще всего объединяют людей, но это могут быть любые сущности, которые мы хотим изучить: компании, товары или явления физического мира. Каждый человек в когорте

может знать о своей принадлежности к этой когорте, точно так же, как дети в первом классе знают, что они относятся к группе первоклассников, или участники испытаний лекарственных препаратов знают, что они являются частью группы, получающей лечение. В других случаях сущности группируются в когорты по каким-то характеристикам, которые нам интересны для целей анализа, например когда компания, разрабатывающая программное обеспечение, объединяет всех клиентов, пришедших в определенном году, чтобы проанализировать, как долго они остаются ее клиентами. Всегда важно следовать этическим принципам группировки людей без их ведома, если это может как-то их задеть.

*Когортный анализ* — это полезный способ сравнить группы сущностей во времени. Для возникновения или формирования многих важных шаблонов поведения требуются недели, месяцы или даже годы, и когортный анализ позволяет увидеть эти изменения. Когортный анализ позволяет обнаружить корреляцию между характеристиками когорт и долгосрочными тенденциями, что может привести к появлению гипотез о причинно-следственных связях. Например, клиенты, привлеченные в рамках маркетинговой кампании, могут иметь другие шаблоны поведения в долгосрочном периоде, чем те, кого «привел друг». Когортный анализ можно использовать для мониторинга новых когорт клиентов и оценки того, как они ведут себя по сравнению со старыми когортами. Такой мониторинг может заранее предупредить, если что-то пошло не так с новыми клиентами (или наоборот — убедиться, что все идет хорошо). Когортный анализ также используется для сбора исторических данных. Анализ экспериментов, которому посвящена *гл. 7*, является основным способом выявления причинно-следственных связей, но мы не можем вернуться в прошлое и провести эксперимент по каждому интересующему нас вопросу задним числом. Конечно, следует с осторожностью делать выводы о причинно-следственных связях по результатам когортного анализа, а вместо этого лучше использовать его, чтобы понять клиентов и выдвинуть гипотезы, которые можно будет тщательно проверить в будущем.

Когортный анализ состоит из трех составляющих: группировка по когортам, временной ряд данных, который представляет собой наблюдения за когортами, и совокупный показатель, который измеряет действия, выполненные участниками когорты.

*Группировка по когортам* часто опирается на начальную дату: дата первой покупки или первой подписки, дата начала обучения и т. д. Однако когорты также могут быть сформированы по другим характеристикам, которые могут быть неизменными или меняться со временем. К неизменным характеристикам относятся год рождения, страна происхождения или год основания компании. Характеристика, которая может меняться со временем, — это, например, город проживания или семейное положение. Когда используются последние, надо быть очень аккуратными или формировать когорты только по начальной дате, иначе сущности могут переходить из одной когорты в другую.

### *Когорта или сегмент?*

Эти два термина часто используют схожим образом или даже взаимозаменяют, но для ясности давайте разберемся, в чем между ними различие. *Когорта* — это группа клиентов (или других сущностей), у которых есть общая начальная дата и которые отслеживаются с течением времени. *Сегмент* — это группа клиентов, которые имеют общую характеристику или набор характеристик в определенный момент времени, независимо от их начальной даты. Как и когорты, сегменты могут создаваться на основе неизменных факторов, таких как возраст или поведенческие признаки. Сегмент пользователей, которые зарегистрировались в этом месяце, можно объединить в когорту и отслеживать с течением времени. Или вы можете изучить различные группы пользователей с помощью когортного анализа, чтобы самим оценить, какие из них наиболее ценны. Анализы, которые мы рассмотрим в этой главе, например анализ удержания, помогают собирать полезные сведения о маркетинговых сегментах.

Второй составляющей любого когортного анализа являются *временные ряды*. Это может быть серия покупок, входов в систему, обращений или других действий, выполняемых клиентами или сущностями, которые можно выделить в когорты. Важно, чтобы временной ряд охватывал весь жизненный цикл сущностей, иначе на более ранних когортах можно будет получить систематическую *ошибку выжившего*. Ошибка выжившего возникает, когда в рассматриваемый набор данных включают только оставшихся клиентов, а ушедших клиентов исключают, потому что они больше не доступны. Эти оставшиеся клиенты кажутся более качественными или лояльными по сравнению с новыми когортами (см. врезку «Ошибка выжившего» в *разд. 4.5*). Также важно иметь достаточно длинный временной ряд, чтобы сущности успели выполнить интересующее вас действие. Например, если клиенты обычно делают покупки раз в месяц, то необходим временной ряд за несколько месяцев. А если покупки совершаются только один раз в год, предпочтительнее взять временной ряд за несколько лет. Естественно, что у недавно пришедших клиентов будет намного меньше времени для выполнения действий, чем у клиентов, которые пришли давно. Поэтому для нормализации в когортном анализе обычно измеряется количество временных периодов, прошедших с начальной даты, а не количество календарных месяцев. Таким образом, когорты можно сравнивать в первом периоде, во втором периоде и т. д., чтобы увидеть, как они развиваются с течением времени, независимо от того, в каком месяце происходили действия фактически. В качестве временных периодов могут выступать дни, недели, месяцы или годы.

*Совокупный показатель* должен быть связан с действиями, которые важны для деятельности организации, например с повторными покупками или подписками. Для каждой когорты вычисляется агрегированное значение показателя, обычно с помощью функций `sum`, `count` или `avg`, хотя можно использовать любую подходящую агрегацию. Результатом будет новый временной ряд, который затем можно использовать для оценки изменений в поведении когорт с течением времени.

В этой главе я расскажу о четырех видах когортного анализа: удержание, выживаемость, возвращаемость и накопительный итог.

*Удержание*

Удержание связано с тем, есть ли у участников когорты еще записи во временном ряду через конкретное количество периодов, прошедших с начальной даты. Этот показатель полезно знать для организаций любого типа, в которых ожидают от клиентов повторных действий, будь то повторный вход в онлайн-игру, повторная покупка товара или продление подписки. Удержание помогает ответить на вопросы о том, насколько привлекателен товар для покупателей и сколько повторных действий можно ожидать в будущем.

*Выживаемость*

Выживаемость связана с тем, сколько сущностей оставалось в наборе данных на протяжении определенного периода времени или дольше, независимо от количества или частоты действий. Выживаемость помогает ответить на вопросы о доле населения, которая ожидаемо останется — либо в положительном ключе (сколько осталось после естественной убыли), либо в отрицательном ключе (сколько человек не закончило обучение или не выполнило каких-то требований).

*Возвращаемость*

Возвращаемость или поведение при повторной покупке связана с тем, произошло ли действие больше определенного количества раз (минимального порога) — чаще всего просто более одного раза — в течение фиксированного временного окна. Этот вид анализа полезен в ситуациях, когда поведение носит непостоянный и непредсказуемый характер, например в розничной торговле, где вычисляется доля повторных покупателей в каждой когорте в течение фиксированного временного окна.

*Накопительный итог*

Накопительный итог связан с общим количеством или общей суммой, вычисляемыми в пределах одного или нескольких фиксированных временных окон, независимо от того, когда именно в этом окне произошли действия. Накопительные расчеты часто используются при вычислении пожизненной ценности клиента (LTV или CLTV).

Четыре вида когортного анализа позволяют нам сравнивать когорты и оценивать, как они различаются с течением времени, чтобы принимать более эффективные решения в отношении товаров, маркетинга и финансов. Расчеты для этих видов анализа аналогичны, поэтому мы начнем с вычисления удержания, а затем я покажу, как этот код изменить для расчета остальных показателей. Прежде чем мы приступим к выполнению нашего когортного анализа, давайте взглянем на набор данных, который мы будем использовать для примеров в этой главе.

## 4.2. Набор данных о законодателях

В примерах этой главы мы будем использовать набор данных о бывших и нынешних членах Конгресса США, хранящийся в репозитории GitHub[1]. Конгресс США

[1] **https://github.com/unitedstates/congress-legislators**

отвечает за написание законов и законодательную деятельность, поэтому его членов также называют законодателями. Поскольку этот набор данных представляет собой файл JSON, я сделала некоторые преобразования, чтобы модель данных более подходила для анализа, и выложила данные в формате CSV, удобном для использования, вместе с примерами в GitHub для этой книги[2].

В исходном репозитории есть отличный словарь данных[3], поэтому я не буду приводить его здесь. Однако я поясню некоторые детали для тех, кто не знаком со структурой правительства США, чтобы они могли разобраться в примерах этой главы.

Конгресс состоит из двух палат: Сената (Senate, `'sen'` в наборе данных) и Палаты представителей (House of Representatives, `'rep'`). Каждый штат представляют два сенатора, которые избираются сроком на 6 лет. Представители распределяются по штатам пропорционально численности населения, по одному от каждого округа. Представители избираются сроком на 2 года. Фактические сроки полномочий в любой из палат могут быть короче в случае смерти законодателя, избрания или назначения его на более высокую должность. Чем дольше законодатели находятся на своем посту, тем больше у них власти и влияния, поэтому баллотироваться на переизбрание — обычное дело. Наконец, законодатель может принадлежать к политической партии или быть «независимым». В современную эпоху подавляющее большинство законодателей являются демократами или республиканцами, и соперничество этих двух партий хорошо известно. Законодатели время от времени меняют партию, оставаясь у власти.

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

**Рис. 4.1.** Фрагмент таблицы `legislators`

[2] **https://github.com/cathytanimura/sql_book/tree/master/Chapter 4: Cohorts**

[3] **https://github.com/unitedstates/congress-legislators#data-dictionary**

Для анализа мы будем использовать две таблицы: `legislators` и `legislators_terms`. Таблица `legislators` содержит список всех людей, включенных в набор данных, с указанием дня рождения, пола и разных идентификаторов, по которым можно найти этого человека в других наборах данных. Таблица `legislators_terms` содержит записи о каждом сроке полномочий каждого законодателя с указанием даты начала и окончания, а также палаты и партии. Поле `id_bioguide` используется как уникальный идентификатор законодателя и присутствует в обеих таблицах. На рис. 4.1 показан пример данных из таблицы `legislators`. На рис. 4.2 показан пример данных из `legislators_terms`.

| * | 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 |

**Рис. 4.2.** Фрагмент таблицы `legislators_terms`

Теперь, когда у нас есть представление о том, что такое когортный анализ и какой набор данных мы будем использовать в примерах, давайте приступим к написанию SQL для анализа удержания. Ключевой вопрос, на который SQL поможет нам ответить: после того, как представители попадают в Конгресс, как долго они сохраняют свои места?

## 4.3. Анализ удержания

Одним из наиболее распространенных видов когортного анализа является *анализ удержания*. Под «удержанием» понимается сохранение или продолжение чего-либо. Компании обычно хотят, чтобы их клиенты продолжали покупать их товары или продолжали пользоваться их услугами, поскольку удержание клиентов более выгодно, чем привлечение новых. Работодатели хотят удержать своих сотрудников, потому что поиск замены затратен и требует много времени. Избранные должност-

ные лица стремятся к переизбранию, чтобы продолжить работу во благо своих избирателей.

Главный вопрос в анализе удержания заключается в том, останется ли начальный размер когорты — количество пользователей или сотрудников, потраченная сумма или другой ключевой показатель — постоянным, уменьшится или увеличится с течением времени. Когда изменяется размер когорты, разница и скорость изменений тоже представляют интерес. В большинстве случаев начальный размер когорты со временем имеет тенденцию к уменьшению, поскольку когорта может терять участников, но не приобретает новых после того, как она уже сформирована. Интересным исключением является доход, поскольку когорта покупателей в более поздний месяц может потратить больше, чем в первый месяц, даже если некоторые из покупателей покинули когорту

В анализе удержания выполняется подсчет (`count`) сущностей или суммирование (`sum`) денежных величин или действий из набора данных для каждого временного периода с начальной даты, а затем эти числа нормализуются путем деления на значение, полученное для первого временного периода. Результат выражается в процентах, и удержание в первом периоде всегда равно 100%. Со временем удержание, основанное на подсчете количества, обычно снижается и никогда не превышает 100%, тогда как удержание на основе денег или действий хотя часто и снижается, но может расти и превысить 100% в какой-то период. Результаты анализа удержания обычно выводятся в виде таблицы или графика, который называется *кривой удержания*. Далее в этом разделе мы построим их для нескольких примеров.

Графики с кривыми удержания можно использовать для сравнения когорт. Первое, на что следует обратить внимание, — это форма кривой в первые несколько периодов, когда часто наблюдается начальный резкий спад. Для многих пользовательских приложений потеря половины когорты в первые несколько месяцев является обычным явлением. Когорта, у которой кривая удержания более или менее крутая, чем у других когорт, может указывать на изменения в товаре или в источнике привлечения клиентов для этой когорты, что требует дополнительного изучения. Второе, на что следует обратить внимание, — это становится ли кривая удержания плоской через какое-то количество периодов или продолжает быстро снижаться до нуля. Сглаженная кривая указывает на то, что существует момент времени, начиная с которого большая часть оставшейся когорты сохранится на неограниченный срок. Кривая удержания, которая со временем начинает идти вверх и иногда называется *кривой улыбки*, может получиться, когда участники когорты возвращаются или повторно активируются после выпадения из набора данных на некоторое время. Наконец, кривые удержания, которые строятся по доходу от подписок, проверяются на предмет увеличения дохода на одного клиента с течением времени, что является признаком успешного бизнеса SaaS.

В этом разделе я расскажу, как выполнить анализ удержания, сгруппировать когорты по временному ряду или по другим таблицам, а также как работать с отсутствующими и разреженными данными временных рядов. Имея на руках этот базис, в *разд. 4.4* вы узнаете, как изменить код для выполнения связанных видов когорт-

ного анализа. В результате раздел об удержании будет самым длинным в этой главе, т. к. в нем разбираются SQL-код и все вычисления.

## Общая кривая удержания

Для анализа удержания, как для любого когортного анализа, нам нужны три составляющих: когорты, временной ряд с данными о действиях и совокупный показатель, который важен для организации или процесса. В нашем случае участниками когорты будут законодатели, временными рядами будут сроки полномочий каждого законодателя, а интересующим нас показателем в каждый временной период будет количество тех, кто до сих пор находится на своей должности, начиная со своего первого срока.

Мы начнем с расчета общего процента удержания, а затем перейдем к примерам с различными когортными группами. Первый шаг — найти самую первую дату вступления каждого законодателя в должность (`first_term`). Мы будем использовать эту дату для расчета количества периодов, прошедших с этой начальной даты до каждой следующей даты временного ряда. Для этого возьмем `min` от `term_start` и сгруппируем `GROUP BY` по уникальному идентификатору законодателя `id_bioguide`:

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

На следующем шаге поместим этот код в подзапрос и соединим с временным рядом с помощью `JOIN`. Функция `age` используется для вычисления временного интервала между `term_start` и `first_term` для каждого законодателя. Применение функции `date_part` с параметром `'year'` к этому интервалу преобразует его в количество полных лет. Поскольку выборы происходят каждые 2 года и каждые 6 лет, мы будем использовать года в качестве единицы измерения для расчета периодов. Мы могли бы использовать более короткие периоды, но в этом наборе данных есть небольшие ежедневные и еженедельные колебания. Нас интересует количество законодателей с записями за полный период:

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

**В базах данных, в которых реализована функция** `datediff`, **конструкцию с** `date_part` **и** `age` **можно заменить на более простой вызов:**

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

**Некоторые базы данных, например Oracle, имеют другой порядок аргументов:**

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

Теперь, когда у нас есть периоды и количество законодателей, оставшихся в каждом периоде, последний шаг — вычислить общий размер когорты `cohort_size` и добавить его к каждой строке, чтобы мы могли разделить `cohort_retained` на размер когорты. Оконная функция `first_value` возвращает первую запись в секции `PARTITION BY` в соответствии с порядком, установленным в `ORDER BY`, что является удобным способом получения размера когорты в каждой строке. В нашем случае `cohort_size` берется из первой записи для всего набора данных, поэтому `PARTITION BY` не указан:

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

Чтобы найти процент удержания, разделим значение `cohort_retained` на эту функцию:

```
SELECT period
,first_value(cohort_retained) over (order by period) as cohort_size
,cohort_retained
,cohort_retained * 100.0 /
 first_value(cohort_retained) over (order by period) as pct_retained
```

```
FROM
(
    SELECT date_part('year',age(b.term_start,a.first_term)) as period
    ,count(distinct a.id_bioguide) as cohort_retained
    FROM
    (
        SELECT id_bioguide, min(term_start) as first_term
        FROM legislators_terms
        GROUP BY 1
    ) a
    JOIN legislators_terms b on a.id_bioguide = b.id_bioguide
    GROUP BY 1
) aa
;
```

```
period  cohort_size  cohort_retained  pct_retained
------  -----------  ---------------  ------------
0.0     12518        12518            100.00
1.0     12518        3600             28.76
2.0     12518        3619             28.91
...     ...          ...              ...
```

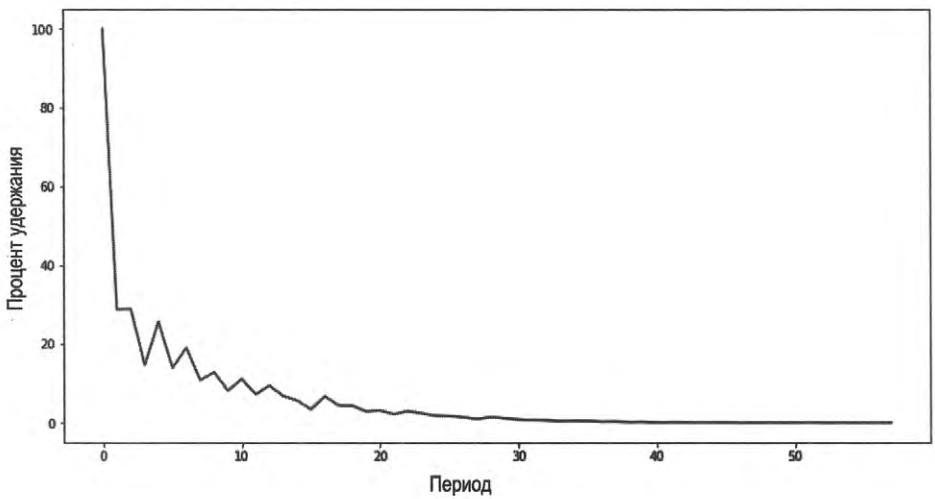


**Рис. 4.3.** Удержание законодателей с начала первого срока

Теперь у нас есть рассчитанный процент удержания, и можно видеть, что есть большая разница между 100% законодателей в когорте в нулевой период 0, т. е. на дату их первого срока, и процентом законодателей со сроком полномочий, который начинается на год позже. График результатов показан на рис. 4.3, и видно, что кри-

вая сглаживается и в конечном итоге стремится к нулю, поскольку даже те законодатели, которые проработали дольше всех, рано или поздно уходят на пенсию или умирают.

Мы можем взять этот запрос и изменить форму выходных данных, чтобы вывести их в виде сводной таблицы. Можно свернуть результаты в столбцы, используя агрегатную функцию с оператором `CASE`. В этом примере я использую функцию `max`, но другие агрегации, например `min` или `avg`, вернут такой же результат. Удержание выводится для первых пяти годов, с 0-го по 4-й период, но можно добавить любые года, следуя этой же схеме:

```
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 * 100.0
     / first_value(cohort_retained) over (order by period)
     as pct_retained
    FROM
    (
       SELECT
       date_part('year',age(b.term_start,a.first_term)) as period
       ,count(distinct a.id_bioguide) as cohort_retained
       FROM
       (
          SELECT id_bioguide, min(term_start) as first_term
          FROM legislators_terms
          GROUP BY 1
        ) a
        JOIN legislators_terms b on a.id_bioguide = b.id_bioguide
        GROUP BY 1
    ) aa
) aaa
GROUP BY 1
;
```

```
cohort_size  yr0     yr1    yr2    yr3    yr4
-----------  ------  -----  -----  -----  -----
12518        100.00  28.76  28.91  14.63  25.64
```

Показатель удержания быстро становится довольно низким, и на графике видно, что в первые несколько лет он скачет. Одной из причин является то, что срок полномочий представителей длится 2 года, сенаторов — 6 лет, а запрос учитывает только записи о начале новых сроков; таким образом, нам не хватает данных за те года, когда законодатель все еще находится на посту, но новый срок еще не начался. Измерение удержания за каждый год в этом случае вводит в заблуждение. Как вариант, можно измерять удержание только для двух- или шестилетних периодов, но есть и другой подход, при котором мы можем заполнить «недостающие» данные. Давайте сначала разберемся с этим, прежде чем перейти к формированию когортных групп.

## Заполнение отсутствующих дат для большей точности

Мы уже обсуждали методы заполнения отсутствующих данных в *разд. 2.5*, и давайте применим их в данном разделе, чтобы получить более плавную и достоверную кривую удержания для законодателей. При работе с данными временных рядов, например в рамках когортного анализа, важно учитывать не только имеющиеся данные, но и то, насколько точно эти данные отражают наличие или отсутствие сущностей в каждый временной период. Это может стать проблемой в тех случаях, когда действие, зафиксированное в данных, приводит к тому, что сущность сохраняется в течение еще некоторого времени, которое не зафиксировано в данных. Например, когда пользователь покупает подписку на программное обеспечение, то в данных фиксируется момент совершения транзакции, но подписка продолжает действовать в течение нескольких месяцев или лет, что не обязательно будет отражено в данных за эти периоды. Чтобы исправить это, нам нужно определить промежуток времени, в течение которого сущность все еще присутствует, с помощью сохраненной даты окончания или продолжительности подписки. Тогда мы сможем сказать, что сущность присутствовала в любой момент времени между датами начала и окончания.

В наборе данных о законодателях у нас есть только одна запись о начале срока полномочий, но нет записей о том, что законодатель продолжает занимать должность в течение последующих двух или шести лет, в зависимости от палаты. Нам нужно заполнить отсутствующие записи за те года, в течение которых законодатель все еще находится у власти. Поскольку этот набор данных содержит значение `term_end` с датой окончания срока полномочий, я покажу, как выполнить более точный анализ удержания, заполнив пробелы между датами начала и окончания срока. Затем я покажу, как можно вычислить дату окончания, если набор данных ее не содержит.

Расчет удержания с использованием начальной и конечной дат, определенных в наборе данных, будет наиболее точным. В следующих примерах мы будем считать, что законодатель удержал свой пост в определенном году, если он все еще нахо-

дился в должности в последний день года, 31 декабря. До принятия Двадцатой поправки к Конституции США срок полномочий законодателей начинался 4 марта, но дата начала была перенесена на 3 января или на последующий рабочий день, если эта дата выпадает на выходные. Законодатели могут быть приведены к присяге и в другие дни года из-за внеочередных выборов или назначений на вакантные места. В результате многие даты `term_start` приходятся на январь, но распределены по всему году. Хотя мы могли бы выбрать любой день, 31 декабря взят просто для нормирования этих разных дат начала срока полномочий.

Первый шаг — создать набор данных, содержащий записи для каждого 31 декабря и для каждого законодателя, находящегося на посту в этот дату. Это можно сделать, присоединив с помощью `JOIN` к подзапросу, находящему `first_term`, таблицу `legislators_terms`, чтобы у нас были `term_start` и `term_end` для каждого срока. Второй `JOIN` с размерной таблицей дат `date_dim` возвращает календарные даты, которые попадают в промежуток между начальной и конечной датами и для которых выполняются условия `c.month_name = 'December'` и `c.day_of_month = 31`. Период рассчитывается как количество полных лет между датой из `date_dim` и `first_term`. Обратите внимание, что хотя между датой принятия присяги в январе и 31 декабря проходит более 11 месяцев, первый год по-прежнему отображается как 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
...          ...         ...         ...         ...          ...
```

Если у вас нет размерной таблицы дат, вы можете создать подзапрос с необходимыми датами несколькими способами. Если ваша база данных поддерживает функцию `generate_series`, подзапрос может выглядеть так:

```
SELECT generate_series::date as date
FROM generate_series('1770-12-31','2020-12-31',
                     interval '1 year')
```

Вы можете сохранить этот результат как таблицу или представление для последующего использования. В качестве альтернативы вы можете выполнить запрос к набору данных или к любой другой таблице в базе, которая имеет полный набор дат. В нашем случае в таблице `legislators_terms` есть все необходимые года, и мы соберем дату 31 декабря для каждого года с помощью функции `make_date`:

```
SELECT distinct make_date(
                date_part('year',term_start)::int,12,31)
FROM legislators_terms
ORDER BY 1
```

Существует несколько хитрых способов получить список необходимых дат. В своих запросах используйте самый простой из доступных.

Теперь у нас есть запись для каждой `date` (на конец года), для которой мы хотели бы рассчитать удержание. Следующим шагом является вычисление `cohort_retained` для каждого периода с помощью функции `count` для уникальных значений `id_bioguide`. Функция `coalesce` применяется к `period`, чтобы вместо `null` возвращалось значение по умолчанию, равное 0. Это касается тех случаев, когда срок полномочий законодателя начинается и заканчивается в одном и том же году, а мы хотим учесть этот год:

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

И на последнем шаге вычислим `cohort_size` и `pct_retained`, как мы это делали ранее, с использованием оконной функции `first_value`:

```
SELECT period
,first_value(cohort_retained) over (order by period) as cohort_size
,cohort_retained
,cohort_retained * 100.0 /
 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            100.00
1.0     12518        12328            98.48
2.0     12518        8166             65.23
...     ...          ...              ...
```

Результаты, представленные на рис. 4.4, теперь более правильные. Почти все законодатели все еще занимают свои должности через год, а первый значительный спад

происходит через 2 года, когда некоторые представители не смогли переизбраться на следующий срок.

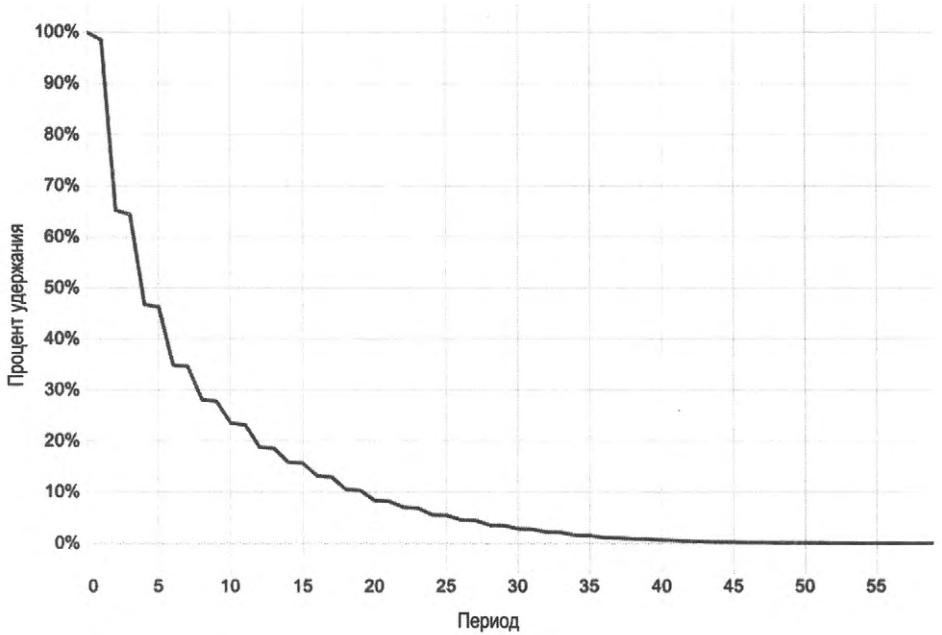


**Рис. 4.4.** Удержание законодателей с поправкой на фактические годы пребывания в должности

Если набор данных не содержит даты окончания, есть несколько способов ее получения. Один из способов — добавить к дате начала фиксированный интервал времени, когда известна продолжительность подписки или срока. Это можно сделать с помощью оператора сложения, добавив постоянный интервал к `term_start`. Например, с помощью оператора `CASE` можно выполнить разное сложение для двух `term_type`:

```
SELECT a.id_bioguide, a.first_term
,b.term_start
,case when b.term_type = 'rep' then b.term_start + interval '2 years'
      when b.term_type = 'sen' then b.term_start + interval '6 years'
      end as term_end
FROM
(
    SELECT id_bioguide, min(term_start) as first_term
    FROM legislators_terms
    GROUP BY 1
) a
JOIN legislators_terms b on a.id_bioguide = b.id_bioguide
;
```

```
id_bioguide  first_term  term_start  term_end
-----------  ----------  ----------  ----------
B000944      1993-01-05  1993-01-05  1995-01-05
C000127      1993-01-05  1993-01-05  1995-01-05
C000141      1987-01-06  1987-01-06  1989-01-06
...          ...         ...         ...
```

Этот фрагмент кода можно использовать в коде, который вычисляет `period` и `pct_retained`. Недостатком этого способа является то, что он не учитывает случаи, когда законодатель не отработал свой полный срок на должности в случае смерти или из-за назначения на более высокую должность.

Второй способ — использовать следующую начальную дату `term_start` минус один день в качестве даты `term_end`. Это можно сделать с помощью оконной функции `lead`. Эта функция похожа на функцию `lag`, которую мы использовали ранее, но вместо того, чтобы возвращать значение из предыдущей строки в секции `PARTITION BY`, `lead` возвращает значение из последующей строки в секции в соответствии с сортировкой `ORDER BY`. По умолчанию возвращается строка, следующая за текущей, что и требуется в нашем случае, но у функции есть необязательный параметр *offset*, позволяющий сместиться на любое количество строк. Мы можем взять дату `term_start` следующего срока с помощью `lead`, а затем вычесть интервал `'1 day'`, чтобы получить условный `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
...          ...         ...         ...
```

Затем этот фрагмент кода можно вставить в код для расчета удержания. У этого способа есть пара недостатков. Во-первых, если законодатель не прошел на следующий срок, функция `lead` вернет `null`, и эта запись останется без `term_end`. В таких случаях по умолчанию можно использовать фиксированный интервал, как было показано в предыдущем примере. Второй недостаток заключается в том, что этот способ предполагает, что сроки полномочий всегда следуют друг за другом, без какого-либо промежутка, проведенного вне занимаемой должности. Хотя большинство законодателей и продолжают работать непрерывно до окончания своей карьеры в Конгрессе, безусловно, существуют промежутки в несколько лет между сроками полномочий.

Каждый раз, когда мы заполняем недостающие данные, нам нужно быть осторожными с предположениями, которые мы делаем. В системах, основанных на подписках или сроках действия, явно сохраненные даты начала и окончания, как правило, дают более точные результаты. Оба описанных способа (добавление к дате начала фиксированного интервала или установка даты окончания относительно следующей даты начала) можно применять, когда дата окончания не известна и у вас есть обоснованные ожидания, что большинство клиентов или пользователей останутся активными в течение предполагаемого временного периода.

Теперь, когда мы знаем, как рассчитать общую кривую удержания и скорректировать отсутствующие даты, мы можем начать работать с когортами. Сравнение показателей удержания различных групп является одной из основных причин выполнения когортного анализа. Сначала я расскажу о формировании групп на основе самого временного ряда, а после этого — на основе данных из других таблиц.

## Когорты, полученные из временного ряда

Теперь, когда у нас есть код SQL для расчета удержания, мы можем начать разбивать сущности на когорты. В этом разделе я покажу, как выделить когорты из самого временного ряда. Сначала я расскажу о временных когортах, основанных на первой дате, а потом объясню, как создавать когорты на основе других атрибутов временного ряда.

Наиболее распространенный способ создания когорт основан на первой или минимальной дате или времени появления сущности во временном ряду. Это означает, что для анализа удержания необходима только одна таблица — сам временной ряд. Объединение в когорты по первому появлению или действию может быть интересно потому, что часто группы, начинающие действовать в разное время, ведут себя по-разному. В сфере потребительских услуг первые клиенты часто проявляют больше энтузиазма и ведут себя иначе, чем пришедшие позднее, тогда как в сфере программного обеспечения типа SaaS более поздние пользователи могут оставаться дольше, потому что продукт уже является более зрелым. Временные когорты могут быть сформированы с разбивкой на любые временные периоды, значимые для организации, но обычно используются еженедельные, ежемесячные или ежегодные когорты. Если вы не уверены, какую группировку выбрать, попробуйте выполнить

когортный анализ для разных группировок, но не слишком мелких, чтобы узнать, когда начинают проявляться значимые закономерности. К счастью, если вы знаете, как формировать когорты и выполнить анализ удержания, то изменение размера детализации по времени становится простой задачей.

Сперва мы будем использовать ежегодные когорты, а затем я покажу как переключиться на столетия. Главный вопрос, который нас интересует, заключается в том, есть ли корреляция между временем, когда законодатель впервые вступил в должность, и его удержанием. Политические тенденции и общественные настроения со временем меняются, но насколько сильно?

Чтобы рассчитать ежегодные когорты, сначала мы добавим год для первой даты `first_term`, уже определенной нами ранее в запросе, который находит `period` и `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      1.0     89
1789.0      2.0     57
...         ...     ...
```

Затем мы используем этот запрос в качестве подзапроса, а значения `cohort_size` и `pct_retained` вычислим во внешнем запросе, как и ранее. Однако в этом случае нам нужно предложение `PARTITION BY` с полем `first_year`, чтобы оконная функция `first_value` выполнялась только для этого `first_year`, а не по всем строкам:

```
SELECT first_year, period
,first_value(cohort_retained) over (partition by first_year
                                    order by period) as cohort_size
```

```
,cohort_retained
,cohort_retained * 100.0 /
 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               100.00
1789.0      1.0     89           89               100.00
1789.0      2.0     89           57               64.04
...         ...     ...          ...              ...
```

Этот набор данных включает в себя более 200 значений `first_year` — слишком много когорт, чтобы их можно было изобразить на одном графике или просмотреть в виде таблицы. Поэтому давайте используем более крупный временной интервал и сгруппируем законодателей по столетиям `first_term`. Это изменение легко выполнить, заменив `'year'` на `'century'` в функции `date_part` в подзапросе `aa`. Напомню, что XVIII век (18-й) длился с 1700 по 1799 г., XIX век (19-й) длился с 1800 по 1899 г. и т. д. Разбиение на секции в функции `first_value` надо заменить на поле `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 * 100.0 /
```

```
 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              100.00
18.0           1.0     368          360              97.83
18.0           2.0     368          242              65.76
...            ...     ...          ...              ...
```

Результаты в виде графика представлены на рис. 4.5. Удержание на своем посту в первые годы службы было выше у тех, кто впервые был избран в XX или XXI веке. XXI век еще не закончился, и поэтому у многих законодателей еще не было возможности продлить свой срок полномочий дольше пяти или более лет, но все они учтены в знаменателе. Возможно, нам нужно исключить из анализа XXI век, но я оставила его на графике, чтобы показать, как кривая удержания искусственно падает до нуля из-за этого обстоятельства.

Кроме первой даты, когорты могут быть определены на основе других атрибутов временного ряда, в зависимости от значений в таблице. В таблице `legislators_terms` есть поле `state`, указывающее, какой штат представляет законодатель на этом сроке. Мы можем использовать это поле для формирования когорт и будем учитывать только первый штат каждого законодателя, чтобы гарантировать, что любой, кто представлял несколько штатов, появится в данных только один раз.

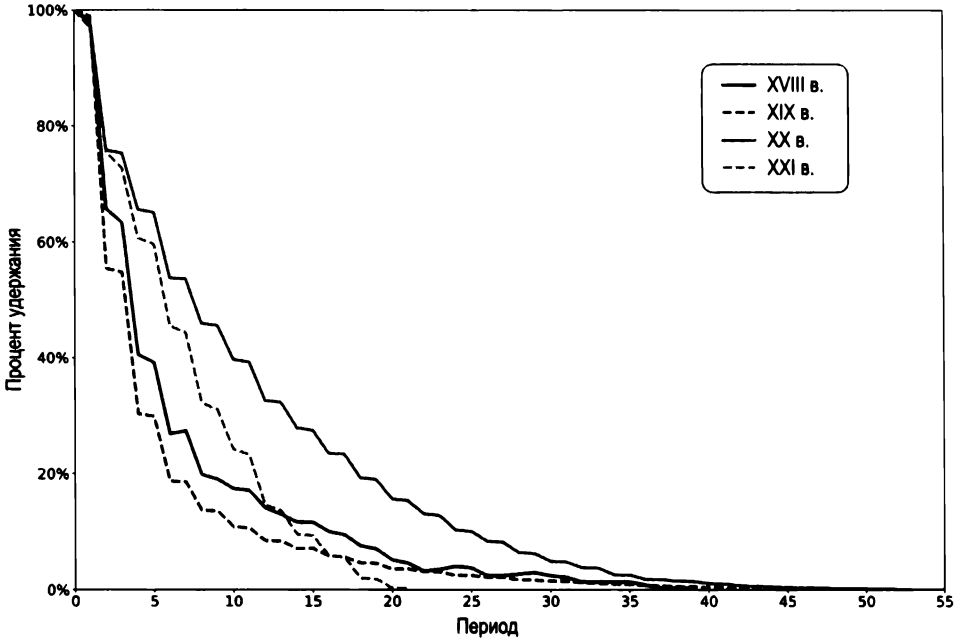


**Рис. 4.5.** Удержание законодателей по столетиям, в которых начался их первый срок

При создании когорт на основе атрибута, который может меняться со временем, важно убедиться, что каждой сущности соответствует только одно значение атрибута. В противном случае сущность может быть представлена в нескольких когортах, что приведет к систематической ошибке в анализе. Обычно используется значение атрибута из самой первой записи в наборе данных.

Чтобы найти первый штат `first_state` для каждого законодателя, мы можем использовать оконную функцию `first_value`. В этом примере мы также превратим `min` в оконную функцию, чтобы избежать длинного оператора `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
...          ...         ...
```

После этого мы можем вставить этот код в написанный нами ранее код, чтобы найти удержание по 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 * 100.0 /
 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               100.00
AK           1.0     19           19               100.00
AK           2.0     19           15               78.95
...          ...     ...          ...              ...
```

Кривые удержания для пяти штатов с наибольшим количеством законодателей представлены на рис. 4.6. У тех, кто был избран в Иллинойсе (IL) и Массачусетсе (MA), самый высокий показатель удержания, а у представителей Нью-Йорка (NY) самый низкий показатель удержания среди этих пяти штатов. Выяснение причин,

почему результаты именно такие, может стать интересной задачей в рамках последующего анализа.

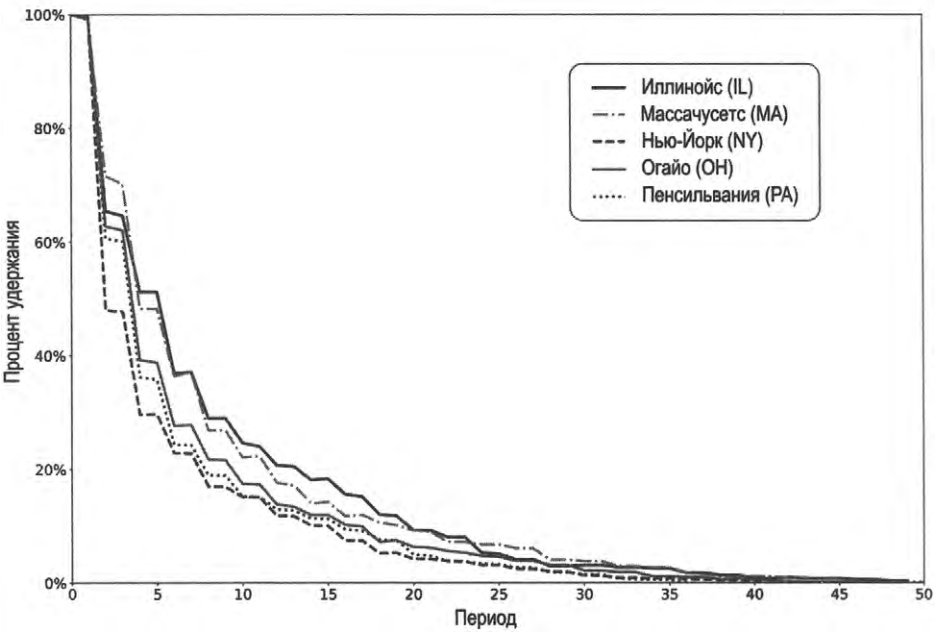


**Рис. 4.6.** Удержание законодателей по первому штату: пять штатов по общему количеству законодателей

Определить когорты по временному ряду довольно просто, например используя минимальную дату для каждой сущности, а затем выделив из этой даты месяц, год или столетие в соответствии с требованиями анализа. Переход от месяца к году или к другому уровню детализации тоже просто сделать, это позволяет вам протестировать несколько вариантов, чтобы найти группировку, значимую для организации. Кроме того, в оконной функции `first_value` для создания когорт можно использовать и другие атрибуты. Далее мы рассмотрим случай, когда атрибут для формирования когорт берется из другой таблицы, отличной от временного ряда.

## Определение когорт по другой таблице

Часто атрибуты, определяющие когорту, хранятся в отдельной таблице, которая не зависит от временного ряда. В базе данных может быть таблица с подробной информацией о клиентах, например откуда они узнали об организации или дата регистрации, и по этим атрибутам можно их группировать. Добавление атрибутов из других таблиц или подзапросов очень просто и может быть выполнено при анализе удержания и связанных анализов, которые будут рассматриваться далее в этой главе.

В следующем примере мы посмотрим, влияет ли пол законодателя на его удержание. В таблице `legislators` есть поле `gender`, в котором значение `'F'` означает женский пол (от *female*), а `'M'` — мужской (от *male*), и это поле мы можем использовать

для формирования когорт законодателей. Мы присоединим с помощью `JOIN` таблицу `legislators` с псевдонимом `d`, чтобы вместо первого года или штата использовать группировку по полу при расчете `cohort_retained`:

```
SELECT d.gender
,coalesce(date_part('year',age(c.date,a.first_term)),0) as period
,count(distinct a.id_bioguide) as cohort_retained
FROM
(
    SELECT id_bioguide, min(term_start) as first_term
    FROM legislators_terms
    GROUP BY 1
) a
JOIN legislators_terms b on a.id_bioguide = b.id_bioguide
LEFT JOIN date_dim c on c.date between b.term_start and b.term_end
and c.month_name = 'December' and c.day_of_month = 31
JOIN legislators d on a.id_bioguide = d.id_bioguide
GROUP BY 1,2
;

gender  period  cohort_retained
------  ------  ---------------
F       0.0     366
M       0.0     12152
F       1.0     349
M       1.0     11979
...     ...     ...
```

Сразу очевидно, что законодательные должности доставались гораздо чаще мужчинам, чем женщинам. Теперь мы можем рассчитать `percent_retained`, чтобы сравнить удержание для этих двух когорт:

```
SELECT gender, period
,first_value(cohort_retained) over (partition by gender
                                    order by period) as cohort_size
,cohort_retained
,cohort_retained * 100.0 /
 first_value(cohort_retained) over (partition by gender
                                    order by period) as pct_retained
FROM
(
    SELECT d.gender
    ,coalesce(date_part('year',age(c.date,a.first_term)),0) as period
    ,count(distinct a.id_bioguide) as cohort_retained
```

```
    FROM
    (
        SELECT id_bioguide, min(term_start) as first_term
        FROM legislators_terms
        GROUP BY 1
    ) a
    JOIN legislators_terms b on a.id_bioguide = b.id_bioguide
    LEFT JOIN date_dim c on c.date between b.term_start and b.term_end
    and c.month_name = 'December' and c.day_of_month = 31
    JOIN legislators d on a.id_bioguide = d.id_bioguide
    GROUP BY 1,2
) aa
;
```

| gender | period | cohort_size | cohort_retained | pct_retained |
|---|---|---|---|---|
| F | 0.0 | 366 | 366 | 100.00 |
| M | 0.0 | 12152 | 12152 | 100.00 |
| F | 1.0 | 366 | 349 | 95.36 |
| M | 1.0 | 12152 | 11979 | 98.58 |
| ... | ... | ... | ... | ... |

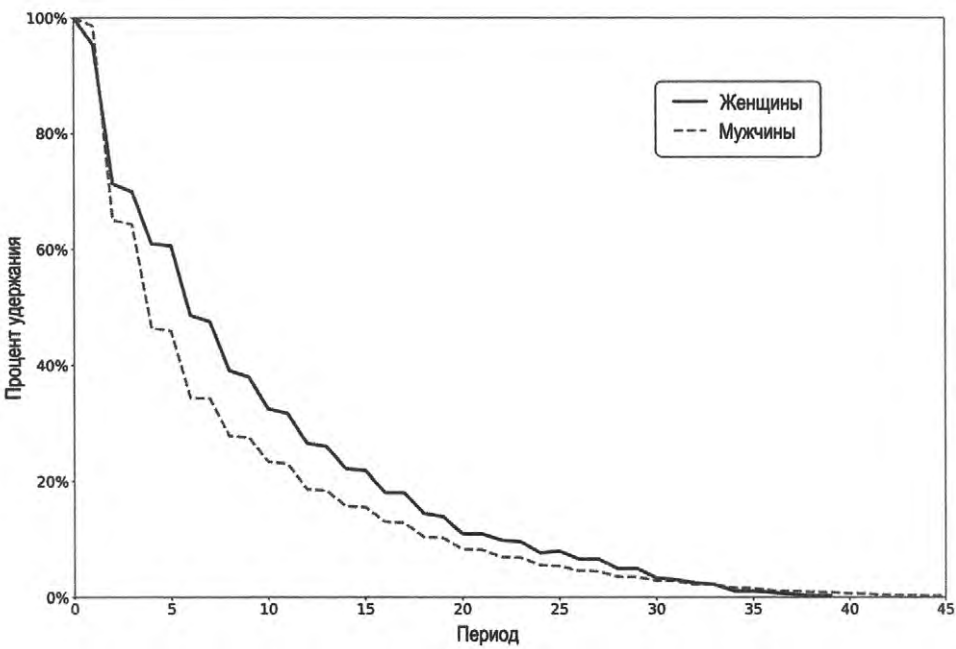


**Рис. 4.7.** Удержание законодателей по полу

Из результатов, представленных на рис. 4.7, видно, что удержание женщин-законодателей выше, чем их коллег-мужчин со 2-го по 29-й период. Женщин не избирали в Конгресс до 1917 г., когда Жаннет Рэнкин вошла в Палату представителей как республиканец от Монтаны. Как мы видели ранее, удержание заметно выросло за последнее столетие.

Чтобы выполнить более корректное сравнение, мы могли бы включить в анализ только тех законодателей, чей первый срок `first_term` начался с того года, как в Конгресс стали избирать женщин. Мы можем сделать это, добавив фильтр `WHERE` к подзапросу `a`. Кроме того, я добавила еще одно ограничение для учета только тех, кто начал работу до 2000 г., чтобы у них в запасе было не менее 20 лет для возможных переизбраний:

```
SELECT gender, period
,first_value(cohort_retained) over (partition by gender
                                    order by period) as cohort_size
,cohort_retained
,cohort_retained * 100.0 /
 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              100.00
```

```
M        0.0      3833          3833          100.00
F        1.0      200           187           93.50
M        1.0      3833          3769          98.33
...      ...      ...           ...           ...
```

Мужчин-законодателей по-прежнему больше, чем женщин-законодателей, но отрыв немного сократился. График удержания для этих двух когорт представлен на рис. 4.8. Для пересчитанных когорт мужчины-законодатели имеют более высокий показатель удержания до 7-го периода, а начиная с 12-го периода женщины-законодатели имеют более высокое удержание. Разница между двумя последними примерами когортного анализами подчеркивает важность создания правильных когорт и необходимость обеспечения того, чтобы у них было сопоставимое количество времени для существования или для выполнения интересующих нас действий. Чтобы еще сильнее улучшить этот анализ, мы могли бы сформировать когорты по первому году или десятилетию и по полу, чтобы оценить дополнительные изменения в удержании на протяжении XX и XXI вв.

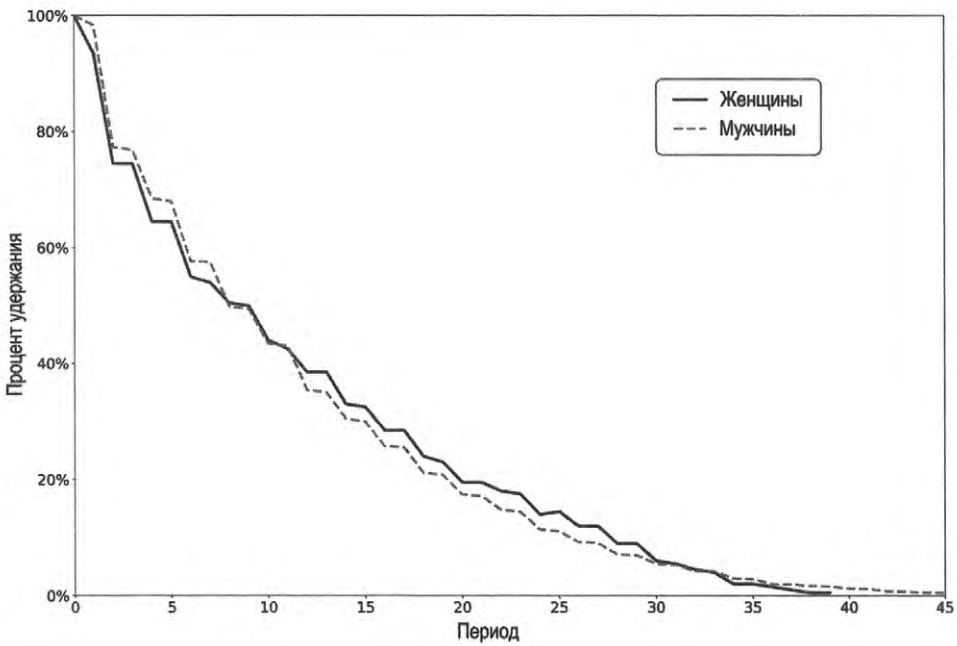


**Рис. 4.8.** Удержание законодателей по полу: когорты с 1917 по 1999 г.

Когорты можно определить разными способами, например по временному ряду и по другим таблицам. Для этого в приведенной структуре запроса можно использовать подзапросы, представления или другие таблицы, что открывает множество возможностей для формирования когорт. Можно использовать несколько атрибутов, например первый год и пол. Но при разделении аудитории на когорты на основе нескольких атрибутов следует учесть, что у вас могут появиться разреженные когорты, кото-

рые слишком малы и представлены в наборе данных не во всех временных периодах. В следующем разделе мы рассмотрим способы решения этой проблемы.

## Работа с разреженными когортами

В идеальном наборе данных для каждой когорты есть какое-то действие или запись во временном ряду для каждого интересующего нас периода. Мы уже видели, почему могут возникать «отсутствующие» даты в случаях с подписками или сроками, длящимися несколько периодов, и мы знаем, как исправить это с помощью размерной таблицы дат для заполнения этих пропусков. Другая проблема может возникнуть, когда из-за критериев группировки когорта становится слишком маленькой, и в результате оказывается, что она представлена в данных лишь время от времени. Когорта может исчезнуть из результатов, хотя мы бы предпочли, чтобы она отображалась с нулевым значением удержания. Такие когорты называют *разреженными*, и эту проблему можно исправить с помощью аккуратного использования соединений `LEFT JOIN`.

В качестве наглядного примера давайте попробуем сгруппировать законодателей по первому штату, который они представляли, чтобы проверить, будут ли какие-либо различия в удержании. Мы уже видели, что женщин-законодателей в принципе достаточно немного. Их дальнейшая группировка по штатам, скорее всего, приведет к появлению нескольких разреженных когорт, в которых будет очень мало участников. Прежде чем вносить изменения в код, давайте добавим `first_state` (мы рассчитывали это значение в разделе о когортах, полученных из временного ряда) в наш предыдущий гендерный пример и посмотрим на результаты:

```
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 * 100.0 /
 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               100.00
AZ          M      0.0    26          26              100.00
AZ          F      1.0    2           2               100.00
...         ...    ...    ...         ...             ...
```

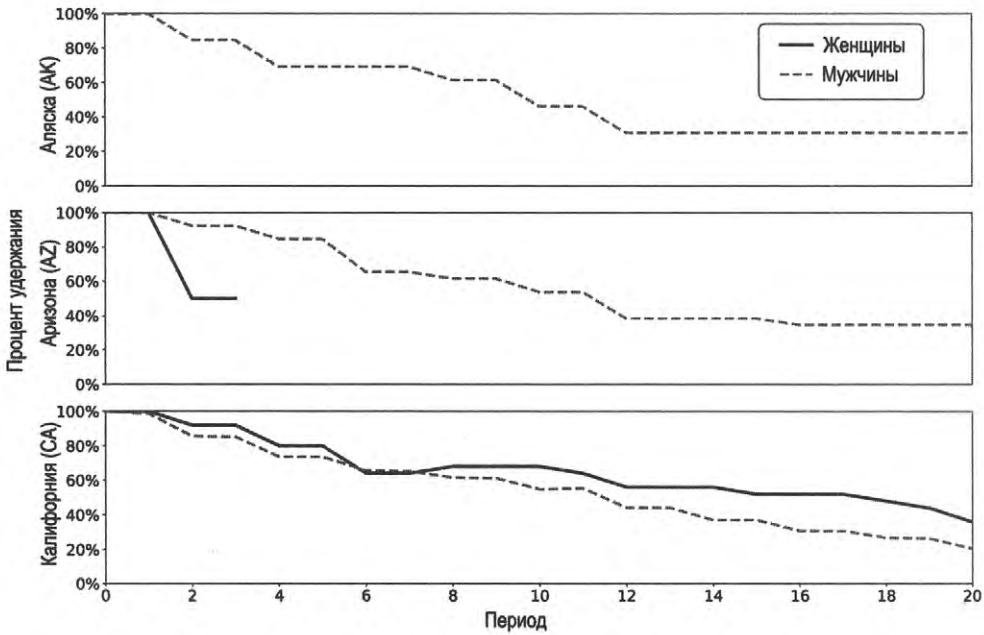


**Рис. 4.9.** Удержание законодателей по полу и первому штату для трех штатов

Графики результатов для первых 20 периодов, приведенные на рис. 4.9, показывают наличие разреженных когорт. На Аляске (AK) не было ни одной женщины-законодателя с 1917 по 1999 г., в то время как для Аризоны (AZ) кривая удержания для женщин исчезает через три года. Только для Калифорнии (CA) — крупного

штата с большим количеством законодателей — получились нормальные кривые удержания для обоих полов. Эта картина будет повторяться для других больших и малых штатов.

Теперь давайте посмотрим, как сделать так, чтобы для каждого периода была запись и запрос возвращал бы нулевые значения вместо `null` при расчете удержания. Первым шагом является написание запроса, возвращающего все комбинации периодов `period` и атрибутов для формирования когорт, в данном случае это `first_state` и `gender`, а также начальный `cohort_size` для каждой комбинации. Это можно сделать, соединив с помощью `JOIN` подзапрос `aa`, который вычисляет когорту, и подзапрос с функцией `generate_series`, который возвращает целые числа от 0 до 20, с условием соединения `on 1 = 1`. Это удобный способ создания декартова соединения, когда два подзапроса не имеют общих полей:

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

Следующим шагом является соединение этого запроса с фактическими периодами пребывания в должности, но с помощью LEFT JOIN, чтобы гарантировать, что все возможные периоды времени останутся в конечном результате:

```
SELECT aaa.gender, aaa.first_state, aaa.period, aaa.cohort_size
,coalesce(ddd.cohort_retained,0) as cohort_retained
,coalesce(ddd.cohort_retained,0) * 100.0 /
 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               100.00
F      AL          1      3           1               33.33
F      AL          2      3           0               0.00
...    ...         ...    ...         ...             ...
```

Затем мы можем свернуть результаты в столбцы и убедиться, что показатель удержания вычислен для каждой когорты в каждом периоде:

```
gender  first_state  yr0    yr2    yr4    yr6    yr8    yr10
------  -----------  -----  -----  -----  -----  -----  ------
F       AL           100.0  0.00   0.00   0.00   0.00   0.00
F       AR           100.0  80.00  20.00  40.00  40.00  40.00
F       CA           100.0  92.00  80.00  64.00  68.00  68.00
...     ...          ...    ...    ...    ...    ...    ...
```

Обратите внимание, что на этом этапе код SQL стал довольно длинным. Практически самым сложным при написании SQL для вычисления удержания в когортном анализе является сохранение четкой логики и упорядоченности кода. Эту тему я подробно рассмотрю в *гл. 8*. При написании кода для расчета удержания я считаю, что полезно двигаться постепенно, усложняя код шаг за шагом и постоянно проверяя результаты. Я также выборочно проверяю некоторые когорты, чтобы убедиться, что окончательный результат верен.

Когорты могут быть определены многими способами. До сих пор мы приводили все сущности в когортах к первой дате их появления во временных рядах. Однако это не единственный вариант, и интересный анализ можно получить, начиная с середины жизненного цикла сущности. Прежде чем завершить изучение анализа удержания, давайте рассмотрим этот дополнительный способ формирования когорт.

## Когорты по датам, отличным от первой даты

Обычно построенные на датах когорты определяются по первому появлению сущности во временном ряду или по любой более ранней дате, такой как дата регистрации. Однако группировка по поздней дате может быть полезной и информационной. Например, нам нужно узнать показатель удержания всех клиентов, использующих наши сервисы, на определенную дату. Этот вид анализа может пригодиться, чтобы понять, оказали ли изменения в нашем сервисе или в маркетинге долгосрочное влияние на существующих клиентов.

При использовании даты, отличной от первой, нам нужно позаботиться о том, чтобы точно определить критерии включения в каждую когорту. Один из способов — выбрать сущности, присутствующие на конкретную календарную дату. Это относительно просто реализовать на SQL, но могут возникнуть проблемы. Если большинство постоянных клиентов не пользуются сервисами каждый день, это может привести к тому, что удержание будет сильно варьироваться в зависимости от выбранного дня. Чтобы исправить подобную ситуацию, можно рассчитать удержание для нескольких начальных дат, а затем усреднить результаты.

Другой способ — использовать временное окно, например неделю или месяц. Любая сущность, которая появляется в наборе данных на протяжении этого окна, включается в когорту. Как правило, такой подход более репрезентативен для бизнеса или процесса, но при этом код SQL становится более сложным, а время выполнения запросов может увеличиться из-за более интенсивных вычислений в базе данных. Поиск разумного компромисса между производительностью запросов и точностью результатов — это своего рода искусство.

Давайте посмотрим, как выполнить такой промежуточный анализ для набора данных о законодателях. Например, вычислим удержание для законодателей, которые находились у власти в 2000 г. Мы сформируем две когорты по полю `term_type`, которое имеет значение `'sen'` для сенаторов и `'rep'` для представителей. В наш анализ попадет каждый законодатель, занимавший пост в Конгрессе в какой-либо день 2000 года: те, кто начал свою работу до 2000 г. и чьи сроки полномочий закончились в течение этого года или позднее, а также те, кто вступил в должность в 2000 г. Мы можем явно указать любую дату в 2000 г. в качестве `first_term`, т. к. в условии `WHERE` мы проверим, находился ли законодатель в должности в течение 2000 г. Вычисляется также значение `min_start` сроков, попадающих в это временное окно, для использования этой даты на более позднем шаге:

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

Затем мы можем вставить этот фрагмент в наш код для вычисления удержания, но с двумя небольшими поправками. Во-первых, добавлено новое условие в `JOIN` между подзапросом `a` и таблицей `legislators_terms` для того, чтобы возвращались только те сроки полномочий, который начались в дату `min_start` и позже. Во-вторых, добавлено дополнительное условие `WHERE` для таблицы `date_dim`, чтобы возвращались только даты 2000 года или более поздние:

```
SELECT term_type, period
,first_value(cohort_retained)
 over (partition by term_type order by period) as cohort_size
,cohort_retained
,cohort_retained * 100.0 / 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              100.00
sen        0.0     101          101              100.00
rep        1.0     440          392              89.09
sen        1.0     101          89               88.12
...        ...     ...          ...              ...
```

Из рис. 4.10 видно, что, несмотря на более длительный срок полномочий сенаторов, удержание в двух когортах схожее, а для сенаторов оно даже становится хуже через 10 лет. Дальнейший анализ, сравнивающий разные годы, когда законодатели были впервые избраны, или использующий другие атрибуты для формирования когорт, может привести к некоторым интересным выводам.

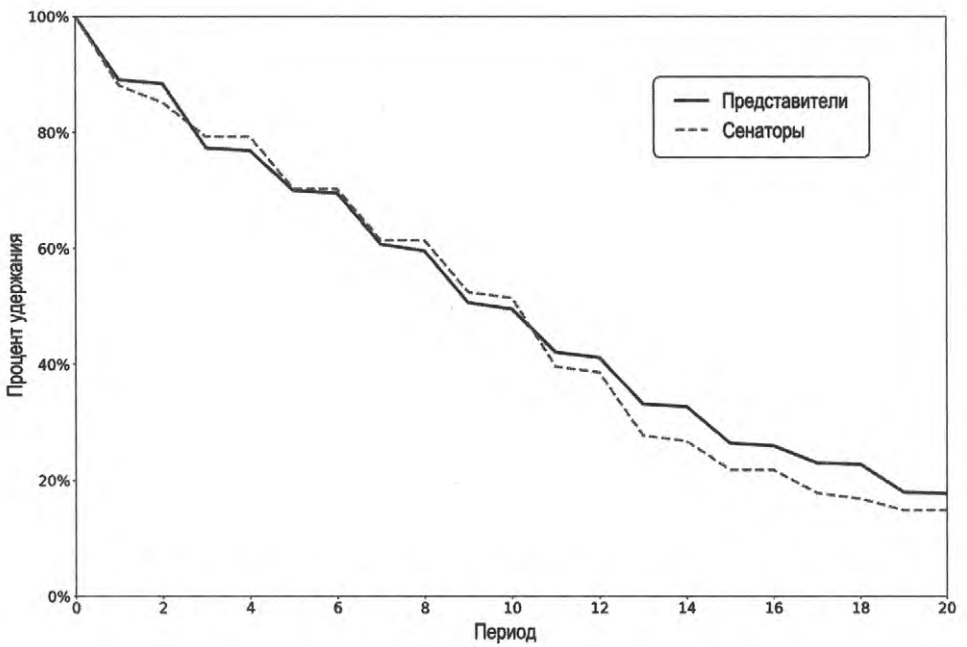


**Рис. 4.10.** Удержание законодателей, занимающих должности в 2000 г., по `term_type`

Типичным случаем формирования когорт по атрибуту, отличному от начального значения, является анализ удержания после того, как сущность достигла заданного порогового значения, например, определенного количества покупок или определенной суммы расходов. Как и в случае с любым делением на когорты, важно внимательно отнестись к определению того, что позволяет сущности быть в когорте и какая дата будет считаться начальной.

Анализ удержания по когортам — это эффективный способ понять поведение сущностей во временном ряду. Мы разобрали, как рассчитать удержание с помощью SQL и как сформировать когорты на основе самого временного ряда или других таблиц, а также из точек в середине жизненного цикла сущности. Мы также рассмотрели, как использовать функции и `LEFT JOIN` для корректировки дат временного ряда и для заполнения разреженных когорт. Существует несколько видов анализов, связанных с удержанием по когортам: анализ выживаемости, возвращаемость и накопительный итог. Для них тоже можно использовать SQL-код, который мы написали для расчета удержания. Давайте рассмотрим их подробнее.

## 4.4. Связанные когортные анализы

В предыдущем разделе мы узнали, как написать SQL для анализа удержания по когортам. Удержание позволяет определить, присутствовала ли сущность во временном ряду на конкретную дату или на протяжении временного окна. Кроме проверки присутствия на определенную дату, с помощью анализа часто пытаются ответить на вопрос о том, как долго находится сущность в наборе данных, совершала ли она несколько действий и сколько таких повторных действий произошло. На все эти вопросы можно ответить с помощью кода, который похож на расчет удержания и подходит практически для любых критериев создания когорт, которые вам могут понадобиться. Давайте рассмотрим первый из связанных анализов — анализ выживаемости.

### Выживаемость

*Анализ выживаемости* отвечает на вопросы о том, как долго что-то продлится или сколько пройдет времени до определенного критического события, такого как отток клиентов или смерть. Анализ выживаемости может ответить на вопросы о доле аудитории, которая, вероятно, сохранится после какого-то периода времени. Когорты могут помочь определить или, по крайней мере, выдвинуть гипотезы о том, какие показатели или обстоятельства увеличивают или уменьшают показатель выживаемости.

Этот анализ похож на анализ удержания, но вместо вычисления того, присутствовала ли сущность в определенный период, мы вычисляем, присутствовала ли она в тот период или позднее во временном ряду. Затем рассчитывается доля от общей когорты. Обычно выбирается один или несколько периодов в зависимости от характера анализируемого набора данных. Например, если мы хотим узнать долю иг-

роков, которые продолжают играть в течение недели или дольше, мы можем проверить действия, которые происходят через неделю после начала игры, и посчитать тех игроков, которые еще «выжили». Другой пример: если нас интересует число учеников, которые все еще учатся в школе после определенного количества лет, мы можем использовать отсутствие записи об окончании школы в наборе данных. Количество периодов можно выбрать, либо рассчитав среднюю или стандартную продолжительность жизни, либо взяв продолжительность, значимую для этой организации или анализируемого процесса, например месяц, год или более длительный период.

В данном примере мы вычислим долю законодателей, которые продержались на своем посту в течение 10 и более лет после начала их первого срока. Так как нам не нужны конкретные даты каждого срока, мы можем сначала определить `term_start` первого и последнего сроков, используя агрегатные функции `min` и `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
...          ...         ...
```

Далее мы добавляем в запрос функцию `date_part`, чтобы найти столетие для `min` от `term_start`, и рассчитываем срок пребывания в должности (`tenure`) как количество лет между `min` и `max` от `term_start`, найденное с помощью функции `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
...          ...           ...         ...         ...
```

Наконец, мы вычисляем `cohort_size` с помощью функции `count` для всех законодателей, а также вычисляем число тех, кто пробыл на посту не менее 10 лет, используя оператор `CASE` и агрегацию `count`. Процент "выживших" определяется путем деления этих двух значений:

```
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)
 * 100.0 / 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           22.55
19       6299    892          14.16
20       5091    1853         36.40
21       760     119          15.66
```

Поскольку сроки полномочий могут идти и не последовательно, мы можем рассчитать долю законодателей в каждом столетии, которые проработали пять и более сроков. В подзапросе добавим `count`, чтобы найти общее количество сроков для каждого законодателя. Затем во внешнем запросе разделим количество законодателей с пятью и более сроками на общую численность когорты:

```
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)
 * 100.0 / 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          17.12
19       6299    711         11.29
20       5091    2153        42.29
21       760     205         26.97
```

Десять лет или пять сроков — это произвольно выбранные цифры. Мы также можем рассчитать выживаемость для каждого количества лет или периодов и представить результаты в виде графика или таблицы. В следующем примере мы рассчитаем выживаемость для каждого количества сроков от 1 до 20. Это выполняется с помощью декартова соединения с подзапросом, который содержит целые числа, сгенерированные функцией `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)
 * 100.0 / count(distinct id_bioguide) as pct_survived
FROM
(
    SELECT id_bioguide
    ,date_part('century',min(term_start)) as first_century
    ,count(term_start) as total_terms
    FROM legislators_terms
```

```
    GROUP BY 1
) a
JOIN
(
    SELECT generate_series as terms
    FROM generate_series(1,20,1)
) b on 1 = 1
GROUP BY 1,2
;
```

```
century  terms  cohort  cohort_survived  pct_survived
-------  -----  ------  ---------------  ------------
18       1      368     368              100.00
18       2      368     249              67.66
18       3      368     153              41.57
...      ...    ...     ...              ...
```

Результаты представлены на рис. 4.11. Выживаемость была самой высокой в XX веке, что согласуется с результатами, которые мы получили ранее, где удержание также было самым высоким в XX веке.

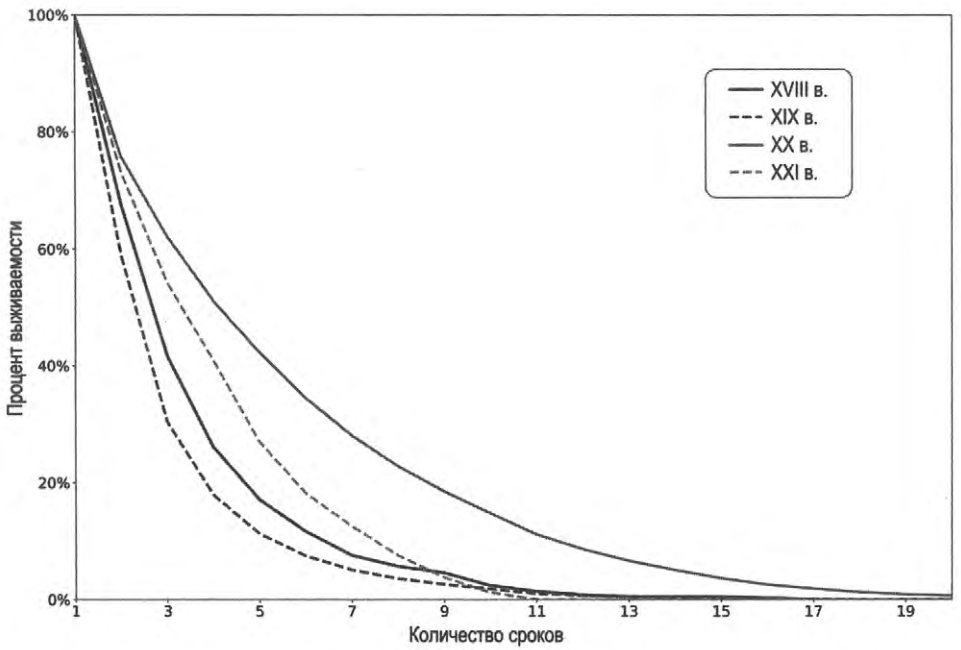


**Рис. 4.11.** Процент выживаемости законодателей по столетиям: кто оставался на посту определенное количество сроков или дольше

Выживаемость тесно связана с удержанием. Если при удержании учитываются сущности, присутствующие в течение заданного количества периодов с самого начала, то при выживаемости учитываются те из них, которые присутствовали в течение определенного периода или дольше. В результате код упрощается, поскольку для вычисления выживаемости нужно знать только первую и последнюю даты во временном ряду или количество сроков. Формирование когорт для выживаемости выполняется так же, как и для удержания, а определение когорты может опираться на данные временного ряда или другой таблицы, или подзапроса.

Далее мы рассмотрим другой тип анализа, который в некотором смысле является обратным по отношению к анализу выживаемости. Вместо того чтобы вычислять, присутствует ли сущность в наборе данных определенное время или дольше, мы будем вычислять, возвращается ли сущность или повторяет действие в определенный период или раньше.

## Возвращаемость или поведение при повторной покупке

Выживаемость нужна для понимания того, как долго когорта будет существовать. Другой полезный тип когортного анализа направлен на то, чтобы понять, можно ли ожидать возвращения участника когорты в течение определенного временного окна и какова активность в этом окне. Это называется *возвращаемостью* или *поведением при повторной покупке*.

Например, для интернет-магазина можно выяснить не только то, сколько новых покупателей было привлечено в результате маркетинговой кампании, но и делали ли они повторные заказы. Один из способов выяснить это — просто подсчитать общее количество покупок на одного покупателя. Однако сравнивать покупателей, пришедших два года назад, с покупателями, пришедшими всего месяц назад, не совсем корректно, поскольку у первых было гораздо больше времени на возвращение. Более старая когорта почти наверняка окажется ценнее, чем более новая. Хотя в некотором смысле это и верно, но не дает полного представления о том, как когорты могут повести себя на протяжении всего жизненного цикла.

Чтобы выполнить корректное сравнение когорт с разными начальными датами, нам нужно сделать анализ на основе *фиксированного временного окна*, начиная с первой даты, и определить, вернулись ли участники когорты в течение этого окна. Таким образом, все когорты будут рассматриваться на равных промежутках времени, если мы возьмем только те из них, для которых прошло полное окно. Анализ возвращаемости часто выполняется для розничной торговли, но его можно применять и в других сферах. Например, университет может захотеть узнать, сколько студентов записалось на второй курс, или больнице нужно знать, сколько пациентов нуждаются в последующем лечении после первого обращения.

В качестве примера анализа возвращаемости мы можем задать новый вопрос о нашем наборе данных: сколько законодателей занимали должности в обеих палатах, и, в частности, какая доля из них начинала как представитель и впоследствии стала сенатором (некоторые сенаторы позднее становятся представителями, но это случа-

ется гораздо реже). Поскольку этот переход совершают относительно немногие, мы будем составлять когорты законодателей по столетию, в котором они впервые стали представителями.

Первый шаг — найти размер когорты для каждого столетия, используя подзапрос и `date_part`, рассмотренные ранее, только для тех, у кого `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
```

Далее мы выполним соединение `JOIN` с таблицей `legislators_terms`, чтобы найти представителей, которые позже стали сенаторами. Для этого используются условия `b.term_type = 'sen'` и `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
```

Наконец, мы соединяем эти два подзапроса вместе с помощью `LEFT JOIN` и вычисляем процент представителей, ставших сенаторами. Здесь обычно используется левое соединение, чтобы все когорты были включены в результат независимо от того, произошло ли последующее событие или нет. Если есть столетие, в котором ни один представитель не стал сенатором, мы все равно хотим включить его в результаты:

```
SELECT aa.cohort_century
,bb.rep_and_sen * 100.0 / 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 = 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_rep_and_sen
--------------  ---------------
18              19.06
19              5.70
20              5.67
21              3.66
```

Представители XVIII в. чаще всего становились сенаторами. Однако мы еще не применили временное окно, чтобы выполнить корректное сравнение. Мы точно знаем, что всех законодателей, работавших в XVIII и XIX веках, уже нет в живых, но многие из тех, кто впервые был избран в XX или XXI веке, все еще продолжают свою деятельность. Добавление фильтра `WHERE age(c.term_start, b.first_term) <= interval '10 years'` к подзапросу `bb` задает временное окно в 10 лет. Обратите внимание, что размер окна можно легко увеличить или уменьшить, изменив значение интервала. Дополнительный фильтр `first_term <= '2009-12-31'`, примененный к подзапросу `a`, исключает тех, чья карьера на момент сбора данных составляла менее 10 лет:

```
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              9.70
19              2.44
20              3.48
21              7.64
```

С учетом новых исправлений XVIII век по-прежнему имеет самый высокий процент представителей, ставших сенаторами в течение 10 лет, но XXI век имеет второй по величине процент, а XX век имеет больший процент, чем XIX век.

Поскольку 10 лет — это произвольно выбранный промежуток, то мы можем рассчитать и сравнить несколько временных окон. Один из способов — выполнить запрос несколько раз для разных интервалов и сохранить результаты. Другой способ — вычислить несколько окон в одном и том же запросе, используя оператор `CASE` внутри агрегации `count distinct` при формировании интервалов, вместо условия на интервал в `WHERE`:

```
SELECT aa.cohort_century
,bb.rep_and_sen_5_yrs * 100.0 / aa.reps as pct_5_yrs
,bb.rep_and_sen_10_yrs * 100.0 / aa.reps as pct_10_yrs
,bb.rep_and_sen_15_yrs * 100.0 / 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              5.02       9.70        14.38
19              0.88       2.44        4.09
20              1.00       3.48        4.78
21              4.00       7.64        8.73
```

С помощью этих результатов мы можем оценить, как со временем менялась доля представителей, ставших сенаторами, как внутри каждой когорты, так и между ко-

гортами. Кроме анализа таблицы результатов, построение графиков часто позволяет увидеть интересные тенденции.

На рис. 4.12 приведены результаты анализа, в котором когорты, основанные на столетии первой даты, заменены когортами, основанными на первом десятилетии, и показаны тенденции для временных окон в 10 и 20 лет. Переход представителей в сенаторы в течение первых нескольких десятилетий работы нового законодательного органа США явно отличался от тенденций в последующие годы.


**Рис. 4.12.** Процент представителей, ставших сенаторами, для каждой когорты, определенной по первому десятилетию, для разных временных окон

Поиск повторных действий в пределах фиксированного временного окна является полезным инструментом для сравнения когорт. Это особенно актуально, когда поведение носит непостоянный характер, например покупка товаров или услуг. В следующем разделе мы рассмотрим, как определить не только наличие повторных действий, но и сколько их было, и рассчитаем их накопительный итог.

## Накопительный итог

Накопительный когортный анализ можно использовать для определения *накопительной пожизненной ценности* (customer lifetime value, CLTV), которая также иногда называется *пожизненной ценностью клиента* (lifetime value, LTV), и для мониторинга новых когорт, чтобы предсказать, какой будет их полная пожизненная ценность LTV. Это предсказание возможно потому, что поведение на ранних этапах зачастую сильно коррелирует с поведением в долгосрочной перспективе. Поль-

зователи сервиса, которые неоднократно возвращаются в первые дни или недели пользования им, с наибольшей вероятностью останутся на длительный срок. Клиенты, которые делают покупки два или три раза в самом начале, скорее всего, продолжат покупать и в течение более длительного периода времени. Подписчики, которые продлевают подписку после первого месяца или первого года, как правило, продолжают ее покупать на протяжении многих месяцев или лет.

В этом разделе я поговорю в основном о приносящих доход действиях клиентов, но этот анализ можно применить и к ситуациям, в которых клиенты или сущности приносят расходы, например возврат товаров, обращение в службу поддержки или использование медицинских услуг.

При расчете накопительного итога нас меньше заботит, совершила ли сущность какое-либо действие в определенную дату, а больше интересует общая сумма на определенную дату. Накопительный итог, используемый в этом типе анализа, чаще всего представляет собой подсчет количества `count` или суммы `sum`. Мы снова будем использовать временные окна, чтобы обеспечить корректное сравнение когорт. Давайте найдем количество сроков, начатых в течение 10 лет после первого `term_start`, разбив законодателей на когорты по столетиям и типу первого срока:

```
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
```

Самая многочисленная когорта — это представители, впервые избранные в XIX веке, но когорта с наибольшим количеством сроков, начавшихся в течение первых 10 лет, — это представители, впервые избранные в XX веке. Такого типа расчет может пригодиться для оценки общего вклада когорты в организацию. Общий объем продаж или общий объем повторных покупок могут быть ценными показателями. Однако, как правило, мы хотим нормализовать такие показатели, чтобы оценить вклад, приносимый каждой сущностью. К этим показателям можно отнести среднее количество действий на человека, средний чек (average order value, AOV), количество товаров на один заказ и количество заказов на одного клиента. Для нормализации по размеру когорты нужно просто разделить на начальную когорту, что мы уже делали ранее с показателями удержания, выживаемости и возвращаемости. Здесь мы сделаем аналогично, а также сведем результаты в таблицу для более удобного сравнения:

```
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) * 1.0
     / 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 |

Имея накопительное количество сроков, нормализованное по размеру когорты, мы можем подтвердить, что представители, впервые избранные в XX веке, имели наибольшее среднее количество сроков, в то время как те, кто начал работать в XIX веке, имели в среднем наименьшее количество сроков. У сенаторов среднее количество сроков меньше, но сами сроки продолжительнее, чем у их коллег-представителей, и, опять же, у тех сенаторов, кто начал избираться в XX веке, среднее количество сроков было наибольшим.

Накопительные итоги часто применяются при расчете пожизненной ценности клиента (LTV). Она обычно высчитывается с использованием денежных показателей, таких как общая сумма, потраченная клиентом, или валовая прибыль (выручка минус затраты), привнесенная клиентом за все время взаимодействия с организацией. Чтобы упростить сравнение когорт, в качестве «срока жизни» часто выбирается средний «срок жизни» клиентов или количество периодов, удобных для анализа, например 3, 5 или 10 лет. Набор данных о законодателях не содержит финансовых цифр, но вам не составит труда переделать любой из предыдущих SQL-запросов для использования денежных сумм. К счастью, SQL — достаточно гибкий язык, и вы можете адаптировать эти запросы для решения различных аналитических задач.

Когортный анализ включает в себя набор методов, которые могут быть использованы для поиска ответов на вопросы, касающиеся поведения во времени и того, как разные атрибуты влияют на различия между группами. Выживаемость, возвращаемость и накопительный итог помогают пролить свет на эти вопросы. Получив общее представление о поведении когорт, мы должны дополнительно рассмотреть разный состав или сочетания когорт с течением времени, чтобы проверить, как это влияет на общее удержание, выживаемость, возвращаемость или накопительные итоги, потому что эти показатели могут неожиданным образом отличаться от показателей для отдельных когорт.

## 4.5. Поперечный анализ через все когорты

До сих пор в этой главе мы рассматривали когортный анализ. Мы проследили поведение когорт во времени с помощью анализов удержания, выживаемости, возвращаемости и накопительного итога. Однако одна из главных проблем этих анализов заключается в том, что даже если они позволяют легко увидеть изменения в когортах, бывает трудно обнаружить изменения в общем составе клиентской или пользовательской базы.

Могут также происходить *смешанные сдвиги*, которые меняют состав клиентской или пользовательской базы с течением времени, из-за чего более поздние когорты отличаются от более ранних. Смешанные сдвиги могут быть вызваны международным влиянием, переходом от органической к платной стратегии привлечения клиентов или переходом от узкой нишевой аудитории энтузиастов к более широкой аудитории массового рынка. Формирование дополнительных когорт по любому из этих предполагаемых направлений может помочь диагностировать, происходит ли такой смешанный сдвиг.

Когортный анализ можно противопоставить поперечному анализу (cross-sectional analysis), при котором сравниваются отдельные клиенты или группы в один момент времени. Например, в поперечных исследованиях можно сравнить годы обучения с текущим доходом. Положительным моментом является то, что сбор данных для поперечного анализа зачастую намного проще, т. к. для него не нужны временные ряды. Поперечный анализ может быть очень информативным и привести к новым гипотезам для дальнейших исследований. Его недостатком является то, что обычно есть некоторая предвзятость отбора, которая называется *ошибкой выжившего* и может привести к ложным выводам.

**Ошибка выжившего**

«Давайте посмотрим на наших лучших клиентов и выясним, что у них общего» — эта, казалось бы, невинная идея может привести к весьма сомнительным выводам. *Ошибка выжившего* — это логическая ошибка, заключающаяся в фокусировании внимания на людях или предметах, которые прошли некий процесс отбора, и игнорировании тех, кто этот отбор не прошел. Обычно это происходит потому, что на момент отбора этих сущностей уже нет в наборе данных, поскольку они потерпели неудачу, изменились или покинули выборку по какой-либо другой причине. Концентрация только на оставшейся аудитории может привести к чересчур оптимистичным выводам, поскольку неудачи игнорируются.

Много пишут о тех редких людях, которые бросили учебу в университете и основали невероятно успешные технологические компании. Это не означает, что вы должны тут же бросить учебу, поскольку подавляющее большинство людей, которые так сделали, не стали успешными руководителями. О них не публикуют столько сенсационных статей, поэтому легко забыть об этой стороне реальной жизни.

Если анализировать лучших клиентов, то ошибка выжившего может проявиться как утверждение о том, что лучшие клиенты, как правило, живут в Москве или Санкт-Петербурге и это возрастная категория от 18 до 30 лет. Начнем с того, что это слишком широкая выборка, и может оказаться, что эти характеристики имеют многие клиенты, которые отсеялись еще до даты анализа. Возвращаясь к исходной выборке, можно обнаружить, что другие демографические группы, например 41–50-летние жители Казани, остаются дольше и со временем покупают больше, хотя в абсолютном выражении их меньше. Когортный анализ помогает выявить и снизить ошибку выжившего.

В когортном анализе можно избежать ошибки выжившего путем включения в анализ всех участников начальной когорты. Мы можем взять серию поперечных срезов, чтобы увидеть, как меняется группа различных сущностей с течением времени. На любую определенную дату будут анализироваться пользователи из разных когорт. Мы можем исследовать эти поперечные срезы, подобно слоям осадочных пород, чтобы выявить новые закономерности. В нашем примере мы выделим временной ряд для части законодателей за каждый год из общего набора данных.

Для начала найдем количество законодателей, занимающих свои должности, для каждого года, объединив таблицы `legislators` и `date_dim` с условием, что дата `date` из `date_dim` попадает между датами начала и окончания каждого срока. Здесь мы используем дату 31 декабря, чтобы найти законодателей, находящихся в должности на конец каждого года:

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

Далее мы добавим столетие для формирования когорт с помощью `JOIN` с подзапросом с `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
...         ...      ...
```

Наконец, для каждого года вычислим процент от общего количества `legislators` в каждой когорте. Это можно сделать несколькими способами, в зависимости от того, в какой форме должен быть представлен результат. Первый способ — вывести построчно каждую комбинацию `date` и `century` и использовать оконную функцию `sum` в знаменателе для вычисления процента:

```
SELECT date
,century
,legislators
,sum(legislators) over (partition by date) as cohort
,legislators * 100.0 / 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     22.63
2018-12-31  21       417          539     77.37
2019-12-31  20       97           537     18.06
2019-12-31  21       440          537     81.94
...         ...      ...          ...     ...
```

Второй способ выводит по строке на каждый год, по столбцу на каждое столетие — в виде сводной таблицы, по которой проще увидеть тенденции:

```
SELECT date
,coalesce(sum(case when century = 18 then legislators end)
          * 100.0 / sum(legislators),0) as pct_18
,coalesce(sum(case when century = 19 then legislators end)
          * 100.0 / sum(legislators),0) as pct_19
,coalesce(sum(case when century = 20 then legislators end)
          * 100.0 / sum(legislators),0) as pct_20
,coalesce(sum(case when century = 21 then legislators end)
          * 100.0 / 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       23.05   76.95
2018-12-31  0       0       22.63   77.37
2019-12-31  0       0       18.06   81.93
...         ...     ...     ...     ...
```

Если построить график, как на рис. 4.13, то можно увидеть, как новые когорты законодателей постепенно вытесняют старые, пока их самих не заменяют последующие.

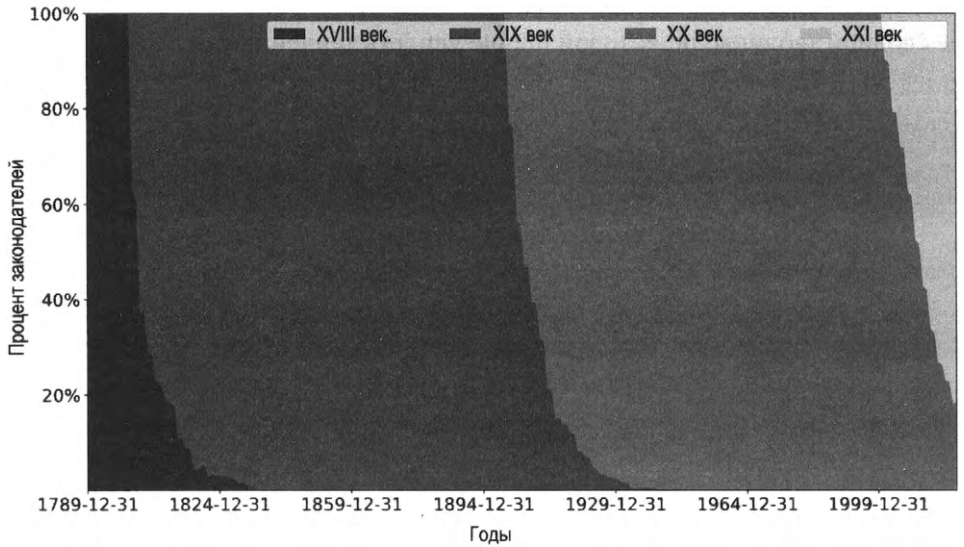


**Рис. 4.13.** Процент законодателей каждый год по столетиям первого срока

Вместо того чтобы создавать когорты по дате `first_term`, мы можем использовать длительность срока пребывания в должности. Определение доли клиентов, которые являются относительно новыми или имеют средний «срок жизни», или длительный «срок жизни» в различные моменты времени, может быть очень полезным. Давайте посмотрим, как менялся стаж законодателей в Конгрессе с течением времени.

Первый шаг — для каждого года и для каждого законодателя подсчитать совокупное количество лет пребывания на посту. Поскольку между сроками могут быть пропуски, когда законодатели покидали свой пост по каким-то причинам, сначала

составим подзапрос, который выводит каждый год, когда законодатель находился на посту на конец года. Затем используем оконную функцию `count` с заданными рамками окна, включающими в себя все предыдущие строки `unbounded preceding` и текущую строку `current row` для каждого законодателя:

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

Затем подсчитаем количество законодателей для каждой комбинации `date` и `cume_years`:

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

Прежде чем рассчитать процентное соотношение для каждой продолжительности пребывания на посту за каждый год, мы, возможно, захотим их сгруппировать. При беглом просмотре наших результатов видно, что в некоторые годы можно получить почти 40 различных сроков пребывания на посту. Это, скорее всего, будет трудно визуализировать и интерпретировать:

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

Поэтому давайте сгруппируем значения. Не существует единственно правильного способа группировки «срока жизни». Если в вашей организации существует какое-то устоявшееся деление на группы, используйте его. В противном случае я обычно стараюсь разбить все значения на три-пять групп примерно одинакового размера. В нашем примере мы сгруппируем сроки пребывания в должности в четыре когорты: `cume_years` меньше или равно 4 годам, от 5 до 10 лет, от 11 до 20 лет, от 21 года и более:

```
SELECT date, tenure
,legislators * 100.0 / 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    29.98
2019-12-31  5 to 10   32.03
2019-12-31  11 to 20  20.11
2019-12-31  21+       17.88
...         ...       ...
```

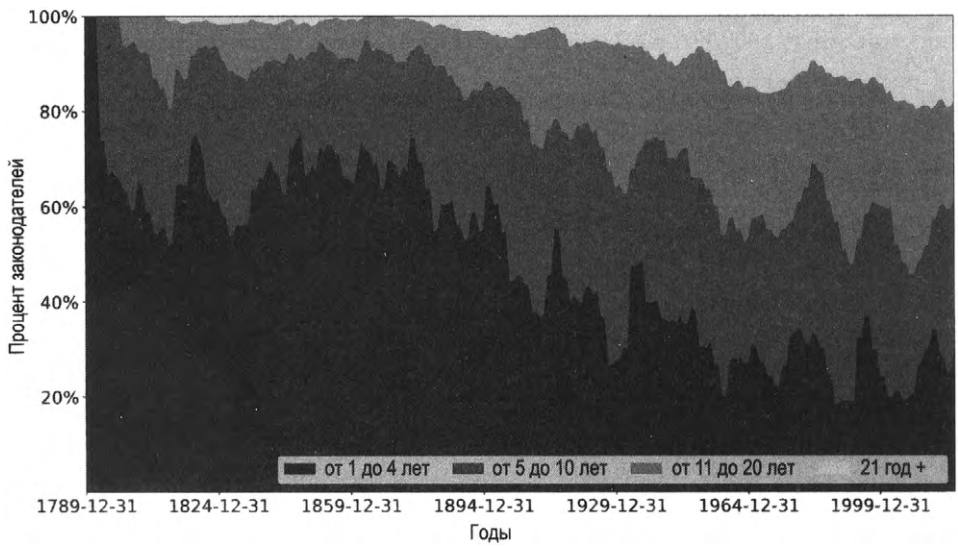


**Рис. 4.14.** Процент законодателей по продолжительности пребывания в должности

Из графика результатов на рис. 4.14 видно, что в первые годы существования Конгресса большинство законодателей имело очень короткий срок пребывания в должности. В последующие годы доля законодателей, занимающих свои посты 21 год и более, неуклонно растет. Также наблюдается интересное периодическое увеличение доли законодателей со сроком полномочий от 1 до 4 лет, что может отражать изменения в политических тенденциях.

Поперечный срез аудитории в любой момент времени должен состоять из представителей нескольких когорт. Создание временного ряда для таких срезов — еще один интересный способ анализа тенденций. Сочетание этого метода с анализом удержания может дать более полное представление о трендах в любой организации.

## 4.6. Заключение

Когортный анализ — это удобный способ изучения того, как группы меняются со временем, с точки зрения удержания, возвращаемости или накопительного итога. Когортный анализ является ретроспективным, смотрящим назад на аудиторию с использованием присущих ей атрибутов или атрибутов, характеризующих ее поведение. С помощью такого анализа можно обнаружить интересные и очень полезные корреляции. Однако корреляция не означает причинно-следственную связь. Для определения действительных причинно-следственных связей используются рандомизированные эксперименты. В *гл. 7* мы подробно рассмотрим анализ экспериментов.

Но прежде чем мы перейдем к экспериментам, нам нужно разобраться еще с несколькими видами анализа. В следующей главе мы рассмотрим текстовый анализ. Методы текстового анализа часто применяются в других анализах, и сам по себе он достаточно интересный.

# ГЛАВА 5
# Текстовый анализ

В предыдущих двух главах мы рассмотрели, как работать с датами и числами при выполнении анализа временных рядов и когортного анализа. Но наборы данных очень часто содержат более объемную информацию, чем просто числовые значения и связанные с ними временные метки. Любые символьные поля, хранящие как качественные признаки, так и произвольный текст, могут содержать потенциально интересную информацию. Кроме того, что базы данных отлично справляются с числовыми вычислениями, такими как подсчет количества, суммирование и усреднение, они также хорошо выполняют операции над текстовыми данными.

Я начну эту главу с рассмотрения типов задач текстового анализа, для решения которых можно использовать SQL и для которых лучше выбрать другой язык программирования. Далее я расскажу о наборе данных о наблюдениях неопознанных летающих объектов (НЛО). Затем мы перейдем к написанию кода, рассмотрим характеристики текста и его профилирование, разбор текста, выполнение различных преобразований, формирование нового текста с помощью конкатенации и, наконец, поиск соответствий, в том числе с помощью регулярных выражений.

## 5.1. Текстовый анализ и SQL

В огромном объеме данных, который генерируется каждый день, значительная их часть представляет собой текст: слова, предложения, абзацы и даже более длинные документы. Текстовые данные, используемые для анализа, могут поступать из различных источников, включая описания, создаваемые людьми или приложениями, лог-файлы, заявки в службу поддержки, опросы клиентов, сообщения в социальных сетях или новостные ленты. Текст в базах данных может быть структурированным (когда данные разложены по различным полям таблицы и имеют смысловые значения) или полуструктурированным (когда данные находятся в отдельных столбцах, но могут нуждаться в синтаксическом разборе или очистке, чтобы использоваться в анализе), но чаще всего текст является неструктурированным (когда длинные `VARCHAR` или `BLOB` поля содержат строки произвольной длины, которые требуют тщательной структуризации перед дальнейшим анализом). К счастью, в SQL есть ряд полезных функций, которые можно комбинировать для выполнения задач по структурированию и анализу текста.

## Что такое текстовый анализ

Текстовый анализ — это процесс извлечения значений и смыслов из текстовых данных. Выделяют две большие категории анализа текста по тому, будет ли результат качественным или количественным. *Качественный анализ*, который также можно назвать *текстуальным анализом*, направлен на осознание и понимание смыслов одного или нескольких текстов, часто с использованием других сведений или уникальных предположений. Эту работу часто выполняют журналисты, историки и UX-исследователи. *Количественный анализ* текста также направлен на получение информации из текстовых данных, но его результат оценивается количественно. Целью являются категоризация и извлечение данных, и в рамках анализа обычно определяется количество или частота появления, которые нередко меняются с течением времени. SQL гораздо больше подходит для количественного анализа, поэтому именно ему и посвящена эта глава. Если у вас есть возможность пообщаться с коллегами, которые специализируются на первом типе анализа текста, воспользуйтесь их опытом. Сочетание качественного и количественного анализов — отличный способ получить новые неожиданные выводы и убедить сомневающихся коллег.

Текстовый анализ включает в себя несколько целей или стратегий. Первая цель — это извлечение данных из текста, когда внутри текста необходимо найти полезный фрагмент. Второй целью является категоризация, когда информация извлекается из текстовых данных, чтобы присвоить теги или категории записям в базе данных. Еще одна цель — анализ настроений, чтобы оценить настрой или намерение автора по шкале от негативного до позитивного.

Хотя текстовый анализ существует уже давно, интерес и разработки в этой области значительно выросли с появлением машинного обучения и увеличением вычислительных мощностей, которые необходимы для работы с большими объемами текстовых данных. *Обработка естественного языка* (Natural language processing, NLP) достигла огромных успехов в распознавании, классификации и даже генерации совершенно новых текстовых данных. Человеческий язык очень сложен: различные языки и диалекты, грамматика и сленг, не говоря уже о тысячах слов, которые имеют скрытый смысл или исподволь меняют смысл соседних слов. Как мы увидим, SQL хорошо подходит для некоторых видов текстового анализа, но для других, более сложных задач нужно использовать другие языки и инструменты.

## Как можно использовать SQL для текстового анализа

Существует ряд веских причин для использования SQL при анализе текста. Одна из самых очевидных — это то, что данные уже находятся в базе. Современные базы данных обладают большой вычислительной мощностью, которую можно использовать не только для решения тех задач, которые мы обсуждали ранее, но и для работы с текстами. Сохранение текстовых данных в отдельный файл для анализа с помощью других инструментов отнимает много времени, поэтому в том, чтобы вы-

полнить как можно больше работы с помощью SQL в базе данных, есть свои преимущества.

Если данные еще не сохранены в базу, то для относительно больших наборов данных может быть целесообразно поместить их все-таки в базу данных. Для выполнения преобразований над множеством записей эффективнее использовать базы данных, чем электронные таблицы типа Excel. SQL более надежен, чем электронные таблицы, с точки зрения внесения лишних ошибок в данные, т. к. не требует копирования и вставки записей при анализе, и исходные данные остаются нетронутыми. Конечно, данные в базе могут быть изменены с помощью оператора `UPDATE`, но этого нельзя сделать случайно.

SQL также будет хорошим выбором, когда конечной целью является некоторая количественная оценка. Подсчет количества обращений в службу поддержки, содержащих ключевую фразу, или поиск категории в более крупном тексте, которая будет использоваться для группировки записей, — примеры того, когда SQL прекрасно справится с поставленной задачей. SQL также хорош для очистки и структурирования текстовых полей. *Очистка текста* включает в себя удаление лишних символов и пробелов, исправление заглавных букв и стандартизацию значений. *Реорганизация текстовых полей* включает в себя создание новых столбцов из фрагментов, извлеченных или полученных из других текстовых полей, или конструирование новых полей из строк, хранящихся в разных местах. Строковые функции могут быть выложенными или применены к результатам других функций, что позволяет выполнять практически любые манипуляции с текстом.

Код SQL для анализа текста может быть простым или сложным, но он всегда будет опираться на ваши правила. В системе, основанной на правилах, компьютер следует набору правил или инструкций — ни больше, ни меньше. Это можно противопоставить машинному обучению, при котором компьютер адаптируется на основе данных. Правила хороши тем, что с ними легко работать. Они записываются в виде кода, и их можно проверить, чтобы убедиться, что они дают нужный результат. Недостаток правил в том, что они могут стать длинными и сложными, особенно когда приходится обрабатывать много разных случаев. Это также может затруднить их последующее сопровождение. Если структура текста или тип данных, введенных в столбец, изменяются, набор правил тоже необходимо обновить. Не раз я начинала с правила, которое казалось простым оператором `CASE` на 4 или 5 строк, но по мере развития приложения оно разрасталось до 50 или 100 строк. Правила по-прежнему могут быть хорошим подходом, но нужно синхронизировать изменения с командой разработчиков.

Наконец, SQL отлично подходит для случаев, когда вы заранее знаете, что ищете. Существует ряд мощных функций, в том числе и регулярные выражения, которые позволяют вам искать, извлекать или заменять определенные фрагменты текста. «Сколько рецензентов упомянули в своем обзоре “быструю разрядку батареи”?» — это вопрос, на который SQL может помочь вам ответить. С другой стороны, ответить на вопрос «Почему эти клиенты недовольны?» будет не так просто.

## Когда не стоит использовать SQL

По сути SQL позволяет вам использовать мощь базы данных для применения своего набора правил к тексту, чтобы сделать его более удобным для анализа. SQL, безусловно, не единственный инструмент для анализа текста, и есть ряд случаев, для которых он не очень подходит. Полезно знать о них.

Первая категория включает в себя случаи, когда для анализа текста больше подходит человек. Когда набор данных очень маленький или совсем новый, ручная расстановка меток может быть быстрее и информативнее. Кроме того, если цель состоит в том, чтобы просмотреть все записи и составить качественное резюме по ключевым темам, то с этим тоже лучше справится человек.

Вторая категория — когда необходимо находить и извлекать записи, содержащие текстовые строки, с минимальной задержкой. Такие инструменты, как Elasticsearch или Splunk, были разработаны для индексирования строк в подобных случаях. Производительность нередко является проблемой при работе с SQL и базами данных — это одна из основных причин, почему мы обычно стараемся структурировать данные по отдельным столбцам, по которым движку базы данных легче выполнять поиск.

Третья категория включает в себя задачи более широкой темы — обработки естественного языка (NLP), для которых лучше выбрать методы машинного обучения и языки, на которых они реализуются, такие как Python. Анализ настроений, используемый для оценки положительных и отрицательных эмоций в текстах, может быть сделан на SQL только очень упрощенно. Например, можно извлечь из текста слова «любовь» и «ненависть» и использовать их для категоризации записей, но, учитывая диапазон слов, которыми могут быть выражены положительные и отрицательные эмоции, а также все идиомы и скрытые смыслы, практически невозможно создать набор правил, чтобы охватить их все. Для разметки частей речи в тексте, когда слова помечаются как существительное, глагол и т. д., лучше использовать библиотеки Python. Генерация текста — создание совершенно нового текста на основе образцов текстов — это еще один пример, для которого лучше использовать другие инструменты. Мы посмотрим, как можно создать новый текст, соединяя фрагменты данных вместе, но SQL все равно ограничен набором правил и не будет автоматически учиться на новых примерах из набора данных, а также адаптироваться под них.

Теперь, когда мы разобрали множество веских причин для использования SQL при анализе текста, а также случаи, которых следует избегать, давайте посмотрим на набор данных, который мы будем использовать для примеров, прежде чем перейти к самому коду SQL.

# 5.2. Набор данных о наблюдениях НЛО

Для примеров в этой главе мы будем использовать набор данных о наблюдениях НЛО, собранных Национальным центром сообщений о НЛО (National UFO

Reporting Center)[1]. Набор данных состоит из 95 463 сообщений, опубликованных в период с 2006 по 2020 г. Сообщения поступают от частных лиц, которые могут вводить информацию через онлайн-форму на сайте центра.

Вы можете скачать файлы CSV с набором данных из GitHub[2] и загрузить их в базу с помощью SQL-кода. Мы будем работать с таблицей `ufo`, в которой всего два столбца. Первый — это составной столбец под названием `sighting_report`, который содержит информацию о том, когда произошло наблюдение, когда о нем было сообщено и когда оно было опубликовано. Он также содержит метаданные о месте, где это произошло, форме объекта и продолжительности наблюдения. Второй столбец — это текстовое поле с названием `description`, которое содержит полное описание события. На рис. 5.1 показан пример данных из этой таблицы.

| * | sighting_report | description |
|---|---|---|
| 1 | Occurred : 6/24/1980 14:00 (Entered as : 06/24/80 14:00)Reported: 4/8/2008 8:45:08 PM 20:45Posted: 5/16/2008Location: Mount W... | Missing Time: Two PeopleMy mother and I have a long history of UFO sightings/involvement |
| 2 | Occurred : 4/8/2008 02:05 (Entered as : 04/06/08 02:05)Reported: 4/8/2008 6:08:21 PM 18:08Posted: 5/16/2008Location: Ottoville, O... | Bright lights near Ottoville, OhioHeading westward on SR 189 out of Ft. Jennings, Ohio at th |
| 3 | Occurred : 9/11/2001 09:00 (Entered as : 9/11/01 10:00)Reported: 4/8/2008 4:04:27 PM 16:04Posted: 5/16/2008Location: Erie, PASh... | Planes guided into the Trade Centers by UFOs.I can't believe not many people reported this |
| 4 | Occurred : 4/8/2008 21:50 (Entered as : 04/08/08 21:50)Reported: 4/8/2008 2:26:52 PM 14:26Posted: 5/16/2008Location: Leeds (UK/... | two bright lights in sky - Brightened then dimmed - definatly not a planetwo lights in the sky |
| 5 | Occurred : 4/4/2008 02:00 (Entered as : 4/4/08 2:00)Reported: 4/8/2008 12:16:18 PM 12:16Posted: 5/16/2008Location: Frazier Park, ... | The Love of my Life is snoring so I can't sleep, I go into the kitchen and watch the snow fall |
| 6 | Occurred : 4/5/2008 22:25 (Entered as : 04/05/08 22:25)Reported: 4/8/2008 11:40:07 AM 11:40Posted: 5/16/2008Location: Oklahom... | Two hovering orange circles drop third orange circle over oklahoma.My girlfriend and I were |
| 7 | Occurred : 11/15/2005 15:00 (Entered as : 11-15-05 15:00)Reported: 4/8/2008 9:53:25 AM 09:53Posted: 5/16/2008Location: Utah (u... | bright changing light hover and landed behind a mountian. lat 37°49'15.60"N lon 112°43'22. |
| 8 | Occurred : 4/5/2008 22:16 (Entered as : 04/05/08 22:16)Reported: 4/5/2008 11:24:51 PM 23:24Posted: 5/16/2008Location: Graham, ... | At first glance, it looked like a plane, we realized that it wasn't moven, but dancing in place |
| 9 | Occurred : 10/15/1994 14:00 (Entered as : 10/15/1994 14:00)Reported: 4/8/2008 8:32:00 AM 08:32Posted: 5/16/2008Location: Circle... | Silver disc saucer hovers above town, then disappears! was driving my car with three friend |
| 10 | Occurred : 4/15/1973 21:30 (Entered as : 04/15/73 21:30)Reported: 4/8/2008 8:24:26 AM 08:24Posted: 5/16/2008Location: Cold Lak... | Strange Lights Indeed Upon returning home on a beautiful spring evening, in the approximat |
| 11 | Occurred : 4/5/2008 03:15 (Entered as : 04/05/08 3:15)Reported: 4/8/2008 7:05:28 AM 07:05Posted: 5/16/2008Location: Erie, PASha... | Three disk like objects flew over Erie, PA at around 3:15, there were lights and not any noise |
| 12 | Occurred : 10/8/2005 06:15 (Entered as : 10/08/05 6:15)Reported: 4/8/2008 6:24:23 AM 06:24Posted: 5/16/2008Location: Orlando, F... | Well me and my girlfriend were waiting for the bus one morning and I was looking at the sky |
| 13 | Occurred : 1/19/2006 19:00 (Entered as : january 19 19:00)Reported: 4/8/2008 3:50:39 AM 03:50Posted: 5/16/2008Location: Tornanc... | square shaped object over Southern CaliforniaDriving home from work, from the corner of my |
| 14 | Occurred : 4/8/2008 01:30 (Entered as : 04/08/08 01:30)Reported: 4/8/2008 2:41:08 AM 02:41Posted: 5/16/2008Location: Scottsdale,... | Black triangle seen flying silently in the night sky.I saw what looked like the Stealth Bomber |
| 15 | Occurred : 1/1/2008 04:45 (Entered as : 010108 04:45)Reported: 4/8/2008 12:22:32 AM 00:22Posted: 5/16/2008Location: Clarsmore,... | Bright geosynchronus light that appears every morning.On 01/01/08 at approx. 0445, my pa |
| 16 | Occurred : 4/5/2008 08:31 (Entered as : 4-5-08 8:31)Reported: 4/5/2008 10:40:21 PM 22:40Posted: 5/15/2008Location: Shingletown,... | scarryll 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: 4/5/2008 9:37:03 PM 21:37Posted: 5/15/2008Location: Oregon (furs... | roley rectangular object falling from the sky and crash landedA friend of mine and I were ou |
| 18 | Occurred : 4/5/2008 20:15 (Entered as : 04/05/08 20:15)Reported: 4/5/2008 8:53:29 PM 20:53Posted: 5/15/2008Location: Centralia, ... | Sighted over Cooks Hill Road and Joppish Road just west of I-5. Dusk clear sky. Large first |
| 19 | Occurred : 4/5/2008 22:25 (Entered as : 04/05/08 22:25)Reported: 4/5/2008 8:50:32 PM 20:50Posted: 5/15/2008Location: San Marco... | Flying triangle in sky over San Marcos! looked up thinking that I was seeing a plane, but it w |
| 20 | Occurred : 4/5/2008 20:30 (Entered as : 04/05/2008 20:30)Reported: 4/5/2008 8:49:44 PM 20:49Posted: 5/15/2008Location: Des Moi... | Weird fireball that just disappeared without fanfare and followed up by 4 fighters.Say it stres |
| 21 | Occurred : 9/14/1981 20:30 (Entered as : 09/14/81 20:30)Reported: 4/5/2008 8:20:41 PM 20:20Posted: 5/15/2008Location: Lilburn, G... | I was coming home from my friends house,I stopped to rest on the side of the road near a v |
| 22 | Occurred : 3/25/2008 13:00 (Entered as : 03-25-08 13:00)Reported: 4/5/2008 7:34:04 PM 19:34Posted: 5/15/2008Location: Summers... | white square object moving very fastThis thing was unlike anything I have ever seen. Instea |

**Рис. 5.1.** Фрагмент таблицы `ufo`

На примерах я покажу, как разобрать первый столбец на даты и дескрипторы, а также покажу, как выполнить различные виды анализа поля `description`. Если бы я работала с этими данными на постоянной основе, то могла бы рассмотреть возможность создания ETL-конвейера — процесса, который обрабатывает данные одним и тем же способом на регулярной основе и сохраняет полученные структурированные данные в новую таблицу. Однако в примерах, приведенных в этой главе, мы будем работать с исходной таблицей `ufo`.

Давайте перейдем к SQL-коду и начнем с изучения характеристик текста сообщений.

## 5.3. Характеристики текста

Самым гибким типом данных в базе является `VARCHAR`, поскольку в поля этого типа можно сохранить практически любые данные. В результате текстовые данные в базах данных могут быть самых разных форматов и размеров. Как и в случае с дру-

[1] **http://www.nuforc.org/**

[2] **https://github.com/cathytanimura/sql_book/tree/master/Chapter 5: Text Analysis**

гими наборами данных, первым нашим шагом будет профилирование и определение характеристик данных. После этого мы можем разработать план действий по очистке и разбору текста, которые могут потребоваться для анализа.

Для начала давайте найдем количество символов в каждом текстовом значении, что можно сделать с помощью функции `length` (или `len` в некоторых базах данных). Эта функция принимает в качестве аргумента строку или символьное поле, как в похожих функциях в других языках и в электронных таблицах:

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

length
------
13
```

Мы можем получить распределение длин строк в поле `sighting_report`, чтобы получить представление о типичной длине текста и о том, есть ли экстремальные выбросы, которые, возможно, придется обрабатывать особым образом:

```
SELECT length(sighting_report), count(*) as records
FROM ufo
GROUP BY 1
ORDER BY 1
;

length  records
------  -------
90      1
91      4
92      8
...     ...
```

На рис. 5.2 видно, что большинство записей имеют длину примерно от 150 до 180 символов, и лишь немногие — менее 140 или более 200 символов. Длина второго поля `description` варьируется от 5 до 64 291 символов. Мы можем предположить, что в этом поле гораздо больше разнообразного текста, даже без проведения дополнительного профилирования.

Давайте посмотрим на несколько выборочных значений из столбца `sighting_report`. Вы можете просмотреть сотню или больше строк, чтобы ознакомиться с их содержанием, но все они будут выглядеть примерно так же, и эти три строки являются репрезентативными для этого столбца:

```
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: Light-
Duration: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: SphereDura-
tion: 5 seconds
```

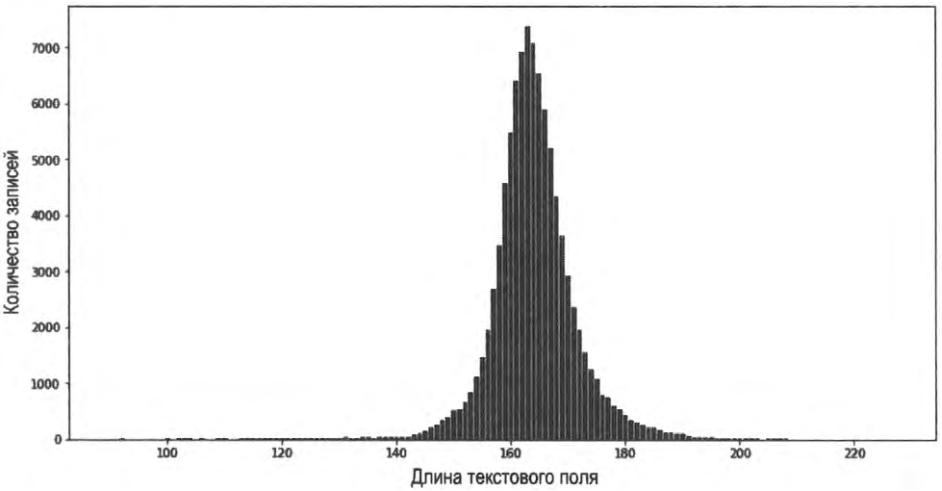


**Рис. 5.2.** Распределение длин строк в поле `sighting_report` в таблице `ufo`

Эти данные я бы назвала полуструктурированными или переполненными. Их нельзя использовать в анализе, но в них явно хранятся отдельные фрагменты полезной информации, и строки составлены по определенному шаблону. Например, в каждой строке есть слова `'Occurred'` (произошло), за которым следует что-то похожее на временную метку[3], `'Location'` (место), после которого идет название населенного пункта, и `'Duration'` (продолжительность), после которого указан временной интервал.

Данные могут оказаться в таких переполненных полях по разным причинам, но есть две самые распространенные, которые я знаю. Первая — когда в исходной системе или приложении не хватает полей для хранения всех необходимых атрибутов, поэтому в одно и то же поле сохраняют несколько атрибутов. Другая возможная причина — когда в приложении данные записываются в формате JSON из-за разреженных атрибутов или частого добавления новых атрибутов. Хотя оба сценария далеки от идеала с точки зрения анализа, но при наличии согласованной структуры мы можем их обработать с помощью SQL.

[3] Поскольку набор данных был создан в США, даты записаны в американском формате `MM/DD/YYYY`. В Европе используется формат `DD/MM/YYYY`. Всегда стоит проверять источник и корректировать формат дат при необходимости.

Далее нам нужно сделать это поле более удобным для использования, разобрав его на несколько новых полей, каждое из которых будет содержать один фрагмент информации. Шаги в этом процессе следующие:

- решить, какое поле (или поля) нужно на выходе;
- применить функции разбора;
- выполнить нужные преобразования, включая преобразование типов данных;
- проверить результаты применения разбора ко всему набору данных, т. к. часто можно встретить записи, которые не соответствуют шаблону;
- повторять эти шаги до тех пор, пока данные не будут разобраны по всем нужным столбцам и в нужных форматах.

Мы выделим из `sighting_report` следующие новые поля: `occurred`, `entered_as`, `reported`, `posted`, `location`, `shape` и `duration`. Далее мы изучим функции для разбора текста и поработаем над структурированием нашего набора данных.

## 5.4. Разбор текста

Разбор текста (parsing) с помощью SQL — это процесс извлечения фрагментов текстового значения, чтобы сделать данные более полезными для анализа. При разборе из данных выделяется та часть, которая нам нужна, и «все остальное», хотя обычно наш код возвращает только ту часть, с которой мы будем работать дальше.

Простейшие функции возвращают фиксированное количество символов с начала или с конца строки. Функция `left` возвращает символы слева или с начала строки, а функция `right` — справа или с конца строки. В остальном они работают одинаково, принимая в качестве первого аргумента текстовое значение, а в качестве второго — количество символов. Оба аргумента могут быть полем базы данных или выражением, что позволяет получать результаты динамически:

```
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
```

В наборе данных о НЛО мы можем получить первое слово `'Occurred'` с помощью функции `left`:

```
SELECT left(sighting_report,8) as left_digits
,count(*)
FROM ufo
GROUP BY 1
;
```

```
left_digits  count
-----------  -----
Occurred     95463
```

Мы можем убедиться, что все значения в этом столбце начинаются с этого слова, что является хорошей новостью, поскольку по крайней мере эта часть шаблона верна для всех значений. Однако на самом деле нам нужны значения того, что произошло, а не само слово `'Occurred'`, поэтому давайте попробуем еще раз. В первой записи из тех, которые я приводил выше в качестве примера, временная метка после `'Occurred'` заканчивается на 25-м символе (слева). Чтобы исключить слово `'Occurred'` и получить только фактическую временную метку, мы можем вернуть крайние правые 14 символов с помощью функции `right`. Обратите внимание, что функции `right` и `left` здесь вложенные — первый аргумент функции `right` является результатом выполнения функции `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
...
```

Хотя для первой записи возвращается правильный результат, но это, к сожалению, не работает для строк с двузначными значениями месяца или дня. Мы можем увеличить количество символов, возвращаемых функциями `left` и `right`, но тогда результат будет включать лишние символы для первой записи.

Функции `left` и `right` удобны для извлечения подстроки фиксированной длины, как в случае со словом `'Occurred'`, но для более сложных шаблонов пригодится функция `split_part`. Эта функция делит строку на части на основе разделителя и возвращает указанную вами часть. *Разделитель* — это один или несколько символов, которые используются для указания границ между частями текста или других данных. Запятая и табуляция, вероятно, являются самыми распространенными разделителями, поскольку они используются в текстовых файлах (с расширениями **csv**, **tsv** или **txt**) для обозначения начала и конца столбцов. Однако в качестве разделителя можно указать любую последовательность символов, необходимую для нашего разбора. Функция имеет следующий вид:

```
split_part(string_or_field_name, delimiter, index)
```

Здесь *index* — это позиция возвращаемой части строки относительно разделителей. Так, при `index = 1` возвращается весь текст слева от первого разделителя, при `index = 2` возвращается текст между первым и вторым разделителем (или весь текст

справа от разделителя, если он встретился в тексте только один раз) и т. д. Нумерация начинается с единицы, а значения должны быть целыми положительными числами:

```
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
```

В MySQL вместо `split_part` используется функция `substring_index`. Microsoft SQL Server вообще не имеет аналогичной функции.

Обратите внимание, что все пробелы в тексте будут сохранены, если они не указаны как часть разделителя. Давайте посмотрим, как можно разобрать столбец `sighting_report` на части. Напомню, как выглядят значения столбца:

```
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: LightDura-
tion:15 minutes
```

Значение, которое мы хотим вернуть нашим запросом, — это фрагмент текст между `'Occurred : '` и `' (Entered'`. То есть нам нужна строка `'6/3/2014 23:00'`. Если внимательно посмотреть, то можно заметить, что `'Occurred : '` и `' (Entered'` встречаются только один раз. Двоеточие `':'` используется несколько раз: для отделения текстовой метки от ее значения и в середине временных меток. Это может сильно затруднить разбор с использованием двоеточия в качестве разделителя. Круглая открывающая скобка появляется только один раз. У нас есть несколько вариантов того, что можно указать в качестве разделителя, и получить либо более длинные значения на выходе, либо только необходимые части при более точном разделении строки. Я склоняюсь скорее к разделению на длинные строки, чтобы можно было проверить, что возвращается именно тот фрагмент, который мне нужен, но это на самом деле зависит от ситуации.

Сначала разделим `sighting_report` по `'Occurred : '` и проверим результат:

```
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
```

Мы успешно удалили первую текстовую метку, но у нас осталось много лишнего текста. Давайте проверим, что получится, если мы разделим с помощью `' (Entered'`:

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

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

Это уже ближе, но все равно осталась текстовая метка. К счастью, мы можем использовать вложенные функции `split_part`, которые и вернут нам нужное значение даты и времени:

```
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
```

Теперь результат содержит нужные значения. Если просмотреть несколько дополнительных строк, то можно увидеть, что двузначные значения дня и месяца возвращаются надлежащим образом, как и даты, записанные без времени. Но оказывается, что в некоторых записях отсутствует текст `'Entered as'`, и текстовая метка `'Reported'` отмечает конец нужного фрагмента, поэтому для обработки таких записей требуется добавить еще одно разбиение:

```
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
...
```

Наиболее часто встречающиеся значения occurred, определенные с помощью SQL-кода, показаны на рис. 5.3.

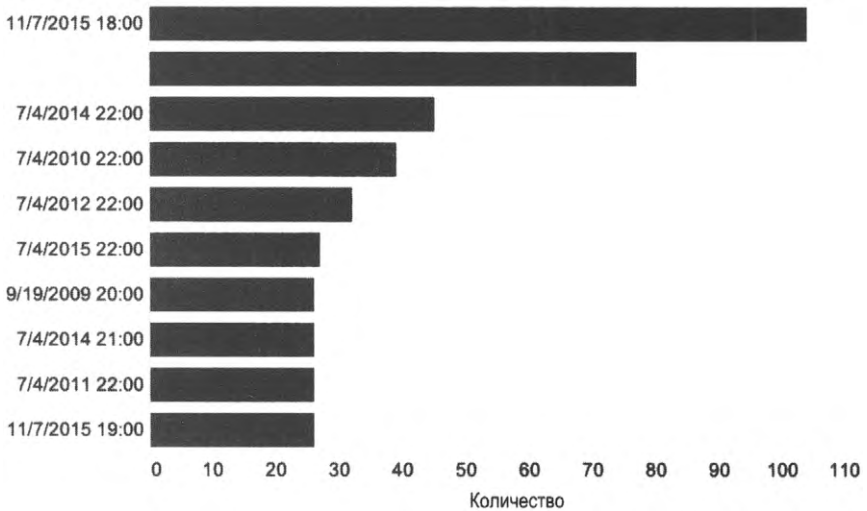


**Рис. 5.3.** Наиболее часто встречающиеся значения `occurred`, полученные из `sighting_report`

Найти функции, которые бы корректно обрабатывали все текстовые значения в наборе данных, — одно из самых сложных при разборе текста. Часто требуется сделать несколько раундов проб и ошибок, каждый раз выполняя профилирование результатов, чтобы найти правильное решение.

Следующим нашим шагом будет составление таких же правил разбора текста для извлечения других нужных значений, используя начальный и конечный разделители для выделения соответствующей части строки. В окончательном запросе

split_part используется несколько раз, с разными аргументами для каждого значения:

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

Благодаря этому SQL-разбору данные теперь имеют гораздо более структурированный и удобный для использования формат. Однако прежде чем мы закончим с ними, необходимо выполнить несколько преобразований, которые немного очистят данные. Далее мы рассмотрим разные функции преобразования текста.

# 5.5. Преобразование текста

Преобразование текста делают, чтобы каким-то образом изменить строковое значение. В *гл. 3* мы рассмотрели несколько функций преобразования дат и временных меток. В SQL есть набор строковых функций, которые по-разному преобразуют строковые значения. Они удобны для работы со значениями, полученными в результате разбора текста, а также с любыми текстовыми данными в базе, которые необходимо изменить или очистить для анализа.

Наиболее часто используемые преобразования — это изменение регистра. Функция `upper` переводит все буквы в верхний регистр (в заглавные буквы), а функция `lower` переводит все буквы в нижний регистр (в строчные буквы). Например:

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

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

SELECT lower('Some sample text');

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

Эти функции нужны для стандартизации значений, которые могли быть напечатаны по-разному. Например, любой человек поймет, что `'California'`, `'caLiforNia'` и `'CALIFORNIA'` означают один и тот же штат, но база данных будет рассматривать их как разные значения. Если бы мы подсчитали количество случаев наблюдения НЛО по штатам, не исправляя такие значения, то в итоге получили бы три записи для Калифорнии, что привело бы к неправильным результатам анализа. Преобразование названий штатов целиком в заглавные или в строчные буквы решит эту проблему. Некоторые базы данных, включая Postgres, имеют функцию `initcap`, которая делает заглавной первую букву каждого слова в строке. Ее удобно использовать для имен собственных, например для названий штатов:

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

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

Поле `shape`, которое мы получили в результате разбора, содержится одно значение, написанное заглавными буквами, — `'TRIANGULAR'`. Чтобы исправить его и привести к

общему виду, как у других значений, у которых заглавной является только первая буква, применим функцию `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
...         ...
```

Количество наблюдений объектов каждой формы показано на рис. 5.4. Наиболее распространенной формой является свет (`'Light'`), далее идут круг и треугольник. В некоторых наблюдениях не указана форма объекта, поэтому на графике присутствует пустое значение.

Еще одной полезной функцией преобразования является функция `trim`, которая удаляет пробелы в начале и в конце строки. Лишние пробельные символы являются частой проблемой при разборе длинных строк, при вводе данных вручную или при копировании данных из одного приложения в другое. В качестве примера мы можем удалить лишние пробелы в следующей строке с помощью функции `trim`:

```
SELECT trim('  California    ');

trim
----------
California
```

У функции `trim` есть несколько дополнительных параметров, которые делают ее очень гибкой для решения различных задач по очистке данных. Во-первых, она может удалять символы только в начале строки или только в конце строки, или в начале и в конце. По умолчанию символы удаляются с обеих сторон, но можно специально указать `leading` (начальные) или `trailing` (завершающие) символы. Кроме того, `trim` может удалять любые символы, а не только пробелы.

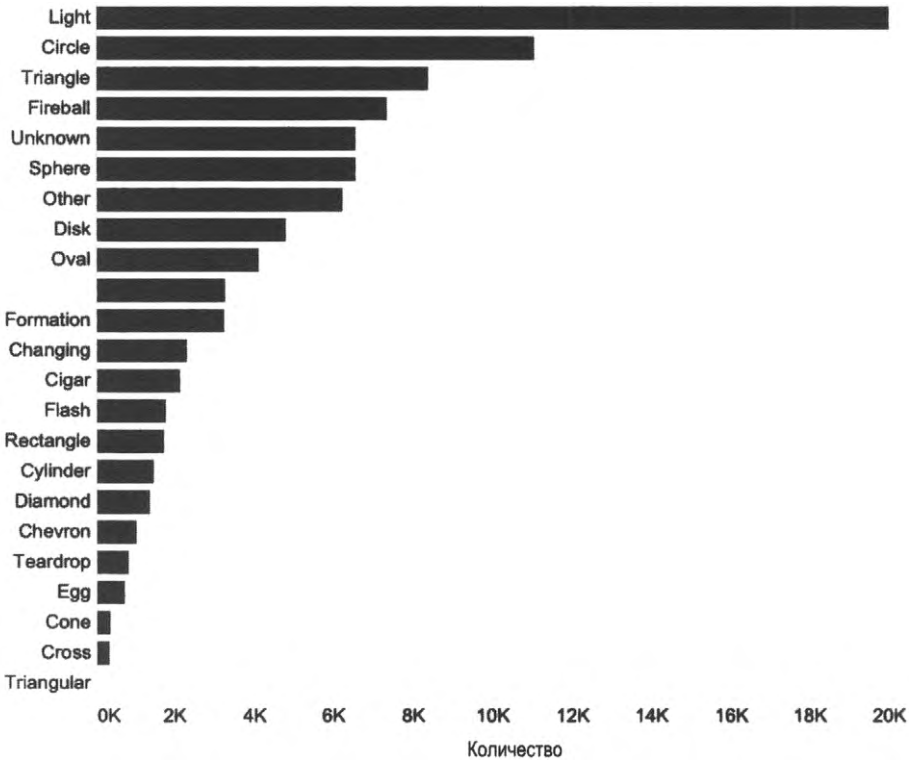


**Рис. 5.4.** Частота упоминания форм объектов в наблюдениях НЛО

Так, например, если приложение случайно поместило знак доллара `'$'` в начале названия каждого штата, мы можем удалить его с помощью `trim`:

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

trim
----------
California
```

Несколько значений в поле `duration` содержат начальные пробелы, поэтому использование `trim` даст более чистый результат:

```
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)
```

На рис. 5.5 показано количество наблюдений с наиболее распространенными продолжительностями. Чаще всего регистрируются наблюдения продолжительностью от 1 до 10 минут. В некоторых случаях продолжительность наблюдений не указана, поэтому на графике отображено пустое значение.

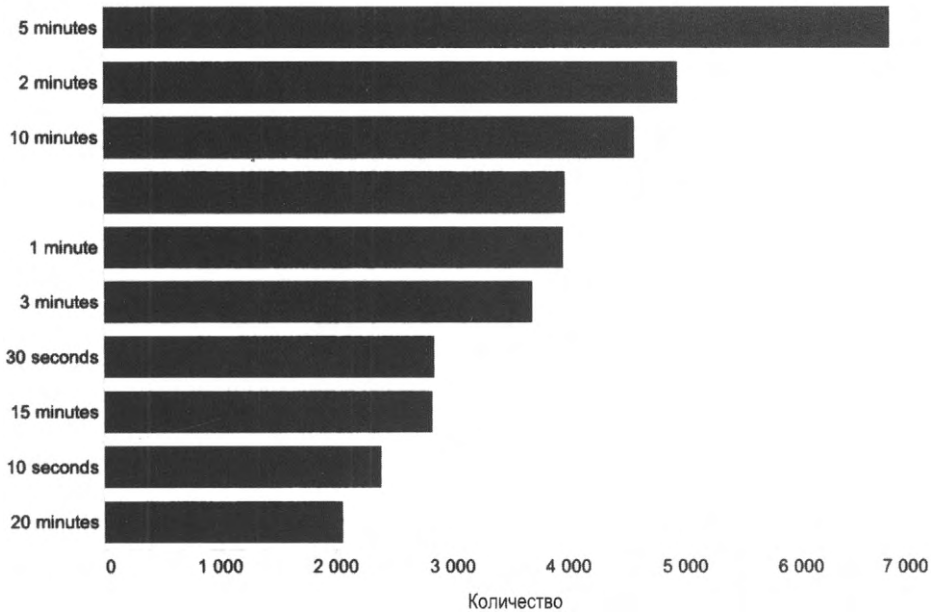


**Рис. 5.5.** Наиболее часто встречающиеся продолжительности наблюдений НЛО

Следующее, что мы сделаем, — выполним преобразование типов данных. Такое преобразование мы уже рассматривали в *разд. 2.5*, и оно нужно для того, чтобы результаты разбора имели необходимые типы данных. В нашем случае есть два поля, в которых должны храниться временные метки: `occurred` и `reported`, а столбец `posted` должен иметь тип даты. Типы данных можно изменить с помощью приведения, используя либо оператор `::`, либо функцию `cast`. Однако если формат даты в настройках вашей базы данных не совпадает с форматом даты в тексте, то для того, чтобы код правильно распознавал даты в американском формате `MM/DD/YYYY`, надо использовать функции `to_timestamp` и `to_date` и явно указать строку формата. Значения `entered_as`, `location`, `shape` и `duration` мы оставим как `VARCHAR`:

```
SELECT to_timestamp(occurred, 'MM/DD/YYYY HH24:MI') as occurred
,to_timestamp(reported, 'MM/DD/YYYY HH24:MI') as reported
,to_date(posted, 'MM/DD/YYYY') as posted
FROM
(
```

```
    SELECT
    split_part(
      split_part(
        split_part(sighting_report,' (Entered',1)
        ,'Occurred : ',2)
      ,'Reported',1)
      as occurred
    ,split_part(
      split_part(
       split_part(
        split_part(sighting_report,'Post',1)
        ,'Reported: ',2)
        ,' AM',1),' PM',1)
       as reported
     ,split_part(
       split_part(sighting_report,'Location',1)
       ,'Posted: ',2)
       as posted
     FROM ufo
) a
;

occurred             reported             posted
-------------------  -------------------  ----------
2015-05-24 19:30:00  2015-05-25 10:07:21  2015-05-29
2015-05-24 22:40:00  2015-05-25 09:09:09  2015-05-29
2015-05-24 22:30:00  2015-05-24 10:49:43  2015-05-29
...                  ...                  ...
```

Для небольшой выборки извлеченные данные преобразовываются в новый формат. Обратите внимание, что к временной метке добавляются секунды, хотя в исходной строке секунд не было. Однако при применении этих преобразований ко всему набору данных возникают проблемы. Некоторые записи вообще не имеют значений, вместо даты пустая строка, а в некоторых указано только время, но нет никакой даты. Хотя кажется, что пустая строка и `null` обозначают одно и то же — отсутствие информации, но базы данных обрабатывают их по-разному. Пустая строка является все-таки строкой и не может быть преобразована в другой тип данных. Установка всех недопустимых значений в `null` с помощью оператора `CASE` позволяет правильно выполнить преобразование типов для всего набора данных. Поскольку мы знаем, что даты должны содержать не менее восьми символов (четыре цифры для года, одна или две цифры для месяца и дня и два символа-разделителя,

- или /), то, как вариант, мы можем вместо каждой строки с длиной меньше 8 возвращать `null`:

```
SELECT
case when occurred = '' then null
     when length(occurred) < 8 then null
     else to_timestamp(occurred, 'MM/DD/YYYY HH24:MI')
     end as occurred
,case when length(reported) < 8 then null
      else to_timestamp(reported, 'MM/DD/YYYY HH24:MI')
      end as reported
,case when posted = '' then null
      else to_date(posted, 'MM/DD/YYYY')
      end as posted
FROM
(
    SELECT
    split_part(
      split_part(
        split_part(sighting_report,'(Entered',1)
        ,'Occurred : ',2)
      ,'Reported',1) as occurred
    ,split_part(
      split_part(
       split_part(
        split_part(sighting_report,'Post',1)
        ,'Reported: ',2)
       ,' AM',1)
      ,' PM',1) as reported
    ,split_part(
       split_part(sighting_report,'Location',1)
       ,'Posted: ',2) as posted
    FROM ufo
) a
;

occurred            reported            posted
------------------  ------------------  ----------
1991-10-01 14:00:00  2018-03-06 08:54:22  2018-03-08
2018-03-04 19:07:00  2018-03-06 07:05:12  2018-03-08
2017-10-16 21:42:00  2018-03-06 05:09:47  2018-03-08
...                 ...                 ...
```

Последнее преобразование, о котором я расскажу в этом разделе, — это функция `replace`. Иногда в тексте встречаются символ, слово или фраза, которую нам нужно заменить на другую строку или полностью удалить. Для этой задачи и пригодится функция `replace`. Она принимает три аргумента — текст или поле, строку, которую нужно найти, и строку, на которую нужно заменить:

```
replace(string_or_field, string_to_find, string_to_substitute)
```

Так, например, если мы хотим заменить упоминания `'unidentified flying objects'` (неопознанные летающие объекты) на более короткое `'UFOs'` (НЛО), мы можем использовать функцию `replace`:

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


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

Эта функция найдет и заменит каждое вхождение строки из второго аргумента. В качестве третьего аргумента можно указать пустую строку `''`, и таким способом можно удалить все ненужные подстроки. Как и другие строковые функции, `replace` может быть вложенной, при этом результат одной замены может быть использован как аргумент в другой замене.

В нашем разобранном наборе данных некоторые значения местоположения `location` включают в себя, кроме населенного пункта, уточнения, что наблюдение произошло `'near'` (рядом с), `'close to'` (недалеко от) или `'outside of'` (за пределами). Мы можем использовать функцию `replace` для стандартизации всех этих значений до `'near'`:

```
SELECT location
,replace(replace(location,'close to','near')
         ,'outside of','near') as location_clean
FROM
(
    SELECT split_part(split_part(sighting_report,'Shape',1)
                      ,'Location: ',2) as location
    FROM ufo
) a
;

location                    location_clean
--------------------------  --------------------
Tombstone (outside of), AZ  Tombstone (near), AZ
Terrell (close to), TX      Terrell (near), TX
Tehachapie (outside of), CA Tehachapie (near), CA
...                         ...
```

Десять самых лучших мест для наблюдения за НЛО приведены на рис. 5.6.

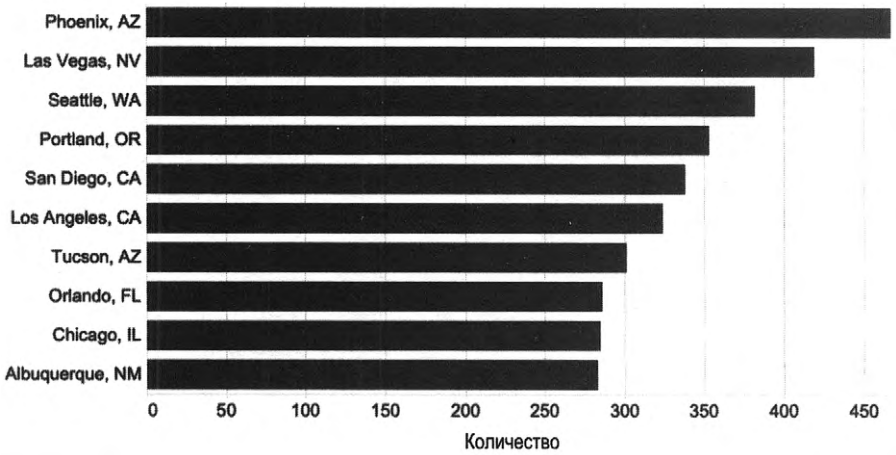


**Рис. 5.6.** Места, где чаще всего наблюдают НЛО

Теперь мы разобрали и очистили все значения из поля `sighting_report` в отдельные типизированные столбцы. Окончательный код может выглядеть так:

```
SELECT
case when occurred = '' then null
     when length(occurred) < 8 then null
     else to_timestamp(occurred, 'MM/DD/YYYY HH24:MI')
     end as occurred
,entered_as
,case when length(reported) < 8 then null
      else to_timestamp(reported, 'MM/DD/YYYY HH24:MI')
      end as reported
,case when posted = '' then null
      else to_date(posted, 'MM/DD/YYYY')
      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
...      ...        ...      ...     ...        ...      ...
```

Этот SQL-код можно повторно использовать в других запросах или для копирования данных об НЛО в новую таблицу с очищенными данными. Как вариант, этот запрос можно превратить в представление (view) или в общее табличное выражение (CTE) для повторного использования. В *гл. 8* обе эти конструкции будут рассмотрены более подробно.

Мы уже разобрались, как применять строковые функции для разбора и преобразования текстовых данных, имеющих определенную структуру, чтобы улучшить их анализ. Далее мы рассмотрим второе поле в наборе данных о наблюдениях НЛО — поле `description` с произвольным текстом — и узнаем, как использовать функции SQL для поиска по тексту.

# 5.6. Поиск в текстовых данных

Разбор и преобразование — это обычные операции, применяемые к текстовым данным для подготовки их к анализу. Еще одна распространенная операция, выполняемая над текстовыми данными, — поиск по тексту. Это может пригодиться для фильтрации результатов, категоризации записей или замены строк альтернативными значениями.

## Подстановочные знаки: *LIKE, ILIKE*

В SQL есть несколько способов для выполнения поиска по шаблону. Оператор `LIKE` ищет указанный шаблон внутри строки. Для того чтобы оператор `LIKE` искал не просто точное совпадение, нужно добавить подстановочные знаки в начало, конец или середину шаблона. Подстановочный знак `'%'` соответствует любому количеству символов (ноль или более), а знак `'_'` (подчеркивание) — ровно одному символу:

```
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
```

Если нужно найти в строке сами символы `'%'` или `'_'`, перед ними надо поставить символ экранирования `'\'` (обратная косая черта).

Оператор `LIKE` можно использовать в нескольких разделах SQL-запроса. Его можно использовать для фильтрации записей в разделе `WHERE`. Например, некоторые авторы сообщений о НЛО указали, что в то время они были с супругой, и поэтому мы можем найти, в скольких сообщениях упоминается слово `'wife'` (жена). Поскольку это слово может находиться в любом месте описания, мы добавим подстановочный знак `'%'` до и после слова `'wife'`:

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

count
-----
6231
```

Более чем в 6000 отчетов упоминается жена. Однако этот запрос ищет только совпадения, набранные в нижнем регистре. Что делать, если некоторые авторы написали 'Wife' с заглавной буквы или даже напечатали 'WIFE' с нажатой клавишей <Caps Lock>? Есть два способа сделать поиск нечувствительным к регистру. Один из них — преобразовать поле, по которому выполняется поиск, с помощью функции upper или lower из предыдущего раздела, что сделает поиск нечувствительным к регистру, поскольку все символы в строке будут либо заглавными, либо строчными:

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

count
-----
6439
```

Того же самого можно добиться с помощью оператора ILIKE, который представляет собой оператор LIKE, но не чувствительный к регистру. Недостатком является то, что он доступен не во всех базах данных. В частности, MySQL и Microsoft SQL Server не поддерживают ILIKE. Однако это хороший компактный вариант синтаксиса, если в вашей базе он есть:

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

count
-----
6439
```

Любой из этих операторов LIKE и ILIKE может быть инвертирован с помощью дополнения NOT. Например, чтобы найти записи, в которых не упоминается жена, мы можем использовать NOT LIKE:

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

count
-----
89024
```

Фильтрация по нескольким строкам возможна с помощью AND и OR:

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

count
-----
10571
```

Будьте осторожны и используйте круглые скобки для контроля за порядком выполнения операций при одновременном использовании AND и OR, иначе вы можете получить неожиданные результаты. Например, следующие запросы возвращают разные результаты, поскольку во втором запросе изменен порядок выполнения операторов с помощью круглых скобок:

```
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
```

Помимо фильтрации в разделе WHERE или в условии JOIN оператор LIKE можно использовать в операторе SELECT для категоризации или агрегации определенных записей. Начнем с категоризации. Оператор LIKE можно использовать в операторе

CASE для группировки записей. В некоторых описаниях упоминается то, чем занимался наблюдатель до или во время наблюдения, например вел автомобиль или гулял. Мы можем выяснить, сколько описаний содержат такие упоминания, используя оператор CASE вместе с оператором 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
```

Самым распространенным видом деятельности было вождение автомобиля (`'driving'`), и не так много людей занимались в это время плаваньем (`'swimming'`) или ехали на велосипеде (`'cycling'`). Это, пожалуй, и неудивительно, поскольку этими видами деятельности люди просто занимаются реже, чем водят автомобили.

Хотя функции, выполняющие разбор и преобразование текста, можно использовать в условиях JOIN, при выполнении таких запросов часто возникает проблема с производительностью базы данных. Лучше выполнить разбор и/или преобразование текста в подзапросе, а затем уже использовать готовый результат в условии оператора JOIN.

Обратите внимание, что этот оператор CASE помечает каждое описание только одним видом деятельности и проверяет на соответствие каждому шаблону в том порядке, в котором написан оператор. Описание, содержащее и `'driving'`, и `'walking'`, будет помечено как `'driving'`. Это допустимо для многих случаев, но при анализе длинных текстов, например отзывов, комментариев к опросам или заявок в службу

поддержки, нужна возможность помечать записи несколькими категориями. Для таких случаев необходим набор флагов с бинарными или логическими значениями.

Ранее мы уже видели, что LIKE возвращает значение типа BOOLEAN (TRUE или FALSE), и мы можем использовать этот оператор для генерации флагов. В наборе данных в ряде описаний упоминается направление, в котором был обнаружен объект, например север или юг, а в некоторых описаниях упоминается более одного направления. Мы можем пометить каждую запись четырьмя флажками, которые обозначают, упоминается ли в тексте соответствующее направление:

```
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
```

В результате получается матрица значений BOOLEAN, которую можно использовать для определения частоты различных направлений или только одного конкретного направления в описаниях.

Эти комбинации направлений могут быть полезны в некоторых контекстах, особенно при создании наборов данных, которые будут использоваться другими людьми для изучения данных, в BI-инструментах или в инструментах визуализации. Однако иногда нужно обобщить данные и выполнить агрегирование записей, содержащих строковый шаблон. Здесь мы будем считать количество записей, но можно использовать и другие агрегации, такие как `sum` и `avg`, если набор данных содержит другие числовые поля, например показатели продаж:

```
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
```

Теперь у нас есть более компактные результаты для частоты направлений в поле описания, и мы видим, что `'east'` упоминается чаще, чем другие направления. Результаты представлены на рис. 5.7.

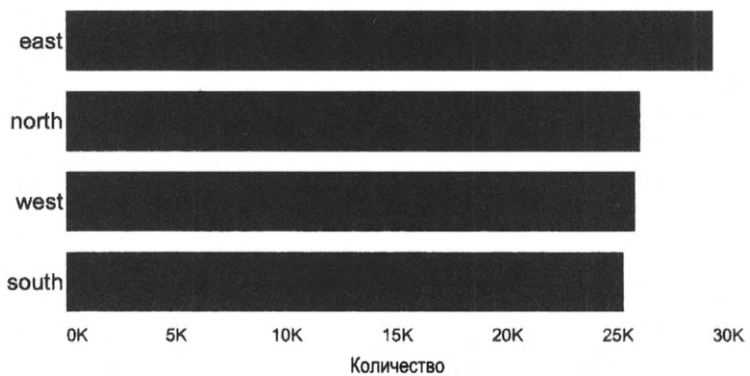


**Рис. 5.7.** Частота упоминания направлений в описаниях наблюдений НЛО

В предыдущем запросе записи, содержащие более одного направления, считаются более одного раза. Однако мы уже не видим, какие комбинации направлений существуют. Если нужно подсчитать количество комбинаций, можно немного усложнить запрос с помощью, например:

```
count(case when description ilike '%east%'
      and description ilike '%north%' then 1 end) as east_north
```

Поиск по шаблонам с помощью операторов LIKE, NOT LIKE и ILIKE является очень гибким и может использоваться в различных разделах SQL-запроса для фильтрации, категоризации и агрегирования данных для разных целей. Эти операторы можно использовать в сочетании с функциями разбора и преобразования текста, которые мы разбирали ранее. Далее я расскажу о проверке на точное соответствие нескольким значениям, а затем опять вернусь к шаблонам, но уже с помощью регулярных выражений.

## Точное соответствие: *IN, NOT IN*

Прежде чем мы перейдем к более сложным шаблонам с помощью регулярных выражений, стоит рассмотреть пару дополнительных операторов, которые могут пригодиться при анализе текста. Хотя они не связаны напрямую с поиском по шаблону, их часто используют в сочетании с LIKE, чтобы получить именно тот набор результатов, который нужен. Это оператор IN и его отрицание NOT IN. Они позволяют вам перечислить список совпадений, что приводит к более компактному коду.

Представим, что нас интересует классификация наблюдений по первому слову в описании description. Мы можем найти первое слово, используя функцию split_part, с символом пробела в качестве разделителя. Многие сообщения начинаются с описания цвета. Возможно, мы захотим отфильтровать записи, чтобы просмотреть сообщения, которые начинаются с названия цвета. Это можно сделать, перечислив каждый цвет с помощью 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...
...          ...
```

Использование оператора IN сделает код более компактным и менее подверженным ошибкам, особенно если в разделе WHERE есть и другие фильтры. Оператор IN принимает список элементов, разделенных запятыми, для поиска точных соответствий. Тип элементов списка должен соответствовать типу данных столбца, по которому ведется поиск. Если тип столбца числовой, в списке должны быть перечислены числа; если тип столбца текстовый, элементы списка должны быть текстовыми:

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

Результаты двух запросов идентичны, а частота показана на рис. 5.8.

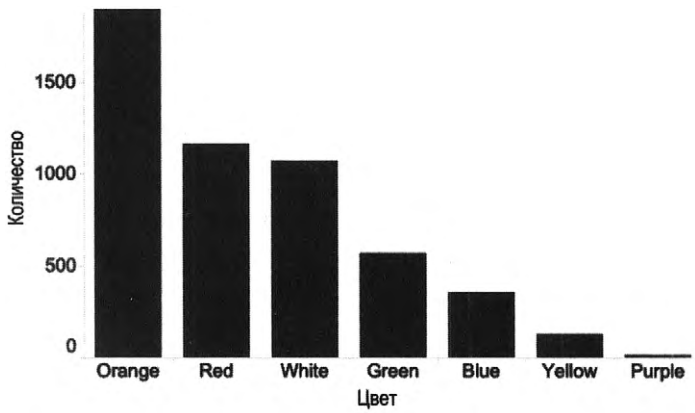


**Рис. 5.8.** Частота использования цветов в качестве первого слова в описаниях наблюдений НЛО

Основное преимущество IN и NOT IN заключается в том, что они делают код более компактным и читаемым. Это может пригодиться для выполнения более сложных категоризаций в операторе SELECT. Представьте, например, что мы хотим категоризировать и подсчитать записи в зависимости от того, чем является первое слово в описании, — это цвет ('Color'), форма ('Shape'), движение ('Motion') или что-то другое ('Other'). Мы можем совместить в одном запросе разбор текста, преобразования, сопоставления с шаблонами и проверку на соответствие спискам 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
```

Конечно, учитывая тематику этого набора данных, для точной категоризации описаний по первому слову, скорее всего, потребуется гораздо больше правил и больше строк кода. SQL позволяет создавать множество сложных выражений для учета разных тонких нюансов в текстовых данных. Далее мы рассмотрим более сложные способы обработки текста с помощью регулярных выражений.

## Регулярные выражения

В SQL существует много способов выполнять поиск по шаблону. Одним из самых сильных методов является использование регулярных выражений. Сказать честно, регулярные выражения пугали даже меня, и я долгое время избегала их использовать, работая аналитиком данных. На крайний случай у меня всегда были рядом коллеги, которые готовы были поделиться фрагментом кода. И только когда я стала работать над большим проектом по анализу текста, то решила, что наконец-то пришло время разобраться с регулярными выражениями.

*Регулярное выражение* (regex) — это последовательность символов, многие из которых имеют специальные значения и определяют шаблон поиска. Основная трудность в изучении регулярных выражений, а также в использовании и сопровождении кода, содержащего их, заключается в том, что их синтаксис не очень интуитивно-понятен. Фрагменты кода не читаются, как естественный язык или даже как компьютерные языки, например SQL или Python. Однако, зная значения специальных символов, регулярное выражение можно разобрать. Как и в случае со всеми нашими примерами, рекомендую начинать с простого запроса, постепенно усложняя его и проверяя результаты по ходу дела. И всегда оставляйте в коде комментарии, как для других аналитиков, так и для себя.

Regex — это отдельный язык, но он может использоваться только из других языков. Например, регулярные выражения могут вызываться из Java, Python и SQL, но нет самостоятельного способа программирования с помощью regex. Все основные базы данных имеют некоторую реализацию regex. Синтаксис не всегда одинаковый, но, как и в случае с другими функциями, если вы имеете представление о возможностях SQL, то найти адаптацию для вашей базы достаточно просто.

Полный разбор синтаксиса и всех способов использования regex выходит за рамки этой книги, но я расскажу вам достаточно, чтобы вы смогли начать работать с regex и решать некоторые часто возникающие задачи на SQL. Для более глубокого знакомства с регулярными выражениями вы можете почитать книгу Бена Форты «Изучаем регулярные выражения» (М.: Вильямс, 2019). А я начну с описания способов использования regex в SQL, затем мы разберем основной синтаксис, после чего перейдем к некоторым примерам того, как regex может быть полезен при анализе сообщений о наблюдениях НЛО.

В операторах SQL regex можно использовать двумя способами: с помощью компараторов POSIX и с помощью функций. POSIX (Portable Operating System Interface, переносимый интерфейс операционных систем) — это набор стандартов IEEE (Institute of Electrical and Electronics Engineers, институт инженеров по электротехнике и радиоэлектронике), но вам не нужно знать подробности, а только то, как использовать компараторы POSIX в коде SQL. Первый компаратор, который мы рассмотрим, — это символ ~ (тильда), который сравнивает два строковых выражения и возвращает `TRUE`, если вторая строка содержится в первой. В качестве простого

примера мы можем проверить, содержится ли в строке 'The data is about UFOs' слово 'data':

```
SELECT 'The data is about UFOs' ~ 'data' as comparison;

comparison
----------
true
```

Возвращаемое значение — типа BOOLEAN (TRUE или FALSE). Обратите внимание, что, несмотря на отсутствие каких-то специальных символов, 'data' здесь является регулярным выражением. Регулярные выражения могут содержать обычные текстовые строки. Этот пример аналогичен использованию оператора LIKE. Компаратор ~ чувствителен к регистру. Чтобы сделать его нечувствительным к регистру, как ILIKE, добавьте к нему символ звездочки ~*:

```
SELECT 'The data is about UFOs' ~* 'DATA' as comparison;

comparison
----------
true
```

Чтобы инвертировать компаратор, поставьте знак ! (восклицательный знак) перед тильдой или комбинацией тильда-звездочка:

```
SELECT 'The data is about UFOs' !~ 'alligators' as comparison;

comparison
----------
true
```

В табл. 5.1 приведены четыре компаратора POSIX.

**Таблица 5.1.** Компараторы POSIX

| Синтаксис | Назначение | Учитывает регистр? |
|---|---|---|
| ~ | Сравнивает два выражения и возвращает TRUE, если первое из них содержит второе. | Да |
| ~* | Сравнивает два выражения и возвращает TRUE, если первое содержит второе. | Нет |
| !~ | Сравнивает два выражения и возвращает FALSE, если первое содержит второе. | Да |
| !~* | Сравнивает два выражения и возвращает FALSE, если первое содержит второе. | Нет |

Теперь, когда мы знаем как использовать regex в нашем SQL, давайте разберемся с некоторыми специальными шаблонами поиска, которые он предлагает. Первый специальный символ regex, который нужно знать, — это . (точка), которая используется вместо любого одиночного символа:

```
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
```

Давайте разберемся с этим поподробнее, чтобы понять, что происходит и как работает regex. В первом сравнении шаблон ищет любой символ, обозначенный точкой, за которым следуют пробел и слово `'data'`. Этот шаблон соответствует подстроке `'e data'` в этом примере, поэтому вернулось `TRUE`. Если это кажется нелогичным, поскольку перед буквой `'e'` и после слова `'data'` есть еще дополнительные символы, помните, что компаратор `~` ищет этот шаблон где-то внутри строки. Во втором сравнении шаблон пытается найти любой символ, за которым следует `'The'`. Поскольку в примере `'The'` является началом строки и перед ним нет никакого символа, вернулось значение `FALSE`.

Чтобы указать несколько символов, используйте специальный символ `*` (звездочка). Это будет обозначать ноль или более символов, аналогично использованию `'%'` в операторе `LIKE`. Такое использование звездочки отличается от ее использования в компараторе `~*`, где она делает его нечувствительным к регистру. Заметьте также, что в регулярных выражениях `'%'` не является подстановочным знаком, а рассматривается как литерный символ, который нужно найти:

```
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
```

Следующие специальные символы, которые необходимо знать, — это `[ ]` (квадратные скобки). Внутри скобок перечисляются символы, и ищется соответствие любому из них. Сами скобки обозначают один символ, даже если в скобках перечислено несколько, но чуть позже мы увидим, как можно искать соответствие более чем одному символу. Одно из применений квадратных скобок — сделать часть шаблона нечувствительной к регистру, заключив заглавную и строчную буквы в квадратные

скобки (здесь нельзя использовать запятую при перечислении, т. к. она будет приниматься за искомый литерал):

```
SELECT 'The data is about UFOs' ~ '[Tt]he' as comparison;

 comparison
 ----------
 true
```

В этом примере шаблон соответствует либо 'the', либо 'The', а поскольку с одного из этих слов начинается предложение, оператор вернул значение TRUE. Это не совсем то же самое, что нечувствительный к регистру компаратор ~*, потому что в этом примере такие варианты, как 'tHe' и 'THE', не соответствуют нашему шаблону:

```
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
```

Другое использование квадратных скобок — это поиск соответствия, включающего в себя одно число из нескольких возможных. Представьте, например, что мы хотим найти любое описание, в котором упоминается '7 minutes', '8 minutes' или '9 minutes'. Этого можно добиться с помощью оператора CASE с несколькими операторами LIKE, но с помощью regex синтаксис будет более компактным:

```
SELECT 'sighting lasted 8 minutes' ~ '[789] minutes' as comparison;

 comparison
 ----------
 true
```

Чтобы найти любое число, мы можем перечислить все цифры в квадратных скобках:

```
[0123456789]
```

Однако в regex можно указать диапазон символов с помощью разделителя - (дефис). Все цифры могут быть обозначены как диапазон [0-9]. Также можно использовать любой меньший диапазон, например [0-3] или [4-9]. Следующий шаблон с

диапазоном эквивалентен предыдущему примеру, в котором были перечислены три числа:

```
SELECT 'sighting lasted 8 minutes' ~ '[7-9] minutes' as comparison;

comparison
----------
true
```

Аналогичным образом можно указывать диапазоны букв. В табл. 5.2 приведены шаблоны диапазонов, которые чаще всего используются при анализе в SQL. Нечисловые и небуквенные символы также можно указывать в квадратных скобках, например `[$%@]`.

**Таблица 5.2.** Шаблоны диапазонов regex

| Шаблон диапазона | Назначение |
|---|---|
| `[0-9]` | Соответствие любой цифре. |
| `[a-z]` | Соответствие любой строчной букве. |
| `[A-Z]` | Соответствие любой заглавной букве. |
| `[A-Za-z0-9]` | Соответствие любой заглавной или строчной букве или любому числу. |
| `[A-z]` | Соответствие любому символу ASCII в диапазоне от A до z; обычно не используется, т. к. в диапазон попадают не только буквы. |

Если требуемый шаблон должен содержать более одного символа из диапазона, можно указать столько диапазонов, сколько нужно, один за другим. Например, мы можем искать трехзначное число, повторив диапазон чисел три раза:

```
SELECT 'driving on 495 south' ~ 'on [0-9][0-9][0-9]' as comparison;

comparison
----------
true
```

Другой способ — использовать один из специальных символов для многократного повторения символа или группы символов. Это может пригодиться, когда вы не знаете точно, сколько раз группа символов может повторяться. Но не забывайте проверять результаты, чтобы убедиться, что вы случайно получили не больше совпадений, чем хотели. Чтобы найти сопоставление один или несколько раз, поставьте символ + (плюс) после диапазона:

```
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
```

В табл. 5.3 приведены другие варианты специальных символов для указания количества повторений. Каждый специальный символ используется сразу после группы символов.

**Таблица 5.3.** Символы regex для многократного сопоставления

| Символ | Назначение |
|---|---|
| + | Соответствие группе символов один или более раз. |
| * | Соответствие группе символов ноль или более раз. |
| ? | Соответствие группе символов ноль или один раз. |
| { } | Соответствие группе символов указанное между фигурными скобками количество раз. Например, {3} соответствует «точно три раза». |
| { , } | Соответствие группе символов любое количество раз в диапазоне, указанном в фигурных скобках через запятую. Например, {3,5} соответствует «от трех до пяти раз». |

Иногда вместо соответствия шаблону мы хотим найти элементы, которые не соответствуют шаблону. Это можно сделать, указав перед шаблоном символ ^ (карет), который служит для отрицания шаблона:

```
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
```

Иногда вам может понадобиться искать по шаблону специальный символ как литерал, поэтому нужно сообщить базе данных, чтобы она не рассматривала этот литерный символ как специальный. Для этого используется символ экранирования, в regex это \ (обратная косая черта) :

```
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
```

Если первом сопоставлении не указать символ экранирования перед `?`, то база данных выдаст ошибку «Invalid regular expression» (недопустимое регулярное выражение) — точная формулировка сообщения об ошибке может отличаться в зависимости от типа базы данных. Во втором сопоставлении, даже если в сравниваемой строке символ `^` сопровождается одной или несколькими цифрами, база данных интерпретирует его в шаблоне `^[0-9]+` как отрицание и будет проверять, что строка не содержит указанных цифр. В третьем сопоставлении символ `^` правильно экранирован, и теперь база данных интерпретирует его как литерный символ, а не специальный.

Текстовые данные обычно содержат пробельные символы. Это могут быть как обычные пробелы между словами, так и невидимые символы табуляции и новой строки. Позже мы увидим, как заменить их с помощью regex, а пока давайте разберемся, как указывать их в шаблоне. Табуляция обозначается как `\t`. Переход на новую строку обозначается как `\r` (возврат каретки) или `\n` (перевод строки), но, в зависимости от операционной системы, иногда они идут вместе: `\r\n`. Поэкспериментируйте с вашими данными, выполнив несколько простых запросов, чтобы проверить, что вернет требуемый результат. Для соответствия любому пробельному символу можно использовать `\s`, но учтите, что сюда относится и символ пробела:

```
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
```

В самих редакторах SQL-запросов могут быть небольшие проблемы с интерпретацией символов перехода на новую строку, если вы вводите текстовое значение непосредственно в окне редактора, и поэтому могут возникать ошибки. В таких случаях попробуйте скопировать текст из любого источника и вставить его в редактор, а не вводить текст вручную. Однако все редакторы SQL-запросов умеют работать с символами перехода на новую строку в текстовых полях, записанных в таблицах базы данных.

Подобно математическим выражениям, круглые скобки `( )` можно использовать для группировки выражений, которые должны рассматриваться вместе. Например, мы можем составить сложный шаблон, который должен повториться несколько раз:

```
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
```

Один и тот же шаблон regex `([0-9]{2}[a-z]){3}` используется для поиска во всех трех строках. Шаблон внутри круглых скобок `[0-9]{2}[a-z]` ищет две цифры, за которыми следует строчная буква. За пределами круглых скобок стоит `{3}`, что указывает на то, что вся группа должна быть повторена три раза. Первая строка соответствует этому шаблону, т. к. содержит строку `'12a34b56c'`. Во второй строке есть первое соответствие группе — две цифры, за которыми следует строчная буква, — `'23a'`, а затем следуют три цифры, а не две цифры и строчная буква — `'23a456'`, поэтому второго совпадения нет. В третьей строке есть совпадение с шаблоном — `99x66y33z`.

Как мы только что видели, в шаблонах regex можно использовать любое количество комбинаций regex-выражений и обычного текста. Помимо того, *что* нужно искать, в шаблонах regex можно использовать указания, *где* искать. Специальный символ `\y` используется для обозначения начала или конца слова (в некоторых базах данных это может быть `\b`). Представьте, например, что нам нужно найти слово `'car'` (автомобиль) в сообщениях о НЛО. Мы могли бы использовать следующий шаблон:

```
SELECT
'I was in my car going south toward my home' ~ 'car' as comparison;
```

```
comparison
----------
true
```

Он найдет соответствие 'car' в строке и вернет TRUE, как и ожидалось. Однако давайте рассмотрим еще несколько строк из набора данных, выполняя поиск того же шаблона:

```
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
```

Все эти строки также соответствуют шаблону 'car', несмотря на то, что 'scares' (пугает), 'carpenter' (плотник) и 'boxcar' (товарный вагон) — это не совсем то, что нам нужно, когда мы ищем упоминания об автомобилях. Чтобы исправить это, мы можем добавить \y в начало и конец 'car' в нашем шаблоне:

```
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
```

Конечно, в этом простом примере мы могли бы просто добавить пробелы до и после слова 'car' и получили бы тот же результат. Преимущество такого шаблона в

том, что он также соответствует случаям, когда слово находится в начале строки и поэтому не имеет перед собой символа пробела:

```
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
```

Шаблон `'\ycar\y'` найдет совпадение без учета регистра в строке, в которой `'Car'` является первым словом, а шаблон `' car'` не найдет ничего. Для обозначения начала всей строки используйте специальный символ `\A`, а для обозначения конца строки — `\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
```

В первой строке шаблон соответствует слову `'Car'` в начале строки. Вторая строка начинается с `'I'`, поэтому совпадения не найдено. В третьей строке шаблон соответствует слову `'car'` в конце строки. Наконец, в четвертой строке последнее слово — `'home'`, поэтому шаблон не совпадает.

Если вы впервые знакомитесь с регулярными выражениями, то вам может потребоваться перечитать объяснения пару раз и позапускать разные шаблоны в своем SQL-редакторе, чтобы освоить эту тему. Для закрепления знаний нет ничего лучше работы с реальными примерами, поэтому далее я расскажу, как можно использовать regex при анализе нашего набора данных о наблюдениях НЛО, а также представлю пару специфических regex-функций SQL.

Реализация регулярных выражений сильно различается у разных поставщиков баз данных. Стандарт POSIX, приведенный в этом разделе, работает в Postgres и в базах данных, производных от Postgres, таких как Amazon Redshift, но не обязательно будет работать в других базах.

Альтернативой компаратору `~` является функция `rlike` или `regexp_like` (в зависимости от типа базы данных). Они имеют следующий синтаксис:

```
regexp_like(string, pattern, optional_parameters)
```

Первый пример в этом разделе тогда будет выглядеть так:

```
SELECT regexp_like('The data is about UFOs','data')
 as comparison;
```

С помощью необязательного аргумента *optional_parameters* можно указать тип сопоставления, например, нечувствительное к регистру.

Многие базы данных имеют дополнительные функции, которые мы не будем здесь рассматривать, такие как `regexp_substr` для извлечения подстроки и `regexp_count` для определения количества вхождений шаблона. Postgres поддерживает стандарт POSIX, но, к сожалению, не поддерживает эти дополнительные функции. Организациям, которые планируют выполнять много текстового анализа, лучше выбрать тип базы данных с широким набором функций для работы с регулярными выражениями.

## Поиск и замена строк с помощью regex

В предыдущем разделе мы рассмотрели регулярные выражения и то, как с помощью regex составлять шаблоны для поиска строк в наших наборах данных. Давайте применим этот метод к набору данных о наблюдениях НЛО, чтобы увидеть, как это работает на практике. Кроме того, я также расскажу о некоторых дополнительных regex-функциях SQL.

Сообщения о наблюдениях НЛО содержат множество деталей, например, что делал наблюдатель в это время, когда и где это произошло. Еще одна часто упоминаемая деталь в сообщениях — это несколько огней. В качестве первого примера давайте найдем описания, содержащие число и слово `'light'` (свет) или `'lights'` (огни). Для удобства просмотра результатов я буду проверять только первые 50 символов, но этот код может работать и со всем полем `description`:

```
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
...
```

Шаблон регулярного выражения соответствует любому количеству цифр `[0-9]+`, за которыми следует пробел и слово `'light'`, после которого идет буква `'s'` или пробел, или запятая, или точка. Помимо поиска соответствий, мы хотим получить эту подстроку, которая совпадает с шаблоном. Для этого мы воспользуемся regex-функцией `regexp_matches`.

Реализация функций для работы с regex сильно варьируется в зависимости от поставщика базы данных, а иногда и от версии базы данных. Microsoft SQL Server не поддерживает эти функции, а в MySQL реализован минимальный набор. Аналитические базы данных, такие как Redshift, Snowflake и Vertica, поддерживают множество полезных функций. В Postgres есть только функции `regexp_matches` и `regexp_replace`. Проверьте документацию к вашей базе данных, чтобы узнать, какие функции доступны вам.

Функция `regexp_matches` принимает два аргумента: текст и шаблон regex. Она возвращает массив строк, которые соответствуют шаблону. Если совпадений нет, возвращается `null`. Поскольку возвращаемое значение является массивом, то мы будем обращаться к его первому элементу через индекс `[1]`, чтобы возвращалось только одно значение в виде `VARCHAR`, что позволит выполнять дополнительные действия со строками. Если вы работаете в базе данных другого типа, у вас может быть есть функция `regexp_substr`, аналогичная `regexp_matches`, но она возвращает значение `VARCHAR`, а не массив, поэтому нет необходимости добавлять индекс `[1]`.

*Массив* — это коллекция объектов, хранящихся вместе в памяти компьютера. В базах данных массивы всегда заключаются в фигурные скобки `{ }`, и это хороший способ сразу увидеть, что перед вами не один из обычных типов данных, с которыми мы работали до сих пор. Массивы имеют некоторые преимущества при хранении и получении данных, но с ними не так просто работать в SQL, поскольку они требуют специального синтаксиса. Доступ к элементам массива осуществляется с помощью индекса, заключенного в квадратные скобки `[ ]`. Для наших целей достаточно знать, что первый элемент массива можно получить с помощью `[1]`, второй — `[2]` и т. д.

Возвращаясь к нашему примеру, мы можем получить нужную подстроку — число и слово `'light(s)'` из поля описания, а затем сгруппировать по ней и подсчитать количество совпадений:

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

График результатов представлен на рис. 5.9.

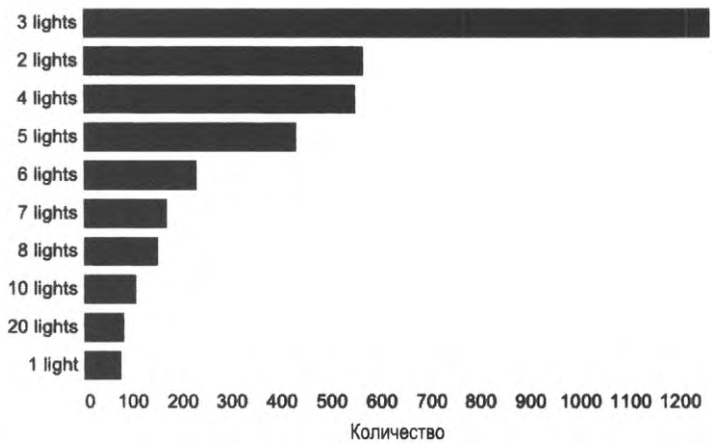


**Рис. 5.9.** Наиболее часто встречающиеся упоминания об огнях в описаниях наблюдений НЛО

Наиболее часто встречаются сообщения о наблюдении от двух до шести огней. А сообщения о трех огнях встречаются в два раза чаще, чем второе по частоте количество огней. Чтобы найти весь диапазон количества огней, мы можем разобрать найденную подстроку, а затем найти минимальное `min` и максимальное `max` значения:

```
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
```

По крайней мере в одном отчете упоминаются 2 000 огней, как и 0 огней. Возможно, надо будет отдельно просмотреть эти сообщения, чтобы выяснить, есть ли еще что-нибудь интересное или необычное, кроме этих экстремальных значений.

Помимо поиска совпадений, мы можем захотеть заменить найденную подстроку на другую. Это бывает особенно удобно при очистке набора данных, содержащего несколько вариантов написания одного и того же слова. Мы будем использовать функцию `regexp_replace`. Она похожа на функцию `replace`, рассмотренную ранее в *разд. 5.5*, но может принимать в качестве аргумента регулярное выражение для поиска соответствия. Синтаксис аналогичен функции `replace`:

```
regexp_replace(field_or_string, pattern, replacement_value)
```

Давайте применим эту функцию для очистки поля `duration`, которое мы ранее извлекли из столбца `sighting_report`. Похоже, что в это поле вводили произвольный текст, и в нем более 8000 различных значений. Однако проверка показывает, что большинство из них состоит из некоторой комбинации часов, минут и/или секунд:

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

В этой выборке продолжительность `'10 minutes'`, `'10 min'` и `'10 mins'` обозначают одно и то же количество времени, но база данных об этом не знает, потому что символы немного отличаются. Мы могли бы использовать несколько вложенных функций `replace` для замены всех этих различных написаний. Однако нам придется учитывать и другие вариации, например заглавные буквы. В этом случае удобнее использовать регулярные выражения, позволяющие писать более компактный код. Для начала давайте создадим шаблон, который будет находить соответствующие вхождения, что мы можем сделать с помощью функции `regexp_matches`. Нелишним

будет проверить этот промежуточный шаг, чтобы убедиться, что мы составили правильный шаблон:

```
SELECT duration
,(regexp_matches(duration
                ,'\m[Mm][Ii][Nn][A-Za-z]*\y')
                )[1] as matched_minutes
FROM
(
    SELECT split_part(sighting_report,'Duration:',2) as duration
    ,count(*) as reports
    FROM ufo
    GROUP BY 1
) a
;

duration      matched_minutes
------------  ---------------
10 min.       min
10 minutes+   minutes
10 min        min
10 minutes +  minutes
10 minutes?   minutes
10 minutes    minutes
10 mins       mins
...           ...
```

Давайте разберем этот код. В подзапросе значение `duration` выделяется из поля `sighting_report`. Затем функция `regexp_matches` ищет строки, соответствующие шаблону:

```
\m[Mm][Ii][Nn][A-Za-z]*\y
```

Этот шаблон начинается со специального символа начала слова `\m`, а затем идет любая последовательность букв `'m'`, `'i'` и `'n'`, независимо от регистра (`[Mm]` и т. д.). Далее ищется любое количество (ноль или более) буквенных символов `[A-Za-z]*`, и, наконец, проверяется конец слова `\y`, чтобы функция вернула только слово, производное от `'minutes'`, а не всю остальную часть строки. Обратите внимание, что специальные символы `+` и `?` здесь не подходят. Теперь с помощью этого шаблона мы можем заменить все эти вариации стандартным значением `'min'`:

```
SELECT duration
,(regexp_matches(duration
                ,'\m[Mm][Ii][Nn][A-Za-z]*\y')
                )[1] as matched_minutes
```

```
,regexp_replace(duration
                ,'\m[Mm][Ii][Nn][A-Za-z]*\y'
                ,'min') as replaced_text
FROM
(
    SELECT split_part(sighting_report,'Duration:',2) as duration
    ,count(*) as reports
    FROM ufo
    GROUP BY 1
) a
;

duration      matched_minutes  replaced_text
------------  ---------------  -------------
10 min.       min              10 min.
10 minutes+   minutes          10 min+
10 min        min              10 min
10 minutes +  minutes          10 min +
10 minutes?   minutes          10 min?
10 minutes    minutes          10 min
10 mins       mins             10 min
...           ...              ...
```

Значения в столбце `replaced_text` теперь гораздо более стандартизированы. Символы `'.'`, `'+'` и `'?'` после минут также можно заменить, чуть подправив регулярное выражение. Однако с аналитической точки зрения, возможно, стоит подумать о том, как отметить эту неопределенность, которую сейчас обозначают плюс и вопросительный знак. Функция `regexp_replace` может быть вложенной, чтобы добиться замены различных частей или типов строк. Например, мы можем стандартизировать и минуты, и часы:

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

Регулярное выражение для часов составляется аналогично регулярному выражению для минут — оно ищет совпадения слову `'hour'` без учета регистра, за которым может следовать ноль или более любых буквенных символов до конца слова. Вспомогательные столбцы `matched_hour` и `matched_minutes` могут и не пригодиться, но я считаю их полезными для дополнительной проверки, чтобы избежать лишних ошибок в дальнейшем. Полная очистка столбца `duration`, скорее всего, потребует гораздо больше строк кода, и очень легко ошибиться и сделать опечатку.

Функция `regexp_replace` может быть вложена любое количество раз, или ее можно использовать вместе с обычной функцией `replace`. Еще одно применение `regexp_replace` — в операторах `CASE` для выборочной замены при выполнении определенных условий. Регулярные выражения — это мощный и гибкий инструмент в SQL, который, как мы видели, можно по-разному использовать в SQL-запросах.

В этом разделе я показала несколько способов поиска и замены специфичных подстрок в длинных текстах, начиная с подстановочных знаков в операторе `LIKE` и заканчивая более сложными шаблонами с регулярными выражениями. Все это, наряду с функциями разбора и преобразования текста, представленными ранее, позволяет нам реализовывать наборы правил такой сложности, какая необходима для работы с реальными наборами данных. Однако стоит помнить о балансе между сложностью кода и легкостью его сопровождения. Для однократного анализа набора данных может быть уместно написать сложные правила, которые идеально бы очищали данные. Но для постоянной отчетности и мониторинга лучше рассмотреть возможность получения более чистых данных из самих источников данных. Далее мы рассмотрим несколько способов создания новых текстовых значений с помощью SQL, используя введенные строки, текстовые поля или разобранные значения.

# 5.7. Конкатенация и реорганизация

Мы уже изучили, как с помощью SQL разбирать, преобразовывать, находить и заменять фрагменты строк для выполнения различных задач очистки и анализа. Помимо этого, SQL можно использовать для создания новых текстов. В этом разделе я сначала расскажу о *конкатенации*, с помощью которой можно объединять различные поля и типы данных в один текст. Затем я покажу, как изменять организацию текстовых строк и столбцов с помощью функций, которые сворачивают несколько записей в одну строку, а также выполняют обратное действие: разбивают одну строку на несколько.

## Конкатенация строк

Новое текстовое значение может быть сформировано в SQL с помощью конкатенации. Любая комбинация набранного текста, текстовых полей базы данных и выражений с этими полями может быть соединена в одну строку. Существует несколько способов конкатенации. Большинство баз данных поддерживают функцию `concat`, которая принимает в качестве аргументов поля или значения, которые нужно объединить:

```
concat(value1, value2)
concat(value1, value2, value3, ...)
```

Некоторые базы данных поддерживают функцию `concat_ws` (от *concatenate with separator* — соединить с разделителем), которая принимает в качестве первого аргумента значение разделителя, а далее список значений для объединения. Это удобно, когда у вас есть несколько значений, которые нужно перечислить через запятую, тире или любой другой разделитель:

```
concat_ws(separator, value1, value2...)
```

И, наконец, оператор конкатенации `||` может использоваться во многих базах данных для объединения строк (но в Microsoft SQL Server вместо него обычный +):

```
value1 || value1
```

Если одно из значений равно `null`, результатом конкатенации будет `null`. Обязательно используйте `coalesce` или `CASE` для замены их значениями по умолчанию, если вы подозреваете, что они могут встречаться.

Конкатенация может объединять поле и строку. Представьте, например, что мы хотим добавить текстовую метку к форме объекта и к подсчитанному количеству сообщений с упоминанием каждой формы. Подзапрос достает форму объекта из поля `sighting_report` и подсчитывает количество записей. Внешний запрос объединяет форму со строкой `' (shape)'` и количество сообщений со строкой `' 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
...               ...
```

Мы также можем объединить два поля вместе, возможно, с разделителем строк. Например, мы можем объединить форму и местоположение в одно поле:

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

График результатов представлен на рис. 5.10.

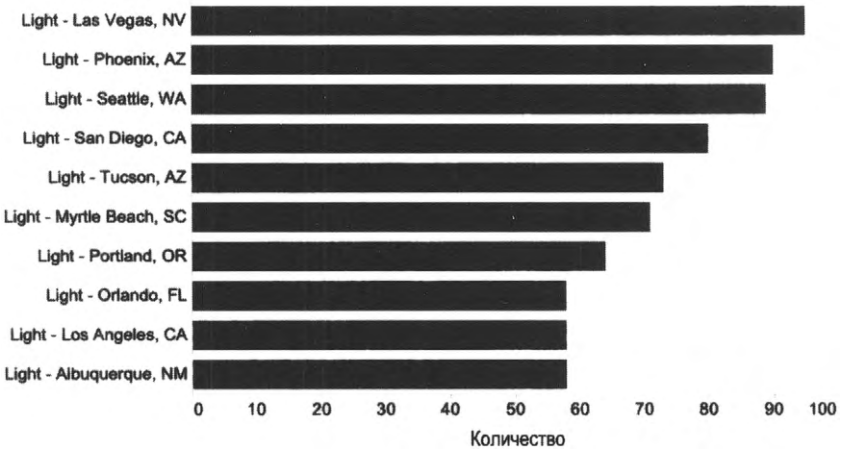


**Рис. 5.10.** Наиболее часто встречающиеся комбинации формы и местоположения в наблюдениях НЛО

Ранее мы видели, что `'light'` является самой распространенной формой, поэтому неудивительно, что все основные комбинации имеют именно эту форму.

В нашем случае, когда мы потратили столько сил на разбор сообщений и получение различных полей, вероятно, не имеет большого смысла снова объединять их вместе. Однако это может понадобиться для переформатирования текста или объединения значений в одно поле для отображения в другом инструменте. Комбинируя различные поля и текст, мы также можем создавать предложения, которые могут служить кратким изложением для использования в электронных письмах или в автоматизированных отчетах. В следующем примере в подзапросе `a` разбираются поля `occurred` и `shape`, как мы делали ранее, и подсчитывается количество записей. Затем в подзапросе `aa` вычисляются минимальное `min` и максимальное `max` значения `occurred`, а также общее количество сообщений `reports`, и результаты группируются по форме `shape`. Строки `occurred` короче восьми символов исключаются из рассмотрения, чтобы удалить неправильно сформированные даты и избежать ошибок при вычислении `min` и `max`. Наконец, во внешнем запросе собирается окончательное предложение с помощью функции `concat`. Для самой ранней и самой поздней даты формат изменен на полный (например `'April 9, 1957'`):

```
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(to_date(occurred, 'MM/DD/YYYY')) as earliest
    ,max(to_date(occurred, 'MM/DD/YYYY')) 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.
```

Мы можем подойти еще более творчески, например отформатировать количество сообщений или добавить `coalesce` или `CASE` для обработки пустых форм. Хотя эти предложения однообразные и очевидно, что они написаны не человеком (и не искусственным интеллектом), они будут динамически меняться, если набор данных обновится, и поэтому такая конструкция может пригодится для создания отчетов.

Наряду с функциями и операторами для создания новых текстовых значений с помощью конкатенации, в SQL есть несколько специальных функций для реорганизации текстовых полей, которые мы рассмотрим далее.

## Реорганизация текстовых полей

Как мы видели в *разд. 2.6*, реорганизация данных — сворачивание строк в столбцы или наоборот, разворачивание столбцов в строки — иногда бывает полезной. Мы знаем, как это сделать с помощью `GROUP BY` и агрегирования или с помощью операторов `UNION`. Однако в SQL есть несколько специальных функций для изменения структуры текстовых полей.

Один из случаев — это когда есть несколько записей с различными текстовыми значениями для конкретной сущности, и мы хотим соединить их в одну строку. Такое соединение значений может, конечно, усложнить анализ, но иногда требуется, чтобы в выходных данных была только одна запись для каждой сущности. Соединение отдельных значений в одном поле позволяет сохранить детализацию. Агрегатная функция `string_agg` принимает два аргумента: поле или выражение, которое надо собрать, и разделитель, чаще всего это запятая, но он может быть любым символом. Функция соединяет только те значения, которые не равны `null`, а порядок следования можно контролировать с помощью необязательного предложения `ORDER BY` в параметрах функции:

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

Поскольку `string_agg` является агрегатной функцией, в разделе `GROUP BY` нужно указать другие поля из запроса. В MySQL эквивалентной функцией является `group_concat`, а аналитические базы данных, такие как Redshift и Snowflake, имеют аналогичную функцию `listagg`.

Другой случай — когда нужно сделать прямо противоположное `string_agg` и разделить одно поле на несколько записей. В различных базах данных эта задача реализована по-разному, а в некоторых она не реализована вообще. В Postgres есть функция `regexp_split_to_table`, а в некоторых базах данных есть функция `split_to_table`, которая работает аналогично (проверьте документацию к вашей базе данных, что доступно вам и какой синтаксис у функции). Функция `regexp_split_to_table` принимает два аргумента: текстовое значение и разделитель. Разделителем может быть регулярное выражение, но не забывайте, что в качестве регулярного выражения можно использовать простой символ, например запятую или пробел. Функция разворачивает текстовое значение на строки:

```
SELECT
regexp_split_to_table('Red, Orange, Yellow, Green, Blue, Purple'
                      ,', ');

regexp_split_to_table
---------------------
Red
```

```
Orange
Yellow
Green
Blue
Purple
```

Разворачиваемая строка может включать в себя все, что угодно, и не обязательно должна быть строгим списком. Мы можем использовать эту функцию для разбиения любой строки, в том числе и целых предложений, на слова. Таким же способом мы можем найти наиболее часто встречающиеся слова, используемые в текстовом поле, что является очень нужным инструментом для анализа текста. Давайте найдем наиболее часто встречающиеся слова, используемые в описаниях наблюдений НЛО:

```
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
```

Подзапрос сначала преобразует поле `description` в нижний регистр, поскольку вариации с регистром не интересны в этом примере. Затем строка разделяется на слова с помощью регулярного выражения `'\s+'`, который обозначает один или несколько пробельных символов.

Наиболее часто используемые слова не вызывают удивления, однако они не являются особенно содержательными, т. к. это просто артикли, союзы и предлоги. Чтобы получить более осмысленный список, мы можем исключить из него так называемые *стоп-слова* — наиболее часто используемые слова в естественном языке. Некоторые базы данных предоставляют встроенные списки этих слов, поставляемые в так называемых словарях, но их реализация не стандартизирована. Также не существует единого согласованного списка стоп-слов, и обычно такой список подгоняется под конкретную задачу. Но в интернете можно найти несколько списков общих стоп-слов для разных языков. Для нашего примера я создала таблицу `stop_words` со списком из 421 английского стоп-слова, и вы можете найти SQL-код

для ее загрузки в репозитории GitHub[4] для этой книги. Стоп-слова удаляются из набора результатов с помощью `LEFT JOIN` с таблицей `stop_words`, отфильтровывая результаты, которых нет в этой таблице:

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

График 10 самых распространенных слов представлен на рис. 5.11.

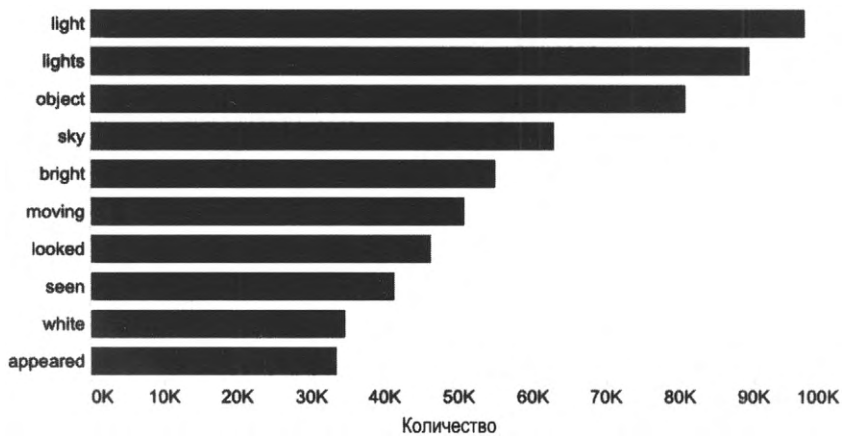


**Рис. 5.11.** Наиболее часто встречающиеся слова в описаниях наблюдений НЛО с исключением стоп-слов

[4] **https://github.com/cathytanimura/sql_book/tree/master/Chapter 5: Text Analysis**

Мы могли бы усложнить задачу, добавив какие-то общие слова в таблицу `stop_words` или объединив результаты с таблицей `ufo`, чтобы выбрать описания, которые содержат интересные слова. Обратите внимание, что `regexp_split_to_table` и подобные функции в некоторых базах данных могут работать медленно, в зависимости от длины текстов и количества записей.

Конкатенация и реорганизация текстовых полей с помощью SQL могут быть выполнены разными способами. Функции конкатенации, агрегирования и разбиения строк можно использовать как отдельно, так и в сочетании друг с другом или с другими функциями и операторами SQL для достижения нужного результата.

## 5.8. Заключение

Хотя SQL не всегда используется как инструмент для текстового анализа, он имеет множество мощных функций и операторов для работы с текстами: от разбора и преобразования, поиска и замены до объединения и разъединения текстовых полей. SQL можно использовать как для очистки и подготовки текстовых данных, так и для выполнения самого анализа.

В следующей главе мы рассмотрим использование SQL для выявления аномалий — еще одну тему, в которой не всегда упоминается SQL, но он обладает широкими возможностями для решения таких задач.

# ГЛАВА 6
# Выявление аномалий

*Аномалия* — это то, что отличается от других участников одной группы. Применительно к наборам данных аномалией считается запись, наблюдение или значение, которое настолько отличается от остальных точек данных, что вызывает опасения или сомнения. Аномалии могут также называться *выбросами*, *шумами*, *отклонениями* и *исключениями*. В этой главе я буду использовать термины «аномалия» и «выброс» как синонимы, но вы можете встретить и другие понятия, которые могут использоваться при обсуждении этой темы. Выявление аномалий может быть конечной целью анализа или одним из шагов более масштабного анализа.

Как правило, аномалии генерирует один из двух источников: реальные события, которые сами по себе являются экстремальными или необычными, или ошибки, допущенные во время сбора или обработки данных. Хотя многие методы, используемые для выявления аномалий, одинаковы для обоих случаев, то, как мы решаем поступить с конкретной аномалией, зависит от ее источника. Таким образом, знание источника аномалии и разница между этими двумя случаями имеют важное значение для анализа.

Реальные события могут генерировать выбросы по разным причинам. Аномальные данные могут сигнализировать о мошенничестве, вторжении в сеть, структурных дефектах товара, лазейках в политиках или использовании продукта не по назначению или в не предусмотренном разработчиками режиме. Выявление аномалий широко используется для борьбы с финансовым мошенничеством и в кибербезопасности. Иногда аномальные данные появляются не потому, что злоумышленник вломился в систему, а потому, что пользователь использует сервис непредвиденным образом. Например, я знала одного человека, который использовал приложение для фитнес-трекинга, предназначенное для бега, езды на велосипеде, ходьбы и других подобных занятий, для записи данных о своих гонках на автодроме. Он не нашел лучшего варианта и не подумал о том, насколько аномальны значения скорости и расстояния для автомобиля на треке по сравнению с данными, полученными при езде на велосипеде или беге. Когда аномалии можно отследить до реального процесса, решение о том, что с ними делать, требует понимания анализа, который необходимо провести, а также знания предметной области и условий использования, а иногда и правовых актов, регулирующих эту деятельность.

Данные также могут содержать аномалии из-за ошибок при сборе или обработке. Например, введенные вручную данные могут содержать опечатки и неправильные

данные. Изменения форм ввода, полей или правил валидации могут привести к появлению неожиданных значений, в том числе и `null`. Отслеживание действий с помощью веб- и мобильных трекингов является обычном делом, но любые изменения того, как и когда записывать, могут привести к аномалиям. Я потратила довольно много времени, пытаясь понять, почему изменились показатели, прежде чем выработала навык всегда сначала уточнять, не были ли недавно внесены какие-то изменения в логировании. Обработка данных может также привести к появлению аномалий, когда некоторые значения выбираются ошибочно, шаги обработки не завершаются или данные загружаются несколько раз, создавая дубликаты. Когда аномалии возникают в результате обработки данных, мы можем с большей уверенностью исправить или отбросить эти значения. Конечно, всегда лучше сразу исправить ввод или обработку данных, если это возможно, чтобы предотвратить будущие проблемы с качеством данных.

В этой главе я сначала приведу причины, почему можно использовать SQL для такого типа анализа, и упомяну случаи, когда SQL не подходит. Затем я представлю набор данных о землетрясениях, который мы будем использовать в примерах этой главы. После этого я расскажу об основных методах для обнаружения выбросов, которые можно выполнить с помощью SQL. Затем мы рассмотрим различные виды аномалий, для поиска которых мы можем применить эти методы. Как только мы нашли аномалии и выяснили их причину, следующий шаг — решить, что с ними делать. Аномалии не всегда сигнализируют о проблемах, как это бывает при выявлении мошенничества, обнаружении кибератак и мониторинге систем жизнеобеспечения. Методы, описанные в этой главе, также могут быть использованы для обнаружения лучших клиентов, удачных маркетинговых кампаний или положительных сдвигов в поведении клиентов. Иногда целью выявления аномалий является передача этой информации другим специалистам для устранения проблем, но зачастую это один из этапов более широкого анализа, поэтому в конце главы расскажу о различных вариантах исправления аномалий.

## 6.1. Возможности SQL для обнаружения аномалий

SQL — это гибкий и мощный язык для решения многих задач анализа данных, но он не все может сделать. Для работы с аномалиями в SQL есть ряд преимуществ, но для некоторых задач лучше использовать другие языки и инструменты.

SQL стоит взять на вооружение, если ваш набор данных уже находится в базе, как мы упоминали в *гл. 3* и *5*. Для выполнения быстрых расчетов над большим количеством записей SQL эффективно использует вычислительные мощности базы данных. При работе с большими таблицами перенос данных из базы в другое хранилище может занять особенно много времени. Работа с базой данных имеет еще большее преимущество, когда выявление аномалий является одним из этапов более масштабного анализа, который выполняется с помощью SQL. Код, написанный на SQL, можно проверить, чтобы определить, почему некоторые записи были помече-

ны как выбросы. Кроме того, SQL может оставаться неизменным с течением времени, даже если данные, поступающие в базу данных, меняются.

В качестве недостатка можно указать на то, что SQL не умеет выполнять сложные статистические вычисления, которые доступны в пакетах таких языков, как R и Python. В SQL есть несколько стандартных статистических функций, но более сложные статистические расчеты могут выполняться очень медленно или слишком нагружать некоторые базы данных. Для случаев, требующих очень быстрого реагирования, например при обнаружении мошенничества или вторжения, анализ данных в базе может просто не подойти, т. к. часто существует задержка при загрузке данных, особенно в аналитические базы данных. Обычный рабочий процесс заключается в использовании SQL для проведения первоначального анализа и определения минимальных, максимальных и средних значений, а после этого следует разработка более оперативного мониторинга с использованием потокового сервиса или специального хранилища данных. Как вариант, сначала можно определить типовые выбросы в SQL, а после реализовать это в потоковых сервисах или в специальных хранилищах данных. И, наконец, код SQL опирается на правила, как обсуждалось в *гл. 5*. Он очень хорош для выполнения определенного набора правил, но не сможет автоматически подстроиться под быстро меняющиеся условия и шаблоны. Как правило, для таких задач лучше подходят методы машинного обучения и связанные с ними языки.

Теперь, когда мы обсудили преимущества SQL и то, в каких случаях его следует использовать, давайте посмотрим на данные, которые мы будем анализировать в примерах этой главы, прежде чем перейдем к самому коду.

## 6.2. Набор данных о землетрясениях

Данные для примеров этой главы представляют собой набор записей обо всех землетрясениях, зарегистрированных Геологической службой США (US Geological Survey, USGS) с 2010 по 2020 г. USGS предоставляет данные в различных форматах, в том числе и в режиме реального времени, на сайте:

**https://earthquake.usgs.gov/earthquakes/feed**

Как и ранее, этот набор данных в формате CSV и SQL для его загрузки сохранены в репозитории GitHub[1] для этой книги. Набор данных содержит около 1.5 миллиона записей. Каждая запись представляет собой одно зафиксированное землетрясение и содержит такие данные, как временная метка, местоположение, магнитуда, глубина и источник информации. Образец данных показан на рис. 6.1. Полный словарь данных можно посмотреть на сайте USGS[2].

---

[1] **https://github.com/cathytanimura/sql_book/tree/master/Chapter 6: Anomaly Detection**

[2] **https://earthquake.usgs.gov/data/comcat/data-eventterms.php**

Землетрясения вызываются скольжением пород вдоль разломов тектонических плит. В местах, расположенных вблизи этих разломов, землетрясения происходят гораздо чаще и более сильно, чем в других местах. Так называемое *Огненное кольцо* — это регион по периметру Тихого океана, в котором происходит множество землетрясений. Различные места в этом регионе, включая Калифорнию, Аляску, Японию и Индонезию, будут часто упоминаться в нашем анализе.

| | 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 |

**Рис. 6.1.** Фрагмент таблицы `earthquakes`

*Магнитуда* — это мера силы землетрясения в его очаге, определяемая по сейсмическим волнам. Магнитуда записывается по шкале десятичного логарифма, т. е. амплитуда землетрясения магнитудой 5 в 10 раз больше, чем амплитуда землетрясения магнитудой 4. Как выполняется измерение силы землетрясения очень интересно, но выходит за рамки данной книги. Если вы хотите узнать больше, начните с материалов на сайте Геологической службы США[3].

## 6.3. Поиск аномалий

Хотя само понятие аномалии или выброса как точки данных, которая сильно отличается от остальных, кажется простым, на самом деле обнаружение такой точки в любом наборе данных сопряжено с некоторыми трудностями. Во-первых, надо определить, когда значение (или точка данных) считается обычным, а когда — исключительным. А во-вторых, нужно определить пороговые значения по обе стороны от этой разделительной линии. При профилировании данных о землетрясениях мы будем исследовать магнитуду и глубину, чтобы понять, какие значения являются нормальными, а какие — необычными.

[3] **https://earthquake.usgs.gov/**

Как правило, чем больше и полнее набор данных, тем легче решить, что именно считать аномалией. В некоторых случаях у нас есть значения, помеченные как «достоверные данные», на которые мы можем опираться. В качестве такой метки обычно выступает столбец, в котором указано, является ли данная запись нормальной или выбросом. Достоверность данных может быть известна из отраслевых или научных источников или из прошлого анализа. Например, мы можем знать, что любое землетрясение с магнитудой более 7 баллов является аномалией. В некоторых случаях нам приходится опираться только на данные и самостоятельно оценивать уровень достоверности. В следующих разделах мы будем считать, что у нас есть достаточно большой набор данных для самостоятельного принятия решений, хотя, конечно, есть и внешние источники, к которым можно обратиться за консультацией о типичных и экстремальных магнитудах землетрясений.

Есть несколько способов обнаружения выбросов в наборе данных. Во-первых, мы можем отсортировать значения с помощью `ORDER BY`. Это можно объединить с различными группировками `GROUP BY` для поиска выбросов по частоте. Во-вторых, мы можем использовать статистические функции SQL для поиска экстремальных значений на любом конце диапазона значений. И, наконец, мы можем построить график и визуально проанализировать данные.

## Сортировка для поиска аномалий

Одним из основных способов для поиска аномалий является сортировка данных, выполняемая с помощью предложения `ORDER BY`. По умолчанию `ORDER BY` сортирует значения по возрастанию (`ASC`). Для сортировки по убыванию нужно добавить `DESC` после имени столбца. Предложение `ORDER BY` может включать один или несколько столбцов, и для каждого столбца можно задать свое направление сортировки, независимо друг от друга. Сначала сортировка выполняется по первому указанному столбцу. Если указан второй столбец, результаты сортируются дополнительно по второму столбцу (с сохранением первой сортировки), и так далее по всем столбцам в `ORDER BY`.

Поскольку сортировка выполняется после того, как база данных вычислила остальную часть запроса, многие базы позволяют ссылаться на столбцы запроса по позиции, а не по имени. Microsoft SQL Server является исключением, он требует полных имен столбцов. Я предпочитаю позиционную нотацию, поскольку она позволяет получить более компактный код, особенно когда столбцы запроса содержат в себе длинные вычисления или вложенные функции.

Например, мы можем отсортировать таблицу `earthquakes` по магнитуде:

```
SELECT mag
FROM earthquakes
ORDER BY 1 desc
;
```

```
mag
------
(null)
(null)
(null)
...
```

Мы получили ряд строк с null-значениями. Отметим, что набор данных может содержать null-значения для магнитуды — это тоже возможные выбросы. Давайте исключим их:

```
SELECT mag
FROM earthquakes
WHERE mag is not null
ORDER BY 1 desc
;

mag
---
9.1
8.8
8.6
8.3
```

Мы получили только одно значение магнитуды больше 9 и два значения больше 8.5. Во многих контекстах эти значения покажутся не особенно большими. Однако, обладая некоторыми знаниями о землетрясениях, мы понимаем, что такая магнитуда на самом деле очень большая и необычная. Геологическая служба США приводит список 20 крупнейших землетрясений в мире[4]. Все они имеют магнитуду 8.4 и выше, только пять из них — магнитуду 9.0 и выше, и всего трое из списка произошли в период с 2010 по 2020 г., т. е. в период, охватываемый нашим набором данных.

Еще один способ определить, являются ли значения аномальными, — это подсчитать их частоту в наборе данных. Мы можем подсчитать count(id) и сгруппировать по полю mag, чтобы найти количество землетрясений для каждой магнитуды. Затем это количество делится на общее количество землетрясений, которое можно найти с помощью оконной функции sum. Для всех оконных функций требуется указывать раздел OVER с предложениями PARTITION BY и/или ORDER BY. Поскольку знаменатель должен учитывать все записи, я добавила PARTITION BY 1, чтобы заставить базу данных считать это оконной функцией и просуммировать данные по всей таблице. Наконец, результаты сортируются с помощью ORDER BY по магнитуде:

```
SELECT mag
,count(id) as earthquakes
```

[4] **https://www.usgs.gov/programs/earthquake-hazards/science/20-largest-earthquakes-world**

```
,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
...  ...          ...
```

Есть только по одному землетрясению для магнитуд, превышающих 8.5, но есть два землетрясения с магнитудой 8.3. В районе магнитуды 6.9 количество землетрясений становится двузначным, но они все равно составляют очень малый процент от всех данных. При профилировании мы также должны проверить другой конец диапазона значений — наименьшие значения магнитуды, изменив порядок сортировки на противоположный:

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

На другом конце значения –9.99 и –9 встречаются чаще, чем мы ожидали. Хотя мы не можем взять логарифм от нуля или отрицательного числа, но сам десятичный логарифм может быть отрицательным, если аргумент больше нуля и меньше единицы. Например, log(0.5) равен приблизительно –0.301. Значения –9.99 и –9 представляют собой чрезвычайно малые магнитуды землетрясений, и мы можем задаться вопросом, действительно ли наблюдались такие малые землетрясения. Учитывая частоту появления этих значений, мне кажется, что они представляют собой скорее неизвестную величину, чем действительно микроскопическое землетрясение, и поэтому мы можем считать их аномалиями.

Кроме общей сортировки данных, может быть полезно сгруппировать `GROUP BY` по одному или нескольким полям атрибутов, чтобы найти аномалии в подмножествах данных. Например, мы можем проверить наибольшие и наименьшие магнитуды, зарегистрированные в определенных географических регионах, записанных в поле `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  1
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
```

Северная Калифорния `'Northern California'` — самое упоминаемое место в наборе данных, и для этого подмножества мы видим, что большие и малые значения магнитуды не являются такими экстремальными, как для набора данных в целом. Землетрясения магнитудой более 5.0 не являются редкостью для всего набора данных, но для Северной Калифорнии они могут быть выбросами.

## Расчет процентилей и стандартных отклонений

Сортировка и произвольная группировка данных с последующим визуальным просмотром результатов — это удобный способ выявления аномалий, особенно когда в данных присутствуют очень экстремальные значения. Однако без знаний предмет-

ной области может быть неочевидным то, что землетрясение магнитудой 9.0 является аномалией. Количественная оценка экстремальности точек данных поднимет точность анализа. Это можно сделать двумя методами: с помощью процентилей или с помощью стандартных отклонений.

*Процентили* представляют собой процент точек в распределении, которые меньше определенного значения. *Медиана* распределения или 50-й процентиль — это значение, относительно которого половина точек имеет меньшее значение, а вторая половина — большее. Медиана очень часто используется, поэтому во многих базах данных реализована отдельная SQL-функция — `median`. Можно вычислить и другие процентили. Например, можно найти 25-й процентиль, для которого 25% всех значений ниже его, а 75% — выше, или 89-й процентиль, для которого 89% значений ниже его, а 11% — выше. Процентили часто встречаются в научных контекстах, таких как стандартизированное тестирование, но их можно использовать в любой предметной области.

В SQL есть оконная функция `percent_rank`, которая возвращает процентиль для каждой строки в секции. Как и во всех оконных функциях, порядок сортировки контролируется с помощью предложения `ORDER BY`. Как и функция `rank`, `percent_rank` не принимает никаких аргументов, она работает со всеми строками запроса. Выглядит она следующим образом:

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

Предложения `PARTITION BY` и `ORDER BY` необязательные, но требуется что-то указать в разделе `OVER`, а уточнить сортировку — всегда хорошая идея. Чтобы найти процентиль магнитуды каждого землетрясения для каждого места, мы можем сначала вычислить `percent_rank` для каждой строки в подзапросе, а затем подсчитать количество вхождений во внешнем запросе. Обратите внимание, что сначала нужно вычислить `percent_rank`, прежде чем выполнять какое-то агрегирование, чтобы при вычислении процентилей учитывались повторяющиеся значения:

```
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
```

В Северной Калифорнии магнитуда 5.6 соответствует процентилю 1 или 100%, что указывает на то, что все остальные точки данных меньше этого значения. Магнитуда –1.6 соответствует процентилю 0, что указывает на то, что нет ни одной другой точки данных, меньше этой. В дополнение к поиску точного процентиля каждой строки SQL может разбить набор данных на определенное количество интервалов и для каждой записи вернуть номер интервала, к которому она относится, с помощью функции `ntile`. Например, мы можем разбить набор данных на 100 интервалов:

```
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
```

Глядя на результаты для `'Central Alaska'`, мы видим, что три магнитуды, превышающие 5, находятся в 100-м интервале, магнитуда 1.5 попадает в 79-й интервал, а наименьшее значение –0.5 попадает в первый интервал. Вычислив эти значения, мы можем найти границы магнитуд для каждого *n*-тиля, используя функции `max` и

min. В следующем примере мы будем использовать только четыре интервала, чтобы упростить отображение:

```
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
```

Самый старший *n*-тиль, 4-й, который представляет собой интервал между 75-м и 100-м процентилями, охватывает самый широкий диапазон магнитуд — от 1.4 до 5.4. В то же время средние 50% всех значений, к которым относятся 2-й и 3-й *n*-тили, варьируются только от 0.8 до 1.4.

Помимо нахождения процентиля или *n*-тиля для каждой записи, мы можем рассчитать конкретный процентиль по всем результатам запроса. Для этого можно использовать функцию percentile_cont или percentile_disc. Обе функции являются оконными, но с немного иным синтаксисом, чем мы использовали ранее, поскольку для них требуется раздел WITHIN GROUP. Функции имеют следующий вид:

```
percentile_cont(numeric) within group (order by field_name) over (partition
by field_name)
```

Значение *numeric* — это процентиль от 0 до 1, который необходимо вычислить. Например, для вычисления 25-го процентиля надо указать 0.25. Предложение ORDER BY определяет поле, по которому будет вычисляться процентиль, а также порядок сортировки. По желанию можно добавить ASC (по умолчанию) или DESC. Раздел OVER (PARTITION BY...) является необязательным (и, что странно, некоторые базы данных не поддерживают его в этих функциях, поэтому если у вас возникла ошибка, проверьте документацию).

Функция percentile_cont вернет интерполированное (рассчитанное) значение, которое соответствует точному процентилю, но которое может отсутствовать в наборе данных. Функция percentile_disc (дискретный процентиль) возвращает значение из набора данных, которое ближе всего к запрашиваемому процентилю. Для больших наборов данных или для наборов с достаточно непрерывными значениями явного различия между результатами этих двух функций почти не будет, но стоит подумать, какая из них больше подходит в вашем случае. Давайте рассмотрим пример, чтобы понять, как это будет выглядеть на практике. Мы рассчитаем 25-й, 50-й (медиана) и 75-й процентили для магнитуд, не равных null, для Центральной Аляски:

```
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
```

Запрос возвращает нужные процентили, определенные по всему набору данных. Обратите внимание, что значения соответствуют максимальным значениям для *n*-тилей 1, 2 и 3, рассчитанных в предыдущем примере. Процентили для разных полей можно рассчитать в одном запросе, указав другое поле в 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
```

В отличие от других оконных функций, percentile_cont и percentile_disc требуют указать раздел GROUP BY для всего запроса, если в запросе возвращаются и другие поля. Например, если мы хотим рассмотреть два места на Аляске и нам надо вклю-

чить поле place в результаты запроса, то нужно также указать это поле в GROUP BY, и процентили будут вычисляться отдельно для каждого места:

```
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
```

С помощью этих функций мы можем найти любой процентиль, необходимый для анализа. В некоторых базах данных также реализована функция median, которая имеет только один аргумент — поле, для которого нужно вычислить медиану. Это удобный и, конечно, гораздо более простой синтаксис, но обратите внимание, что то же самое можно сделать с помощью percentile_cont, если функция median недоступна в вашей базе данных.

Функции percentile_cont, percentile_disc и median могут быть медленными и тяжелыми для больших наборов данных. Это связано с тем, что база данных должна сортировать и ранжировать все записи. Некоторые поставщики баз данных реализовали приближенные версии функций, например approximate_percentile, которые работают намного быстрее и возвращают результаты, очень близкие к функции, которая считает по всему набору данных.

Значения процентилей или *n*-тилей позволяют нам применить количественную оценку для выявления аномалий. Позже в *разд. 6.5* мы увидим, как эти значения помогают обрабатывать аномалии в наборах данных. Поскольку процентили всегда масштабируются от 0 до 100, сами они не дают представления о том, насколько странными и необычными являются некоторые значения. Для этого нам нужно использовать дополнительные статистические функции, поддерживаемые SQL.

Чтобы определить, насколько экстремальными являются значения в наборе данных, мы можем использовать *стандартное отклонение*. Стандартное отклонение — это показатель вариативности набора значений. Меньшее значение этого показателя означает меньшую вариативность, и наоборот. Когда данные имеют нормальное распределение, около 68% всех значений лежат в пределах +/– одно стандартное

отклонение от среднего значения, а около 95% — в пределах +/– два стандартных отклонения. Стандартное отклонение σ рассчитывается по формуле:

$$\sigma = \sqrt{\sum (x_i - \mu)^2 / N},$$

где $x_i$ — наблюдение, μ — среднее арифметическое всех наблюдений, а $N$ — количество наблюдений. Более подробную информацию о том, как рассчитывается стандартное отклонение, можно найти в любом учебнике по статистике или на интернет-ресурсах[5].

В большинстве баз данных реализовано три функции стандартного отклонения. Функция `stddev_pop` возвращает стандартное отклонение всей совокупности. Если набор данных представляет всю совокупность точек данных, как это часто бывает с набором данных о клиентах, используйте `stddev_pop`. Функция `stddev_samp` находит стандартное отклонение для выборки и отличается от приведенной выше формулы делением на $N - 1$ вместо $N$. Это увеличивает стандартное отклонение, компенсируя потерю точности при использовании только выборки, а не всей совокупности. Функция `stddev`, доступная во многих базах данных, идентична функции `stddev_samp` и может быть использована просто потому, что имеет короткое название. Если вы работаете с данными, которые являются выборкой, например из опроса или исследования более крупной совокупности, используйте `stddev_samp` или `stddev`. На практике, когда вы работаете с большими наборами данных, разница между результатами `stddev_pop` и `stddev_samp` обычно невелика. Например, для 1.5 миллионов записей таблицы `earthquakes` значения начинают расходиться только после пяти знаков после запятой:

```
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
```

Эти различия достаточно малы, чтобы для большинства практических приложений не имело значения, какую функцию стандартного отклонения использовать.

Теперь мы можем рассчитать, на сколько стандартных отклонений значение из набора данных удалено от среднего значения. Этот показатель называется *z-оценкой* (z-score) и является способом стандартизации данных. Значения, которые выше среднего, имеют положительную z-оценку, а те, которые ниже среднего, — отрицательную z-оценку. На рис. 6.2 показано, как z-оценка и стандартное отклонение соотносятся с нормальным распределением.

---

[5] На сайте **https://www.mathsisfun.com/data/standard-deviation-formulas.html** есть хорошее объяснение, что такое стандартное отконение.

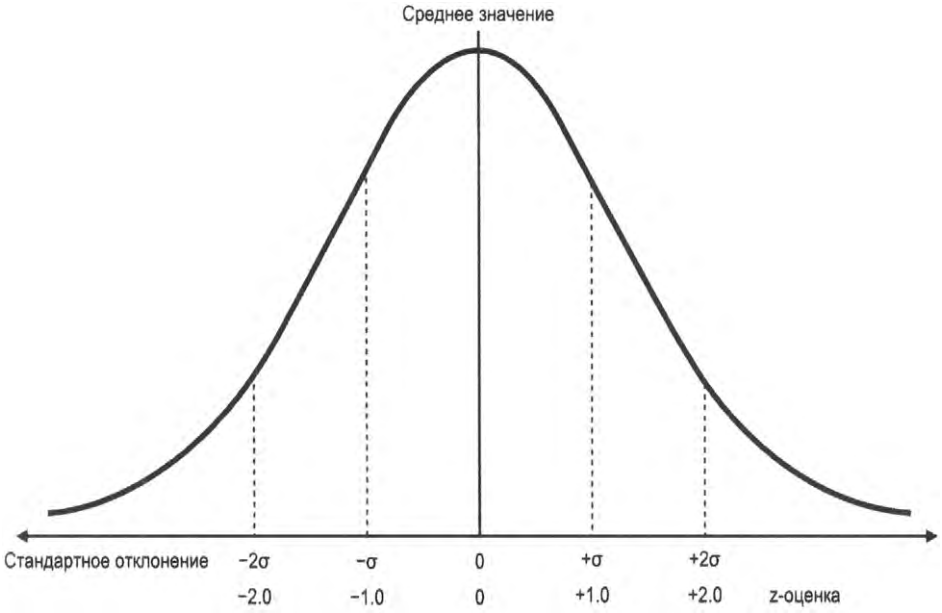


**Рис. 6.2.** Стандартное отклонение и z-оценка для нормального распределения

Чтобы найти z-оценку для землетрясений, сначала надо рассчитать среднее значение и стандартное отклонение для всего набора данных в подзапросе. Затем соединим их с набором данных с помощью декартова соединения `JOIN`, чтобы среднее значение и стандартное отклонение были добавлены к каждой записи. Это делается с помощью условия `1 = 1`, поскольку большинство баз данных требуют обязательно указать какое-нибудь условие `JOIN`.

Во внешнем запросе вычисляем разницу между каждой магнитудой и средним значением, а затем делим на стандартное отклонение:

```
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
```

Крупнейшие землетрясения имеют z-оценку, равную почти 6, тогда как самые небольшие землетрясения (исключая магнитуды –9 и –9.99, которые, по-видимому, являются аномалиями при вводе данных) имеют z-оценку, близкую к –3. Можно сделать вывод, что магнитуды самых крупных землетрясений являются более экстремальными выбросами, чем магнитуды небольших землетрясений.

## Поиск аномалий с помощью графиков

Помимо сортировки данных и расчета процентилей и стандартных отклонений, представление данных в виде графика также может помочь в поиске аномалий. Как мы уже видели в предыдущих главах, одним из достоинств графиков является их способность обобщать и представлять множество точек данных в компактной форме. Анализируя графики, мы можем заметить закономерности и отклонения от нормы, которые могли бы пропустить, если бы рассматривали только исходные данные. Наконец, графики помогают презентовать данные и любые потенциальные проблемы, связанные с аномалиями, другим людям.

В этом разделе мы рассмотрим три типа графиков, которые полезны для выявления аномалий: гистограмма, диаграмма рассеяния и диаграмма размаха («ящик с усами»). Чтобы создать выходные данные для этих графиков, используется простой SQL-код, хотя вам может понадобиться дополнительная реорганизация данных, обсуждавшаяся в предыдущих главах, — это зависит от возможностей и ограничений программного обеспечения, используемого для создания графиков. Любой серьезный BI-инструмент и электронные таблицы, а также такие языки, как Python и R, могут создавать такие графики. Для построения графиков в этом разделе я использовала язык Python с библиотекой Matplotlib.

*Гистограмма* используется для построения распределения значений и удобна как для знакомства с данными, так и для обнаружения выбросов. По оси $x$ откладывается полный диапазон значений поля, а по оси $y$ — количество повторений каждого значения. Интересны самые высокие и самые низкие значения, а также форма графика. Мы можем быстро определить, является ли распределение близким к нормальному (симметричным относительно пика или среднего значения), или имеет другой тип распределения или дополнительные пики при каких-то значениях.

Чтобы построить гистограмму для магнитуд землетрясений, сначала нужно сформировать набор данных, в котором будет группировка по магнитуде и подсчет количества землетрясений. Затем мы можем построить график, который будет выглядеть так, как показано на рис. 6.3.

```
SELECT mag
,count(*) as earthquakes
FROM earthquakes
GROUP BY 1
ORDER BY 1
;

mag    earthquakes
-----  -----------
-9.99  258
-9     29
-5     1
...    ...
```

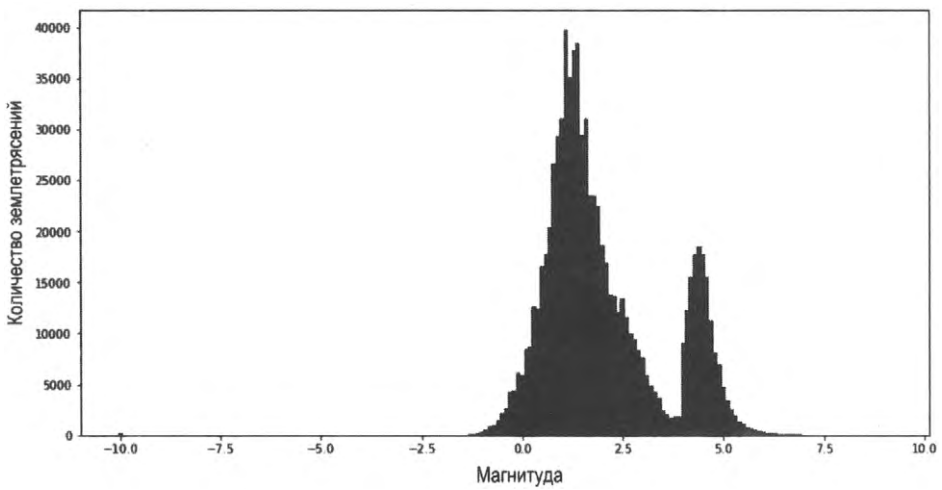


**Рис. 6.3.** Распределение магнитуд землетрясений, которые произошли с 2010 по 2020 г.

График находится между −10.0 и +10.0, что вполне логично, учитывая наше предыдущее исследование данных. Он примерно симметричен относительно пика, находящегося в диапазоне от 1.1 до 1.4 и достигающего почти 40 000 землетрясений, но имеет второй пик почти в 20 000 землетрясений в районе значения магнитуды 4.4. Однако на этом графике трудно заметить экстремальные значения, поэтому мы можем увеличить правый участок графика, как показано на рис. 6.4.

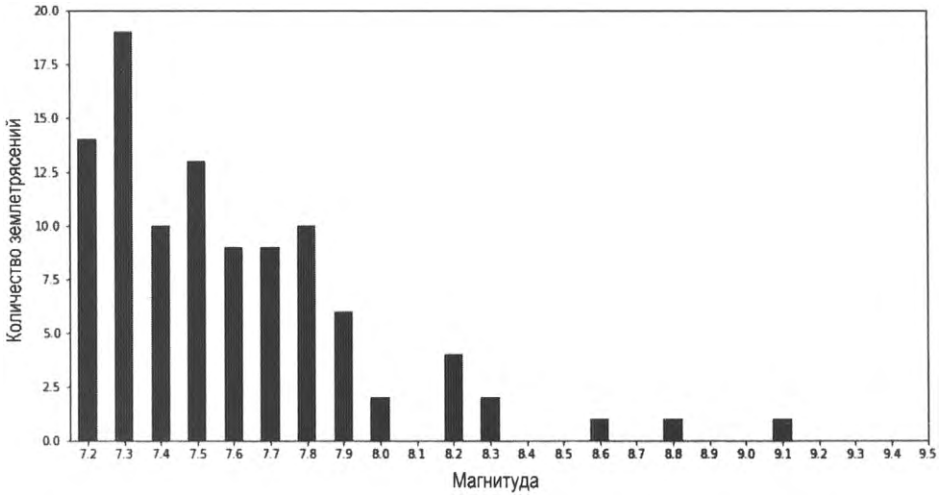


**Рис. 6.4.** Увеличенный фрагмент распределения магнитуд землетрясений с самыми высокими значениями магнитуд

На этом графике легче увидеть частоту землетрясений с очень высокой магнитудой, а также резкое снижение частоты с 10 до 1 по мере того, как значение магнитуды подходит к 8. К счастью, такие сильные землетрясения происходят крайне редко.

Второй тип графика, который можно использовать для визуального анализа данных и обнаружения выбросов, — это *диаграмма рассеяния*. Диаграмма рассеяния подходит, когда набор данных содержит как минимум два числовых поля, которые представляют интерес. По оси *x* откладывают значения первого поля, по оси *y* — значения второго поля, и для каждой пары значений *x* и *y* из набора данных на график наносится точка. Например, мы можем построить график зависимости магнитуды от глубины землетрясений. Сначала давайте напишем запрос, чтобы создать набор данных для каждой пары значений, а затем построим график, как на рис. 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
...    ...     ...
```

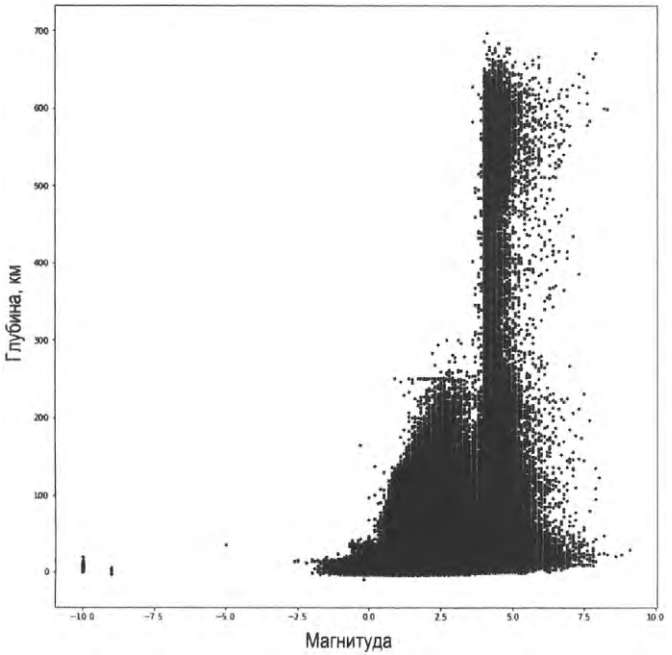


**Рис. 6.5.** Диаграмма рассеяния магнитуды и глубины землетрясений

На этом графике мы видим тот же диапазон магнитуд, но теперь в сравнении с глубиной, которая варьируется от чуть ниже нуля до почти 700 км. Интересно, что большие значения глубины (более 300 км) соответствуют магнитудам от 4 и выше. Возможно, такие глубокие землетрясения могут быть зафиксированы только после того, как они превысят какой-то минимальный порог магнитуды. Обратите внимание, что из-за большого объема данных я сократила количество записей и сгруппировала значения по магнитуде и глубине, вместо того чтобы наносить на график все 1.5 миллиона точек данных. Количество землетрясений можно использовать для определения размера каждой точки на диаграмме рассеяния, как показано на рис. 6.6, где приведен увеличенный фрагмент для диапазона магнитуды от 4.0 до 7.0 и глубины от 0 до 50 км.

Третий тип графика, удобный для поиска и анализа выбросов, — это *диаграмма размаха*, также известная как «ящик с усами». Этот график обобщает данные в середине диапазона значений, сохраняя при этом выбросы. Тип графика назван так из-за прямоугольника в центре. Нижняя сторона прямоугольника расположена на уровне 25-го процентиля, верхняя сторона — на уровне 75-го процентиля, а линия в середине — на уровне 50-го процентиля или медианы. Процентили мы разбирали в предыдущем разделе. «Усы» ящика — это линии, выходящие за пределы прямоугольника, обычно в 1.5 раза превышающие межквартильный размах. *Межквартильный размах* — это разница между 75-м и 25-м процентилями. Любые значения, выходящие за границы усов, отображаются на графике как выбросы.

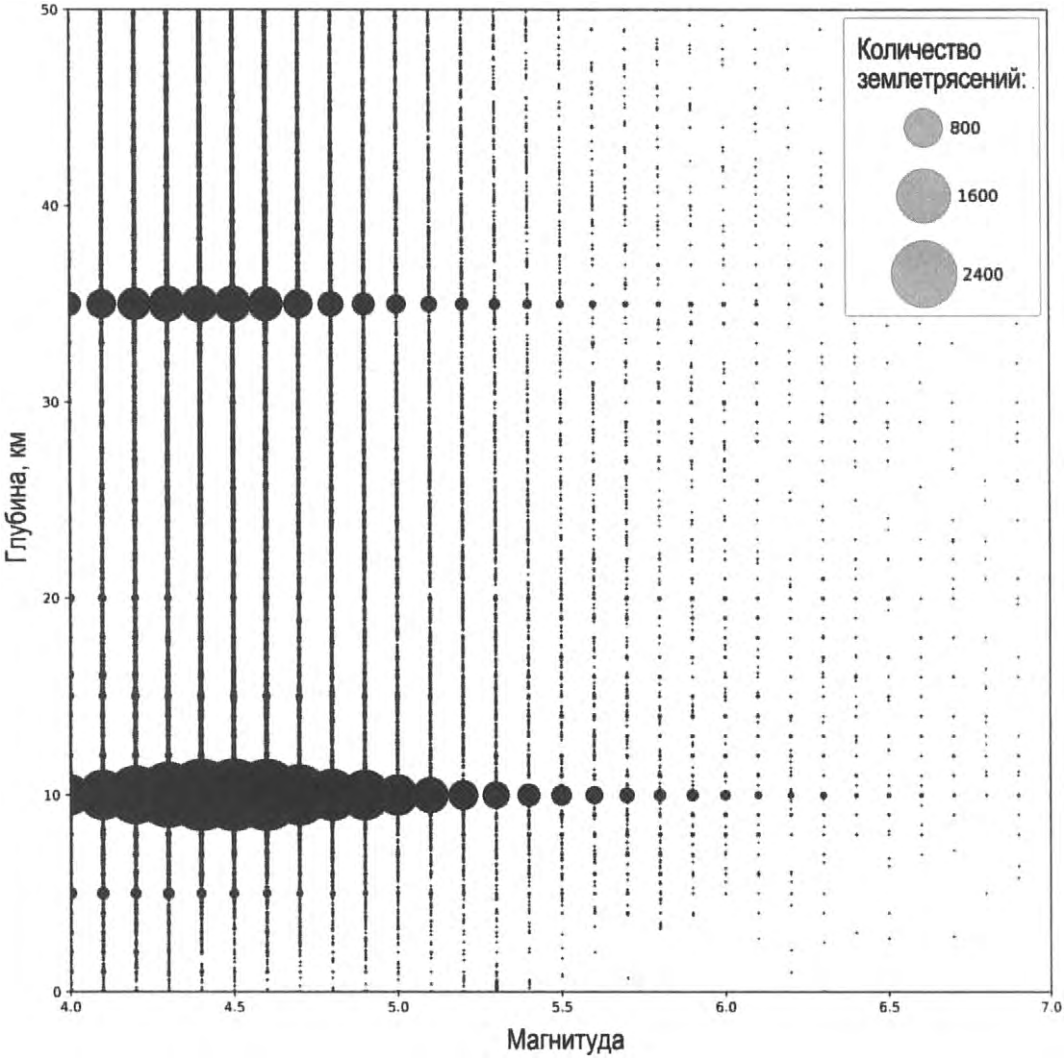


**Рис. 6.6.** Увеличенный фрагмент диаграммы рассеяния магнитуды и глубины землетрясений с учетом размера точек, который обозначает количество землетрясений

Какой бы инструмент или язык программирования вы ни использовали для построения диаграмм размаха, он сам выполнит вычисление процентилей и межквартильного размаха. Многие из них также предлагают варианты построения усов на основе стандартных отклонений от среднего или более широких процентилей, например 10-го и 90-го. Расчеты всегда будут симметричны относительно средней точки (например, одно стандартное отклонение выше и ниже среднего), но длины верхнего и нижнего усов могут отличаться в зависимости от данных.

Как правило, на диаграмме размаха отображаются все значения. Поскольку наш набор данных очень большой, давайте рассмотрим подмножество из 16 036 землетрясений, в поле `place` которых упоминается `'Japan'` (Япония). Сначала создадим

набор данных с помощью SQL, который представляет собой простой SELECT всех значений mag, удовлетворяющих критериям фильтра:

```
SELECT mag
FROM earthquakes
WHERE place like '%Japan%'
ORDER BY 1
;

mag
---
2.7
3.1
3.2
...
```

Затем можно создать диаграмму размаха в любом инструменте для построения графиков, как на рис. 6.7.

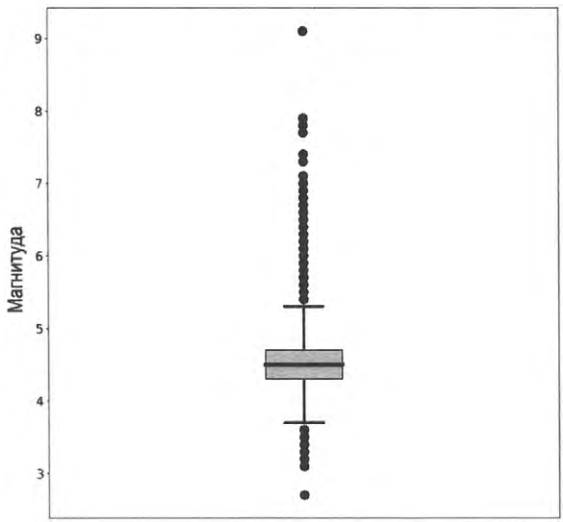


**Рис. 6.7.** Диаграмма размаха магнитуд землетрясений в Японии

Хотя программное обеспечение для построения графиков часто предоставляет эту информацию, мы также можем найти ключевые значения для диаграммы размаха с помощью 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
```

Медиана магнитуды японских землетрясений равна 4.5, а усы тянутся от 3.7 до 5.3. Точки на графике представляют собой выбросы, как незначительные, так и большие. Великое землетрясение Тохоку в 2011 г. с магнитудой 9.1 является очевидным выбросом даже среди более сильных землетрясений, которые пережила Япония.

По моему опыту, диаграмма размаха — один из самых сложных графиков для понимания для тех, кто не имеет статистического образования или не проводит все дни за созданием и просмотром визуализаций. Особенно запутанным является понятие межквартильного размаха, хотя почти все понимают, что такое выбросы. Если вы не уверены, что ваша аудитория знает, как интерпретировать диаграмму размаха, потратьте время на то, чтобы объяснить ее в четких, но не слишком технических терминах. У меня есть схема, как на рис. 6.8, которая объясняет, из чего состоит диаграмма размаха, и я отправляю ее вместе со своими графиками, на всякий случай, если моей аудитории понадобится пояснение.

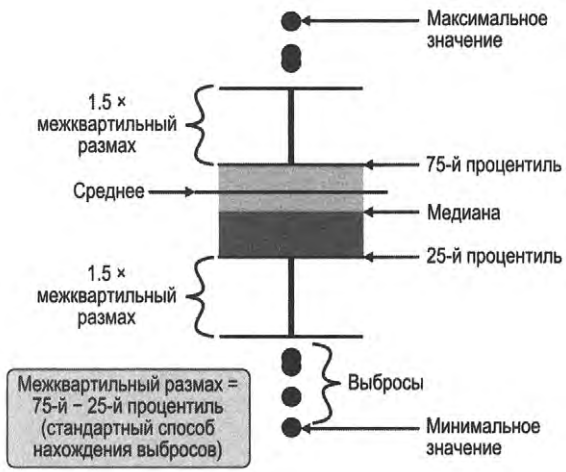


**Рис. 6.8.** Составляющие диаграммы размаха

Диаграммы размаха также можно использовать для сравнения различных групп, чтобы определить, где именно встречаются выбросы. Например, мы можем сравнить землетрясения в Японии за разные годы. Для этого в результаты запроса нужно добавить год из поля `time`, а затем построить график, как на рис. 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
...   ...
```

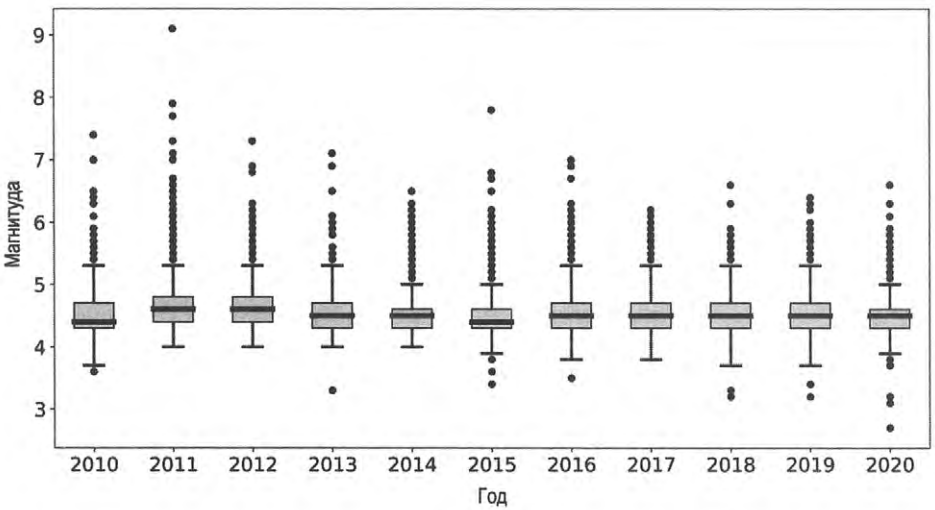


**Рис. 6.9.** Диаграмма размаха магнитуд землетрясений в Японии по годам

Хотя медиана и размер ящика немного меняются от года к году, они всегда находятся в пределах от 4 до 5. В Японии каждый год происходили сильные землетрясения, магнитуды которых превышали 6 баллов, а в шести годах зафиксировано, по крайней мере, одно землетрясение магнитудой 7 баллов и более. Япония, несомненно, очень сейсмически активный регион.

Гистограммы, диаграммы рассеяния и диаграммы размаха очень часто используются для обнаружения выбросов в наборах данных. Они позволяют нам быстро оценить большой объем данных и начать разбираться в их истории. Наряду с сорти-

ровкой, процентилями и стандартными отклонениями, графики являются важной частью набора инструментов для выявления аномалий. Имея на руках эти инструменты, мы можем приступить к изучению различных видов аномалий, в дополнение к тем, которые мы видели до сих пор.

## 6.4. Виды аномалий

Аномалии могут принимать самые разные формы и размеры. В этом разделе я рассмотрю три основных вида аномалий: значения, количества (или частоты), а также наличие или отсутствие данных. Это отправные точки для исследования любого набора данных в рамках профилирования или потому, что подозревается наличие аномалий. Выбросы и другие необычные значения часто специфичны для конкретной предметной области, поэтому чем больше вы знаете о том, как и почему были сгенерированы данные, тем лучше. Однако всегда можно начать свои исследования с этих методов выявления аномалий.

### Аномальные значения

Наверное, самый распространенный тип аномалий и первое, что приходит на ум, — это когда единичные значения являются либо экстремально высокими, либо экстремально низкими, либо когда значения в середине распределения являются очень необычными.

В предыдущем разделе мы рассмотрели несколько способов поиска таких аномалий: с помощью сортировки, процентилей и стандартных отклонений, а также с помощью графиков. Мы выяснили, что в нашем наборе данных о землетрясениях есть как необычно большие значения магнитуд, так и некоторые значения, которые кажутся слишком маленькими. Магнитуды также содержат разное количество *значащих цифр* или десятичных разрядов. Например, мы можем проверить выборку значений магнитуды около 1 и попробовать найти шаблон, который используется во всем наборе данных:

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

Время от времени встречается значение с восемью знаками после десятичной запятой. Многие значения имеют два десятичных разряда, но чаще встречается только один знак после запятой. Вероятно, это связано с разным уровнем точности приборов, измеряющих магнитуду. Кроме того, в базе данных не отображается второй знак после запятой, если он равен нулю, поэтому `1.10` отображается просто как `1.1`. Однако большое количество записей с `1.1` указывает на то, что это не просто проблема отображения. В зависимости от цели анализа мы можем с помощью округления корректировать значения так, чтобы все они имели одинаковое количество десятичных знаков, или ничего здесь не менять.

Кроме поиска аномальных значений, как правило, требуется определить, почему они возникли, или найти другие атрибуты, которые коррелируются с ними. Именно здесь нам необходим творческий подход и детективная работа с данными. Например, 1215 записей в наборе данных имеют очень высокие значения глубины, превышающие 600 км. Мы можем узнать, где возникли эти аномалии или как они были собраны. Давайте посмотрим на источник данных, который записан в поле `net` (от *network* — сеть):

```
SELECT net, count(*)
FROM earthquakes
WHERE depth > 600
GROUP BY 1
;

net  count
---  -----
us   1215
```

На сайте USGS указано, что `'us'` — это Национальный центр информации о землетрясениях (USGS National Earthquake Information Center, PDE). Однако это не очень информативно, поэтому давайте проверим значения `place`, которые содержат место землетрясений:

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

Видно, что многие из этих очень глубоких землетрясений происходят вокруг острова Ндой на Фиджи. Однако место включает в себя расстояние и направление, например `'100km NW of'` (100 км к северо-западу от), что затрудняет обобщение. Мы можем применить разбор текста, чтобы упростить текст и сосредоточиться только на упоминании самого места. Для значений, которые содержат расстояние и направление, а затем предлог `' of '` и еще некоторый текст, разделим строку по `' of '` и возьмем вторую часть:

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

Теперь мы можем с большей уверенностью утверждать, что большинство очень глубоких землетрясений были зарегистрированы в районе Фиджи, с концентрацией вокруг небольшого вулканического острова Ндой. Можно продолжать усложнять анализ, например снова разобрать текст и сгруппировать все землетрясения по более крупным регионам, чтобы показать, что кроме Фиджи другие глубокие землетрясения были зарегистрированы около Вануату и Филиппин.

Аномалии могут проявляться из-за опечаток, разного регистра символов или текстовых ошибок. Легкость их обнаружения зависит от количества уникальных значений или *кардинальности* поля. Различия в регистре можно проверить путем подсчета уникальных значений и уникальных значений с применением функции `lower` или `upper`:

```
SELECT count(distinct type) as distinct_types
,count(distinct lower(type)) as distinct_lower
```

```
FROM earthquakes
;

distinct_types  distinct_lower
--------------  --------------
25              24
```

Существуют 25 уникальных значений поля `type`, но без учета регистра — 24. Чтобы найти это отличающееся значение, мы можем использовать флаг и отметить те значения, чья строчная форма не совпадает с фактическим значением. Добавление количества записей для каждого уникального значения поможет определиться, что в дальнейшем мы хотим делать с этими значениями:

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

Аномальное значение `'Ice quake'` легко обнаружить, поскольку это единственное значение, для которого флаг равен `false`. Так как существует только одна запись с этим значением, по сравнению с 10 136 записями в строчной форме, мы можем предположить, что это опечатка и ее можно отнести к этим 10 136 записям. Можно попробовать применить и другие текстовые функции, например `trim`, если мы предполагаем, что значения содержат лишние пробелы, или `replace`, если некоторые слова имеют несколько написаний, например число `'2'` и слово `'two'`.

Сложнее обнаружить ошибки в написании слов, чем варианты написания. Если существует набор правильных значений и написаний, то его можно использовать для проверки данных либо с помощью `OUTER JOIN`, либо с помощью оператора `CASE` в сочетании со списком `IN`. В любом случае, цель состоит в том, чтобы выделить неожиданные или недопустимые значения. В отсутствие такого набора правильных значений нам, как правило, остается либо применять знания предметной области, либо делать обоснованные предположения. В таблице `earthquakes` мы можем прове-

рить значения поля type, которые используются всего лишь в нескольких записях, а затем попытаться определить, есть ли другие, более распространенные значения, на которые их можно заменить:

```
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
```

Ранее мы уже нашли 'Ice Quake' и решили, что это, скорее всего, то же самое, что и 'ice quake'. Для 'rockslide' (обвал) имеется только одна запись, хотя мы можем считать, что по смыслу это достаточно близко к 'landslide' (оползень), для которого есть 15 записей. Значение 'collapse' (обрушение) более неоднозначно, поскольку набор данных включает в себя как 'mine collapse' (обрушение шахты), так и 'building collapse' (обрушение здания). Будем ли мы что-то делать с такими записями или нет, зависит от целей анализа, о чем я расскажу далее в *разд. 6.5*.

## Аномальное количество или частота

Иногда аномалии проявляются не в виде единичных значений, а в виде закономерностей или групповой активности в данных. Например, клиент, потративший 100 до́лларов в интернет-магазине, не считается исключением, но тот же клиент, тративший по 100 долларов каждый час на протяжении 48 часов, почти наверняка является аномалией.

Существует ряд параметров, по которым групповую активность можно считать аномальной, причем многие из этих параметров зависят от самих данных. Информация о времени и местоположении есть во многих наборах данных, в том числе и

в нашем наборе данных о землетрясениях, поэтому я буду использовать эти поля в качестве примеров в этом разделе. Помните, что эти же методы часто можно применить и к другим атрибутам.

События, которые происходят с необычной частотой в течение короткого промежутка времени, могут указывать на аномальную активность. Это может быть как хорошим сигналом, например, когда какая-то знаменитость неожиданно упомянула ваш продукт, и это привело к всплеску его продаж. Но это может быть и плохим сигналом, когда, например, необычные всплески свидетельствуют о мошенничестве с кредитной картой или попытках вывести сайт из строя с помощью избыточного трафика. Чтобы разобраться в этом типе аномалий и понять, есть ли отклонения от нормального тренда, мы сначала применяем соответствующие агрегирования, а затем используем методы, приведенные ранее в этой главе, а также методы анализа временных рядов, рассмотренные в *гл. 3*.

В следующих примерах я разберу шаг за шагом несколько запросов, которые помогут нам понять обычные закономерности и выявить необычные. Это итеративный процесс, в котором на каждом шаге используются профилирование данных, знание предметной области и результаты предыдущих запросов. Мы начнем с проверки количества землетрясений по годам, выделив из поля `time` год и подсчитав количество записей. Если ваша база данных не поддерживает функцию `date_trunc`, используйте вместо нее `extract` или `trunc`:

```
SELECT date_trunc('year',time)::date as earthquake_year
,count(*) as earthquakes
FROM earthquakes
GROUP BY 1
;

earthquake_year  earthquakes
---------------  -----------
2010-01-01       122322
2011-01-01       107397
2012-01-01       105693
2013-01-01       114368
2014-01-01       135247
2015-01-01       122914
2016-01-01       122420
2017-01-01       130622
2018-01-01       179304
2019-01-01       171116
2020-01-01       184523
```

Мы видим, что в 2011 и 2012 гг. количество землетрясений было ниже, чем в другие года. В 2018 г. произошло резкое увеличение числа записей, которое сохранялось в течение 2019 и 2020 гг. Это кажется необычным, и мы можем предположить,

что Земля вдруг стала более сейсмически активной или что в данных есть ошибки, например дублирование записей, или что изменился процесс сбора данных. Давайте рассмотрим данные на месячном уровне, чтобы понять, сохраняется ли эта тенденция на более мелких временных периодах:

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

Результаты показаны на рис. 6.10. Мы видим, что хотя количество землетрясений варьируется от месяца к месяцу, наблюдается общий рост, начиная с 2017 г. Мы также видим, что есть три выброса: апрель 2010 г., июль 2018 г. и июль 2019 г.

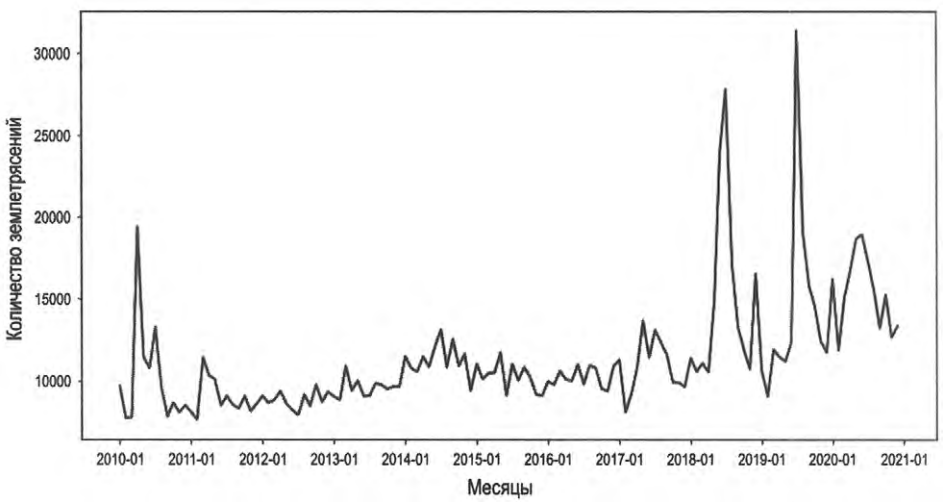


**Рис. 6.10.** Количество землетрясений по месяцам

Далее мы можем продолжить проверку данных на более коротких временных периодах, дополнительно отфильтровав записи по диапазону дат, чтобы сосредоточиться на этих необычных временных интервалах. После сужения временного диапазона до конкретных дней или даже времени суток для точного определения, когда именно произошли выбросы, мы можем разбить данные на части по другим атрибутам. Это поможет объяснить аномалии или, по крайней мере, сузить условия,

при которых они возникли. Например, оказалось, что увеличение количества землетрясений, начавшееся в 2017 г., может быть, по крайней мере частично, объяснено полем `status`. Статус показывает, было ли событие проверено человеком (`'reviewed'`) или автоматически опубликовано системой (`'automatic'`):

```
SELECT date_trunc('month',time)::date as earthquake_month
,status
,count(*) as earthquakes
FROM earthquakes
GROUP BY 1,2
ORDER BY 1
;

earthquake_month  status     earthquakes
----------------  ---------  -----------
2010-01-01        automatic  620
2010-01-01        reviewed   9031
2010-02-01        automatic  695
...               ...        ...
```

Динамика изменения количества землетрясений со статусами `'automatic'` и `'reviewed'` показана на рис. 6.11.

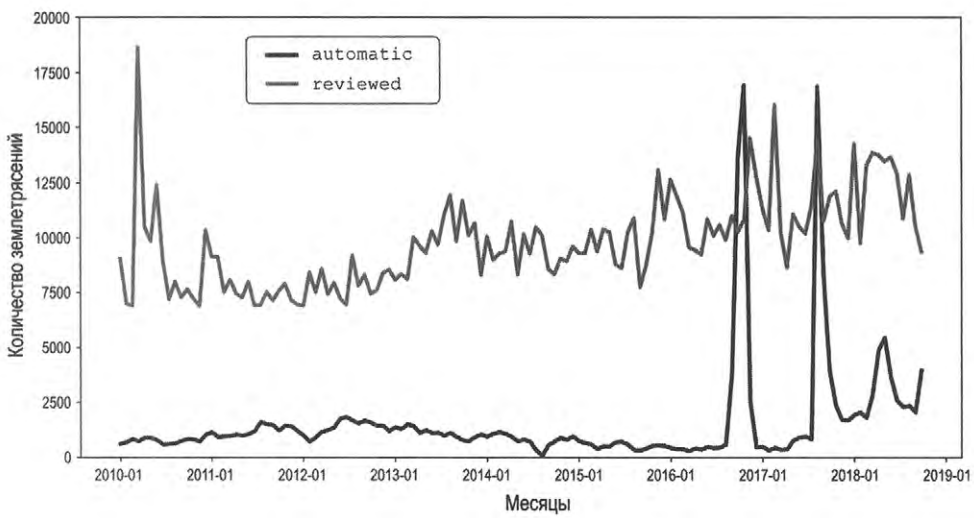


**Рис. 6.11.** Динамика количества землетрясений по месяцам, в зависимости от статуса проверки

На графике видно, что выбросы в июле 2018 г. и июле 2019 г. связаны со значительным увеличением числа землетрясений со статусом `'automatic'`, в то время как пик в апреле 2010 г. был из-за землетрясений со статусом `'reviewed'`. Возможно, в

2017 г. появился новый тип оборудования для автоматической регистрации землетрясений или не было достаточно времени для проверки записей.

Анализ местоположения в наборах данных, содержащих такую информацию, может стать еще одним эффективным способом поиска аномалий. Таблица `earthquakes` содержит огромное количество записей об очень мелких землетрясениях, что мешает нам анализировать очень крупные, заслуживающие особого внимания землетрясения. Давайте посмотрим на местоположения самых сильных землетрясений, с магнитудой 6 и более, и выясним, где они группируются географически:

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

В отличие от времени, когда мы выполняли запросы, постепенно уточняя и уменьшая временные интервалы, значения в поле `place` и так сильно детализированы, что трудно увидеть полную картину, хотя японский регион Хонсю уже явно выделяется. Мы можем применить некоторые методы текстового анализа из *гл. 5* для разбора и последующей группировки географической информации. В данном случае мы будем использовать функцию `split_part` для удаления текста с указанием направления (например, `'near the coast of '` или `'100km N of '`), который часто встречается в начале значений `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
...                    ...
```

В регионе вокруг Хонсю (Япония) произошло 89 землетрясений, что делает его не только местом самого крупного землетрясения в наборе данных, но и выбросом по количеству зарегистрированных сильных землетрясений. Мы могли бы продолжить разбор, очистку и группировку текстовых значения `place`, чтобы получить более общую картину того, где происходят самые крупные землетрясения в мире.

Поиск аномальных количеств, сумм или частот в наборе данных обычно включает в себя несколько последовательных шагов с созданием запросов разного уровня детализации. Обычно начинают с общей картины, а затем переходят к деталям, уменьшая масштаб для сравнения тенденций и опять увеличивая его на конкретных интервалах значений. К счастью, SQL — отличный инструмент для выполнения таких быстрых итераций. Объединяя методы анализа временных рядов, о котором говорилось в *гл. 3*, и методы текстового анализа, которому посвящена *гл. 5*, можно сделать анализ более полным и всесторонним.

## Аномальное отсутствие данных

Мы уже видели, что необычно высокая частота событий может сигнализировать об аномалии. Но отсутствие записей также может быть признаком аномалий. Например, когда ведется мониторинг сердцебиения пациента, перенесшего операцию, то отсутствие сердцебиения в любой момент времени поднимет тревогу. Однако во многих случаях обнаружить отсутствие данных сложно, если только не проверять это специально. Клиенты не всегда объявляют о том, что они собираются уйти к другому поставщику. Они просто перестают пользоваться продуктом или услугой и тихо пропадают из набора данных.

Один из способов гарантировать, что отсутствие данных будет замечено, — использовать методы когортного анализа, о которых говорилось в *гл. 4*. В частности, соединение `JOIN` с размерной таблицей дат, чтобы обеспечить наличие записей для всех сущностей, независимо от того, присутствовали ли они в заданный временной период или нет, помогает обнаружить отсутствие данных.

Еще один способ выявления отсутствия — это определение временных разрывов (gap), т. е. времени с момента последнего присутствия. В некоторых географических регионах чаще регистрируются сильные землетрясения из-за расположения тектонических плит на земном шаре. Мы также определили некоторые из таких регионов по набору данных в предыдущих примерах. Землетрясения, как известно, трудно предсказать, даже если мы знаем, где они могут происходить. Это не мешает некоторым людям строить предположения о следующем «большом землетрясении» просто на основании промежутка времени, прошедшего с момента последнего

крупного землетрясения. Мы можем использовать SQL, чтобы найти временные промежутки между крупными землетрясениями и время, прошедшее с момента последнего землетрясения:

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

В самом внутреннем подзапросе выполняются разбор и очистка текстового поля `place`, чтобы получить более крупные регионы или страны, а также возвращается время `time` для всех землетрясений магнитудой 5 и более. Во втором подзапросе для каждого места и времени определяются время следующего землетрясения `next_time`, если таковое имеется, с помощью функции `lead`, а также интервал `gap` между следующим землетрясением и текущим. Оконная функция `max` возвращает самое последнее землетрясение для каждого места. Внешний запрос вычисляет количество дней, прошедших с момента последнего землетрясения магнитудой 5 и более в наборе данных, используя функцию `extract` для возврата только дней из временного интервала, который получен при вычитании двух дат. Поскольку набор данных включает в себя записи только до конца 2020 г., используется временная метка `'2020-12-31 23:59:59'`, хотя при постоянном обновлении данных можно указать `current_timestamp` или любое эквивалентное выражение. Аналогичным образом извлекаются дни из среднего и максимального значений интервала `gap`.

Время, прошедшее с момента последнего сильного землетрясения в определенном месте, на практике имеет незначительную предсказательную силу, но во многих областях определение разрывов и времени с момента последнего присутствия может иметь практическое применение. Знание типичных промежутков между действиями позволяет определить норму, с которой можно сравнивать текущее значение временного разрыва. Если текущий разрыв находится в пределах исторических значений, мы можем считать, что клиент еще не ушел, но если он намного больше, возрастает риск оттока. Результаты запроса, возвращающего исторические разрывы, могут сами стать предметом анализа для выявления аномалий, помогая отвечать на разные вопросы, например, каким был самый продолжительный период времени, в течение которого клиент отсутствовал, а затем вернулся.

## 6.5. Обработка аномалий

Аномалии могут возникать в наборах данных по разным причинам и могут быть разных видов, как мы видели ранее. После обнаружения аномалий следующим шагом является их обработка. Что с ними делать, зависит как от причины возникновения аномалии (например, проблемы с самим процессом или с качеством данных), так и от конечной цели набора данных или анализа. При этом возможны следующие варианты: исследование без изменений, исключение, замена, изменение масштаба и исправление на более ранних этапах.

### Исследование аномалий

Поиск или даже попытка найти причину аномалии обычно является первым шагом в принятии решения о том, что с ней делать. Эта часть процесса может как увлечь, так и разочаровать — увлечь в том смысле, что поиск и решение этой загадки задействуют все наши навыки и творческие способности, а разочаровать, потому что мы часто работаем в условиях нехватки времени, и поиск аномалий может пока-

заться бесконечной чередой кроличьих нор и привести нас к мысли, не является ли весь анализ ошибочным.

Когда я исследую аномалии, то обычно выполняю серию запросов, постоянно переключаясь между поиском закономерностей и рассмотрением конкретных примеров. Настоящее аномальное значение легко обнаружить. В таких случаях я обычно проверяю всю запись, содержащую аномалию, для получения полной информации о времени, источнике и любых других доступных атрибутах. Затем я проверяю записи, которые содержат такие же атрибуты, чтобы узнать, есть ли в них значения, которые кажутся необычными. Например, я могу проверить, какие значения имеют другие записи за тот же день — нормальные или необычные. Проверка трафика с того же веб-сайта или покупок того же товара может выявить другие аномалии.

После изучения источника и атрибутов аномалий в данных, собранных внутри моей организации, я связываюсь с заинтересованными лицами или производителями товаров. Иногда это известная ошибка или недостаток, но довольно часто в процессе или в системе обнаруживается реальная проблема, требующая решения, и моя контекстная информация оказывается полезной. Для внешних или публичных наборов данных вряд ли можно найти первопричину аномалий. В таких случаях моя цель — собрать достаточно информации, чтобы решить, какой из следующих способов обработки аномалий выбрать.

## Исключение аномальных записей

Один из вариантов обработки аномалий — просто удалить их из набора данных. Если есть основания подозревать, что при сборе данных была допущена ошибка, которая может повлиять на результаты, удаление будет уместным. Удаление также является хорошим вариантом, когда набор данных достаточно велик, и исключение нескольких записей из рассмотрения вряд ли повлияет на результаты. Еще одна веская причина для удаления — это когда выбросы так экстремальны, что могут исказить результаты настолько, что это приведет к совершенно неуместным выводам.

Ранее мы видели, что набор данных о землетрясениях содержит несколько записей с магнитудами –9.99 и –9. Поскольку землетрясения, которым соответствуют эти значения, чрезвычайно малы, можно предположить, что это ошибочные значения или они просто были введены, когда фактическая магнитуда была неизвестна, а надо было зафиксировать событие. Исключение записей с такими значениями реализуется просто с помощью фильтра `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
```

Однако прежде чем исключить такие записи, можно попробовать определить, действительно ли выбросы влияют на результаты. Например, мы можем узнать, как удаление выбросов повлияет на среднее значение магнитуды, поскольку средние значения могут быть искажены выбросами. В наших силах легко сделать это, вычислив среднее значение по всему набору данных и среднее значение без учета экстремально низких значений магнитуды, используя оператор CASE для их исключения:

```
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
```

Средние значения различаются только в третьем знаке после запятой (1.625 против 1.627), что является довольно небольшим отклонением. Однако если отфильтровать землетрясения только в Йеллоустонском национальном парке, где зафиксировано много магнитуд –9.99, отличие будет более заметным:

```
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
```

Хотя эти значения магнитуд и небольшие, но разница между средними значениями 0.406 и 0.923 достаточно велика, поэтому мы, скорее всего, приняли бы решение об удалении выбросов.

Обратите внимание, что это можно сделать двумя способами: либо в разделе WHERE с удалением выбросов из всех результатов, либо с помощью оператора CASE, который исключит их только из определенных вычислений. Выбор способа зависит и от контекста анализа, и от того, нужно ли учитывать их при подсчете общего количества строк и пригодятся ли значения в других полях таких записей.

## Замена на альтернативные значения

Очень часто аномальные значения можно заменить на другие, не исключая такие записи из набора данных. Альтернативное значение может быть значением по умолчанию, заменяющим значением, ближайшим числовым значением в диапазоне или сводной статистикой (средним значением или медианой).

Ранее мы видели, что `null`-значения можно заменить значением по умолчанию с помощью функции `coalesce`. Если значения не обязательно равны `null`, но являются проблемными по какой-то другой причине, можно использовать оператор `CASE` для замены на значение по умолчанию. Например, чтобы не учитывать различные сейсмические события, которые не являются землетрясениями, мы можем сгруппировать их в отдельный тип `'other'` (прочее):

```
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
```

Конечно, это сокращает детализацию данных, но это также может быть способом обобщения набора данных, в котором есть несколько оишбочных значений `type`, как мы видели ранее. Когда вы знаете, что такие выбросы неверны, и вам известно правильное значение, можно заменить их с помощью оператора `CASE`, что сохранит строку в общем наборе данных. Например, в конец значения случайно мог быть добавлен лишний ноль или значение могло быть записано в дюймах вместо миль.

Другой вариант обработки числовых выбросов заключается в замене экстремальных значений на ближайшее большее или меньшее значение, которое не является экстремальным. Такой подход позволяет сохранить большую часть диапазона значений, но предотвращает появление вводящих в заблуждение средних значений, которые могут возникнуть из-за экстремальных выбросов. *Винсоризация* — это специальный метод, при котором выбросы заменяются на определенный процентиль да́нных. Например, значения выше 95-го процентиля заменяются на значение 95-го процентиля, а значения ниже 5-го процентиля — на значение 5-го процентиля. Чтобы рассчитать это с помощью SQL, мы сначала вычислим значения 5-го и 95-го процентилей:

```
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
```

Мы можем поместить этот расчет в подзапрос, а затем использовать оператор CASE для установки значений для выбросов ниже 5-го процентиля и выше 95-го. Обратите внимание на декартово соединение, которое позволяет нам сравнивать каждую отдельную величину со значениями процентилей:

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

Значение 5-го процентиля равно 0.12, а 95-го процентиля — 4.5. Значения магнитуды ниже и выше этих пороговых значений заменяются на пороговое значение в поле `mag_winsorize`. Значения между этими процентилями остаются неизменными. В методе винсоризации нет фиксированных процентилей. Можно использовать 1-й и 99-й процентили или даже 0.01-й и 99.9-й процентили в зависимости от требований к анализу и от того, насколько распространены и экстремальны значения выбросов в наборе данных.

## Изменение масштаба

Вместо того чтобы отфильтровывать записи или изменять значения выбросов, можно изменить масштаб значений, что обеспечит сохранение всех значений, но упростит анализ и построение графиков.

Ранее мы уже вычисляли z-оценку, но стоит отметить, что ее можно использовать как способ изменения масштаба значений. Ее удобно использовать, потому что она применима как к положительным, так и к отрицательным значениям.

Еще одно распространенное преобразование — перевод в логарифмическую (log) шкалу. Преимущество такого метода заключается в том, что значения сохраняют ту же очередность, но малые числа распределяются шире. Логарифмические величины могут быть преобразованы обратно в исходную шкалу, что упрощает обратную интерпретацию. Недостатком является то, что логарифмическое преобразование нельзя использовать для отрицательных чисел. В наборе данных о землетрясениях `earthquakes` мы знаем, что магнитуда уже выражена в логарифмической шкале. Великое землетрясение Тохоку магнитудой 9.1 является экстремальным, но если бы это значение уже не было выражено в логарифмической шкале, оно казалось бы еще более аномальным!

Поле `depth` измеряется в километрах. Давайте получим распределение как обычных значений глубины, так и с применением функции `log`, а затем построим графики на рис. 6.12 и 6.13, чтобы увидеть разницу. Функция `log` по умолчанию вычисляет логарифм по основанию 10. Чтобы уменьшить набор результатов для упрощения построения графика, глубину также округлим до одного знака после запятой с помощью функции `round`. Исключим также значения глубины меньше 0.05, т. к. они будут округляться до нуля:

```
SELECT round(depth,1) as depth
,log(round(depth,1)) as log_depth
,count(*) as earthquakes
FROM earthquakes
WHERE depth >= 0.05
GROUP BY 1,2
;
```

```
depth  log_depth             earthquakes
-----  -------------------   -----------
0.1    -1.0000000000000000   6994
0.2    -0.6989700043360188   6876
0.3    -0.5228787452803376   7269
...    ...                   ...
```

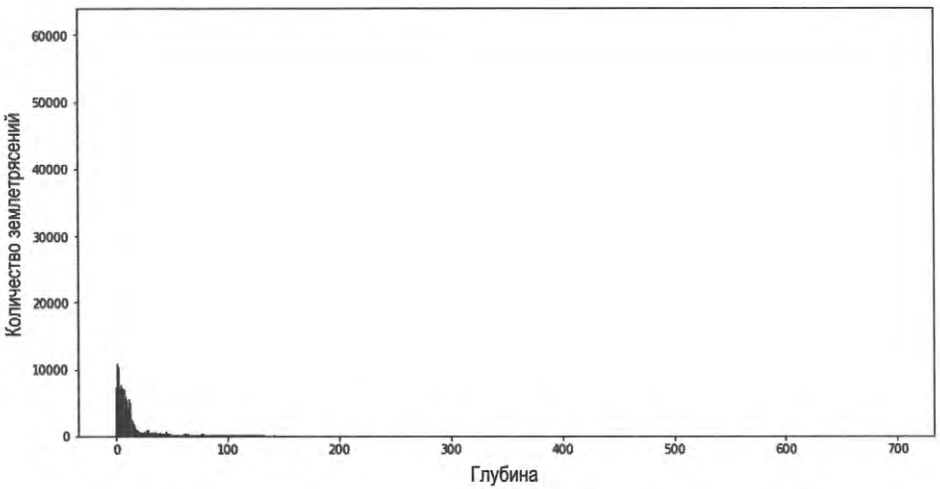


**Рис. 6.12.** Распределение землетрясений по глубине в обычной шкале

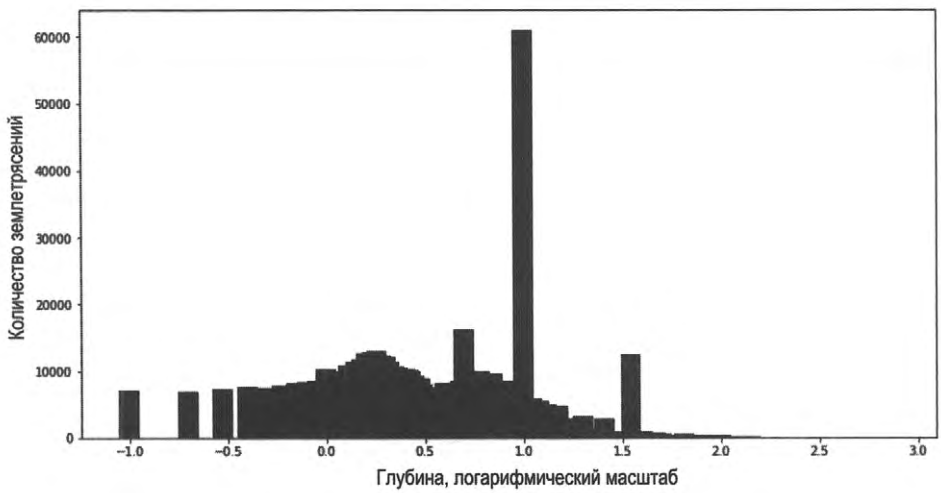


**Рис. 6.13.** Распределение землетрясений по глубине в логарифмической шкале

На рис. 6.12 видно, что существует большое количество землетрясений в диапазоне глубин от 0.05 до, возможно, 20, но за пределами этого диапазона трудно увидеть распределение, поскольку масштаб оси *x* растянут до 700, чтобы охватить весь диа-

пазон данных. Однако если глубину преобразовать в логарифмическую шкалу, как показано на рис. 6.13, распределение меньших значений видно гораздо лучше. Любопытно, что есть всплеск на 1.0, который соответствует глубине 10 км.

Существуют и другие преобразования масштаба, которые не всегда подходят для работы с выбросами, но могут быть выполнены с помощью SQL. Самые распространенные из них следующие:

- квадратный корень — с помощью функции `sqrt`;
- кубический корень — с помощью функции `cbrt`;
- обратное преобразование — простое деление `1/field_name`.

Перевод единиц измерения, например дюймов в футы или фунтов в килограммы, выполняется с помощью умножения или деления на соответствующие коэффициенты.

Изменение масштаба может быть выполнено в коде SQL или, как вариант, в программном обеспечении или на языке программирования, которые используются для построения графиков. Логарифмическое преобразование особенно полезно, когда есть большой разброс положительных значений, а закономерности, которые нужно проанализировать, находятся в более маленьких значениях.

Как и в любом анализе, решение о том, что делать с аномалиями, зависит от целей и объема анализа или от ваших знаний о наборе данных. Исключение выбросов — самый простой метод, но чтобы сохранить все записи, хорошо работают такие методы, как винсоризация и изменение масштаба.

## 6.6. Заключение

Выявление аномалий — распространенная практика в анализе данных. Целью может быть обнаружение выбросов или их обработка для подготовки набора данных к дальнейшему анализу. В любом случае такие простые методы, как сортировка, расчет процентилей и построение графиков по результатам SQL-запросов, помогут вам эффективно найти выбросы. Аномалии бывают разных видов, но наиболее распространеными являются единичные значения, странные всплески активности и необычные отсутствия. Знание предметной области почти всегда помогает в процессе поиска и сбора информации о причинах аномалий. Способы обработки аномалий включают в себя исследование, удаление, замену альтернативными значениями и изменение масштаба данных. Выбор способа во многом зависит от вашей задачи, но любой из них может быть реализован с помощью SQL. В следующей главе мы рассмотрим эксперименты, цель которых — выяснить, отличается ли поведение изучаемой группы от контрольной группы.

# ГЛАВА 7
# Анализ экспериментов

*Анализ экспериментов*, также известный как *A/B-тестирование* или *сплит-тестирование*, считается основным способом установления причинно-следственных связей. Очень многие задачи в анализе данных связаны с установлением корреляций: что-то может произойти с большей вероятностью, если уже произошло что-то другое, и это может касаться действий, характеристик или сезонного поведения. Однако вы наверняка слышали высказывание «корреляция не означает причинно-следственную связь» (correlation does not imply causation), и именно эту проблему в анализе данных пытаются решить с помощью экспериментов.

Все эксперименты начинаются с *гипотезы* — предположения об изменении поведения, которое произойдет в результате изменения товара, процесса или предложений. Изменения могут касаться пользовательского интерфейса, процесса регистрации нового пользователя, алгоритма, на котором основаны рекомендации, маркетинговых сообщений, сроков или многого другого. Если организация сама инициировала это изменение или имеет над ним полный контроль, с ним можно экспериментировать, по крайней мере, теоретически. Гипотезы часто возникают на основе результатов другого анализа данных. Например, мы обнаружили, что большой процент людей прерывают процесс оформления заказа, не дойдя до конца, и можно выдвинуть гипотезу о том, что люди будут чаще завершать этот процесс, если сократить количество шагов.

Вторая составляющая каждого эксперимента — это *целевая метрика* (success metric). Изменение поведения, которое мы выдвинули в качестве гипотезы, может быть связано с заполнением формы, количеством кликов, удержанием, вовлеченностью или любым другим поведением, которое важно для организации. Целевая метрика должна количественно измерять это поведение, быть достаточно простой в измерении и достаточно чувствительной к изменениям. Часто хорошими целевыми метриками являются количество кликов, завершение оформления заказа или время, необходимое для завершения процесса. Несмотря на то, что удержание и удовлетворенность клиентов очень важны, но они, как правило, не подходят в качестве целевой метрики, поскольку на них часто влияют многие факторы, выходящие за рамки отдельного эксперимента, и поэтому эти показатели менее чувствительны к изменениям, которые мы хотели бы проверить. Хорошими целевыми метриками часто являются те, которые вы уже отслеживаете для оценки деятельности компании или организации.

У вас может возникнуть вопрос: можно ли в эксперименте использовать несколько целевых метрик? Конечно, с помощью SQL можно реализовать множество вычислений для различных показателей. Однако вам следует помнить о проблеме множественных сравнений. Я не буду приводить здесь развернутое объяснение, но суть в том, что чем в большем количестве мест вы ищете существенные изменения, тем с большей вероятностью вы где-то их найдете. Проверяя одну метрику, вы можете увидеть или не увидеть ее существенного изменения в эксперименте. Однако если вы проверяете 20 метрик, есть большая вероятность, что хотя бы одна из них будет изменяться, независимо от того, имел ли эксперимент какое-то отношение к этой метрике вообще. Как правило, необходимо выбрать одну или, может быть, две целевые метрики. От одной до пяти дополнительных метрик могут быть использованы для защиты от ошибок. Их иногда называют *ограждающими метриками* (guardrail metric). Например, можно убедиться, что эксперимент не увеличил время загрузки страницы, даже если оно и не являлось целью эксперимента.

Третья составляющая эксперимента — это система, которая случайным образом относит сущность к контрольной группе или к вариантным группам эксперимента и соответствующим образом меняет для ее условия эксперимента. Некоторые поставщики программного обеспечения предлагают вспомогательные инструменты для проведения экспериментов, хотя иногда организации предпочитают создавать их своими силами, чтобы добиться большей гибкости. В любом случае для проведения анализа экспериментов с помощью SQL информация о принадлежности сущности к какой-то группе должна сохраняться в базу данных, где также лежат данные о действиях.

В этой главе обсуждаются именно онлайн-эксперименты, в которых распределение вариантов происходит с помощью компьютера, а поведение отслеживается в цифровом виде. Безусловно, существует множество видов экспериментов, проводимых в разных областях науки и обществознания. Главным отличием является то, что целевые метрики и поведение, которые исследуются в онлайн-экспериментах, как правило, уже записываются для каких-то других целей, в то время как во многих научных экспериментах поведение отслеживается специально для эксперимента и только в период его проведения. В онлайн-экспериментах иногда приходится творчески подходить к поиску метрик, которые были бы хорошими показателями действия, когда прямое измерение невозможно.

Имея гипотезу, целевую метрику и систему разделения на варианты, вы можете проводить эксперименты, собирать данные и анализировать результаты с помощью SQL.

## 7.1. Плюсы и минусы SQL для анализа экспериментов

SQL удобно использовать для анализа экспериментов. Во многих случаях данные о когортах участников эксперимента и об их поведении уже записываются в базу данных, что позволяет сразу применить SQL. Целевые метрики, как правило, уже

входят в отчетность организации и в проводимые анализы, и соответствующие SQL-запросы уже написаны. Добавить варианты к существующей логике запросов относительно просто.

SQL — хороший выбор для автоматизации отчетности о результатах экспериментов. Один и тот же запрос можно выполнить для каждого эксперимента, изменив название или идентификатор эксперимента в условии `WHERE`. Многие организации с большим количеством экспериментов создают стандартизированные отчеты, чтобы ускорить считывание данных и упростить процесс интерпретации результатов.

Хотя SQL удобно использовать на многих шагах анализа экспериментов, у него есть один существенный недостаток: SQL не умеет вычислять статистическую значимость. Многие базы данных позволяют разработчикам расширять функциональность SQL с помощью определения *пользовательских функций* (user-defined function). Пользовательские функции могут вызывать статистические тесты на других языках, например на Python, но это выходит за рамки данной книги. Хороший вариант — рассчитать сводную статистику на SQL, а затем использовать онлайн-калькулятор, например представленный на сайте **evanmiller.org**, чтобы определить, является ли результат эксперимента статистически значимым.

### Почему корреляция не то же самое, что причинно-следственная связь, и как величины могут соотноситься друг с другом

Легче доказать, что две величины коррелируют (растут или падают вместе, или одна существует только при наличии другой), чем доказать, что одна из них является причиной другой. Почему так происходит? Хотя наш мозг настроен на выявление причинно-следственной связи, на самом деле существуют пять способов, которыми две величины $X$ и $Y$ могут быть связаны друг с другом:

- *$X$ вызывает $Y$* — это то, что мы все пытаемся обнаружить. Благодаря какой-то закономерности, $Y$ является следствием $X$.
- *$Y$ вызывает $X$* — связь существует, но направление связи обратное. Например, не раскрытые зонтики вызывают дождь, а дождь заставляет людей использовать зонтики.
- *$X$ и $Y$ имеют общую причину* — величины связаны между собой, потому что существует некая третья величина, которая объясняет их обеих. Летом увеличиваются продажи мороженого и использование кондиционеров, но ни одно из этих явлений не является причиной другого. Повышение температуры воздуха вызывает рост обеих величин.
- *Петля обратной связи между $X$ и $Y$* — когда $X$ увеличивается, $Y$ увеличивается, чтобы компенсировать это, что, в свою очередь, приводит к повторному увеличению $X$, и т. д. Это может произойти, например, когда клиент начинает отказываться от услуги. Меньшее количество обращений со стороны клиента приводит к меньшему количеству предложений и напоминаний со стороны организации, что приводит к меньшему количеству обращений, и т. д. Являет-

ся ли отсутствие предложений причиной меньшего количества обращений, или все наоборот?

- *Между* величинами *нет никакой взаимосвязи* — это просто случайное совпадение. Если искать достаточно долго, то всегда можно найти величины, которые на первый взгляд коррелируют между собой, хотя на самом деле между ними нет никакой связи.

## 7.2. Набор данных о мобильной игре

В этой главе мы будем использовать набор данных о мобильной игре от вымышленной компании Tanimura Studios[1]. Имеются четыре таблицы. Таблица `game_users` содержит записи о людях, скачавших мобильную игру, с указанием даты и страны. Пример данных из этой таблицы показан на рис. 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 |

**Рис. 7.1.** Фрагмент таблицы `game_users`

Таблица `game_actions` содержит записи о действиях пользователей в игре. Пример данных показан на рис. 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 |

**Рис. 7.2.** Фрагмент таблицы `game_actions`

[1] **https://github.com/cathytanimura/sql_book/tree/master/Chapter 7: Experiment Analysis**

В таблице `game_purchases` записаны покупки внутриигровой валюты в долларах США. Пример данных показан на рис. 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 |

**Рис. 7.3.** Фрагмент таблицы `game_purchases`

Наконец, таблица `exp_assignment` содержит записи о том, к какой группе были отнесены пользователи в конкретном эксперименте. Пример данных показан на рис. 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 |

**Рис. 7.4.** Фрагмент таблицы `exp_assignment`

Все данные в этих таблицах вымышленные, созданы с помощью генератора случайных чисел, хотя аналогичную структуру вы можете увидеть в базе данных реальной компании, занимающейся онлайн-играми.

## 7.3. Типы экспериментов

Существует огромное количество разных экспериментов. Если вы можете изменить что-то, с чем взаимодействует пользователь, клиент или другая сущность, то теоретически вы можете протестировать это изменение. С точки зрения анализа, существуют два основных типа экспериментов: эксперименты с бинарными результатами и эксперименты с непрерывными результатами.

## Эксперименты с бинарными результатами: тест хи-квадрат

Как и следует из названия, эксперимент с бинарными результатами имеет только два возможных выхода: действие либо выполнено, либо нет. Пользователь завершил процесс регистрации или нет. Покупатель щелкнул по рекламе на сайте или нет. Студент закончил обучение или нет. В экспериментах этого типа для каждого варианта мы рассчитываем долю тех, которые завершили действие. Числитель — это количество тех, кто выполнил действие, а знаменатель — все, кто участвовал в группе. Такую метрику также называют *коэффициентом*: коэффициент завершения процесса, коэффициент кликабельности, коэффициент окончания и т. д.

Чтобы определить, различаются ли коэффеценты в вариантах статистически, мы можем использовать *тест хи-квадрат*, который является статистическим тестом для категориальных переменных[2]. Данные для теста хи-квадрат часто представляются в виде *таблицы сопряженности*, которая показывает частоту наблюдений на пересечении двух атрибутов. Она выглядит так же, как и сводная таблица, для тех, кто знаком с этим типом таблиц.

Давайте рассмотрим пример, используя наш набор данных о мобильной игре. Менеджер по продукту представил новую версию игрового обучения (onboarding) — серии экранов, которые поясняют новичку, как работает игра. Менеджер по продукту надеется, что новая версия увеличит количество игроков, которые пройдут процесс обучения и начнут свою первую игровую сессию. Эта новая версия обучения была использована в эксперименте под названием `'Onboarding'`, в котором пользователи были отнесены либо к контрольной группе `'control'`, либо к вариантной группе `'variant 1'`, что записывалось в таблицу `exp_assignment`. Событие с названием `'onboarding complete'` в таблице `game_actions` означает, что пользователь завершил процесс обучения.

В таблице сопряженности отражается частота для каждого варианта (`'control'` или `'variant 1'`) и для каждого события (был ли завершен процесс обучения или нет). Мы можем использовать запрос, чтобы найти значения для этой таблицы. Подсчитаем количество пользователей с действием `'onboarding complete'` и без него и выполним группировку по полю `variant`:

```
SELECT a.variant
,count(case when b.user_id is not null then a.user_id end) as completed
,count(case when b.user_id is null then a.user_id end) as not_completed
FROM exp_assignment a
LEFT JOIN game_actions b on a.user_id = b.user_id
 and b.action = 'onboarding complete'
WHERE a.exp_name = 'Onboarding'
GROUP BY 1
;
```

[2] Хорошее описание этого теста можно найти здесь: **https://www.mathsisfun.com/data/chi-square-test.html**.

```
variant    completed  not_completed
---------  ---------  -------------
control    36268      13629
variant 1  38280      11995
```

Если добавить итоговые значения для каждой строки и каждого столбца, то этот результат можно превратить в таблицу сопряженности, как показано на рис. 7.5.

| | Закончили обучение? | | |
|---|---|---|---|
| Вариант | Да | Нет | Итого |
| Контрольный | 36 268 | 13 629 | **49 897** |
| Вариант 1 | 38 280 | 11 995 | **50 275** |
| Итого | **74 548** | **25 624** | **100 172** |

**Рис. 7.5.** Таблица сопряженности для игроков, завершивших процесс обучения

Чтобы воспользоваться онлайн-калькулятором достоверности, нам понадобится количество раз, когда действие было выполнено, и общее число участников для каждого варианта. SQL для поиска необходимых нам значений прост. Варианты и количество пользователей, назначенных на каждый вариант, запрашиваются из таблицы `exp_assignment`. Затем мы присоединяем с помощью `LEFT JOIN` таблицу `game_actions`, чтобы найти количество пользователей, завершивших процесс обучения. Соединение `LEFT JOIN` необходимо, поскольку мы ожидаем, что не все пользователи выполнили соответствующее действие. Наконец, мы находим процент завершенных действий в каждом варианте путем деления количества пользователей, которые завершили действие, на размер группы:

```
SELECT a.variant
,count(a.user_id) as total_cohorted
,count(b.user_id) as completions
,count(b.user_id) * 100.0 / 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        72.69
variant 1  50275           38280        76.14
```

Видно, что в варианте 1 было больше тех, кто завершил обучение, чем в контрольном варианте: 76.14% по сравнению с 72.69%. Но является ли этот результат статистически значимым, позволяющим нам опровергнуть гипотезу об отсутствии разницы? Для этого мы вводим наши результаты в онлайн-калькулятор и получаем, что при уровне достоверности 95% статистическая значимость для варианта 1 значительно выше, чем для контрольного варианта. Вариант 1 можно объявить победителем в этом эксперименте.

Обычно используется уровень достоверности 95%, хотя это не единственно возможный вариант. В интернете ведется много дискуссий о смысле уровня достоверности, об используемом значении и о его корректировках в сценариях, в которых вы сравниваете несколько вариантов с контрольным.

Эксперименты с бинарными результатами следуют этой основной схеме. Подсчитайте количество успешных исходов или завершений, а также общее количество участников для каждого варианта. SQL, используемый для вычисления успешных событий, может быть более сложным в зависимости от структуры таблиц и от того, как действия записаны в базе данных, но результаты будут согласующимися. Далее мы перейдем к экспериментам с непрерывными результатами.

## Эксперименты с непрерывными результатами: *t*-тест

Многие эксперименты направлены на улучшение *непрерывных метрик*, а не бинарных результатов, о которых шла речь в предыдущем разделе. Непрерывные метрики могут принимать различные значения. В качестве примера можно привести сумму, потраченную покупателем, время, проведенное на странице, и количество дней использования приложения. Сайты электронной коммерции часто стремятся увеличить продажи, поэтому они могут начать экспериментировать со страницами товаров или процессом оформления заказа. Тематические сайты могут тестировать макет, навигацию и заголовки, чтобы увеличить количество прочитанных статей. Организация, выпускающая мобильное приложение, может запустить кампанию ремаркетинга, чтобы напомнить пользователям о приложении.

Для таких экспериментов с непрерывными метриками цель состоит в том, чтобы выяснить, отличаются ли средние значения в каждом варианте друг от друга со статистической достоверностью. Соответствующим статистическим тестом является *двухвыборочный t-тест*, который определяет, можем ли мы отклонить нулевую гипотезу о том, что средние значения равны, с заданным уровнем достоверности, обычно равным 95%. Статистический тест имеет три входных параметра, которые легко вычислить с помощью SQL: среднее значение, стандартное отклонение и количество наблюдений.

Давайте рассмотрим пример с использованием данных о мобильной игре. В предыдущем разделе мы рассмотрели, увеличил ли новый процесс обучения в игре коэф-

фициент завершивших обучение. Теперь мы посмотрим, увеличил ли этот новый процесс расходы пользователей на внутриигровую валюту. Целевой метрикой будет потраченная сумма, и нам нужно будет рассчитать среднее значение и стандартное отклонение для каждого варианта. Сначала нам нужно определить сумму, потраченную каждым пользователем, поскольку они могли совершить несколько покупок. Из таблицы `exp_assignment` извлекаем название когорты, к которой относится каждый пользователь, чтобы потом подсчитать размер когорты. Затем выполняем `LEFT JOIN` с таблицей `game_purchases`, чтобы собрать данные о потраченных суммах. Соединение `LEFT JOIN` необходимо, поскольку не все пользователи совершают покупки в игре, но нам все равно нужно включить их в расчеты среднего значения и стандартного отклонения. Если пользователь ничего не покупал, с помощью функции `coalesce` сумма по умолчанию устанавливается равной 0. Поскольку функции `avg` и `stddev` игнорируют `null`-значения, необходимо установить значение 0 по умолчанию, чтобы эти записи учитывались. Внешний запрос группирует результаты по вариантам:

```
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
```

Далее мы вводим эти значения в онлайн-калькулятор и обнаруживаем, что нет значительной разницы между контрольной группой и вариантом 1 при уровне достоверности 95%. Похоже, что для варианта 1 увеличилось количество завершивших обучение, но не количество потраченных средств.

Другой вопрос, который мы могли бы рассмотреть, — повлиял ли вариант 1 на расходы тех пользователей, которые завершили процесс обучения. Те, кто не завершил обучение, никогда не заходят в игру и, следовательно, не имеют возможности выполнить покупку. Чтобы ответить на этот вопрос, мы можем использовать запрос, аналогичный предыдущему, но мы добавим `JOIN` с таблицей `game_actions`, чтобы учитывать только тех пользователей, у кого есть действие `'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
```

Подставив эти значения в онлайн-калькулятор, можно увидеть, что среднее значение для контрольной группы статистически значимо выше, чем для варианта 1 при уровне достоверности 95%. Этот результат может показаться странным, но он иллюстрирует, почему так важно заранее согласовать целевую метрику для эксперимента. Вариант 1 оказал положительное влияние на завершение обучения, поэтому его можно считать успешным. И он не оказал никакого влияния на общий уровень расходов. Это может быть связано с изменением состава участников: дополнительные пользователи, прошедшие обучение в варианте 1, не собирались что-то тратить в игре. Если исходная гипотеза заключалась в том, что увеличение коэффициента завершивших обучение повысит доход, то эксперимент следует считать неуспешным, а менеджеру по продукту нужно придумать несколько новых идей для тестирования.

# 7.4. Спасение неудачных экспериментов

Хотя эксперименты отлично подходят для определения причинно-следственных связей, существует ряд причин, из-за которых они могут пойти по ложному пути. Если все предположения были ошибочными, то с помощью SQL нельзя ничего сделать, чтобы спасти положение. Если же недостаток носил технический характер, мы можем выбрать данные таким образом, чтобы скорректировать или совсем исключить проблемные точки и все же получить некоторые результаты. Проведение экспериментов требует временных затрат разработчиков, дизайнеров или маркетологов, кто будет создавать варианты. Также присутствуют и *альтернативные издержки* — упущенная выгода, которую можно было бы получить, направив клиентов по оптимальному пути конверсии или сформировав у них лучшее впечатление от продукта. С практической точки зрения, время, потраченное на то, чтобы с помощью SQL хоть что-то узнать из эксперимента, часто окупается.

## Система назначения вариантов

Случайное распределение сущностей, участвующих в эксперименте (это могут быть пользователи, сессии и проч.), на контрольные и вариантные группы является одной из ключевых составляющих эксперимента. Однако иногда в процессе распределения случаются ошибки, например из-за недостатков в описании эксперимента, технического сбоя или ограничений программного обеспечения для формирования когорт. В результате контрольная и вариантная группы могут оказаться разных размеров, или в когорту попадет меньшее количество сущностей, чем ожидалось, или распределение может оказаться неслучайным.

Иногда с помощью SQL можно спасти эксперимент, в котором в когорту попало слишком много сущностей. Я столкнулась с такой ситуацией, когда эксперимент предназначался только для новых пользователей, но вместо этого в когорту попали все пользователи. Еще один случай — когда в эксперименте тестируется что-то, что увидит только часть пользователей, потому что для этого надо сделать несколько дополнительных кликов или должны быть выполнены определенные условия, например сделана покупка. Из-за технических ограничений все пользователи попадут в когорту, хотя часть из них никогда не увидит то, что тестируется в эксперименте. Это можно решить, добавив лишний `JOIN` в SQL, что позволит ограничить пользователей и далее работать только с теми, кто соответствует нужным критериям. Например, в эксперименте с новыми пользователями мы можем добавить `INNER JOIN` к таблице или подзапросу, содержащему дату регистрации пользователя, и задать условие `WHERE`, чтобы исключить пользователей, которые зарегистрировались намного раньше событий эксперимента и не могут считаться новыми. Такую же стратегию можно применить, когда необходимо выполнить определенное условие, чтобы увидеть эксперимент. Ограничьте количество учитываемых сущностей с помощью `JOIN` и условий `WHERE`, исключив те, которые не соответствуют нужным критериям. После этого необходимо убедиться, что оставшиеся представляют собой достаточно большую выборку для получения достоверных результатов.

Если в когорту попало слишком мало пользователей или сущностей, важно проверить, достаточно ли этой выборки для получения достоверных результатов. Если нет, проведите эксперимент еще раз. Если размер выборки достаточно велик, то второй вопрос — есть ли предвзятость в том, кто или что было включено в когорту. Например, мне попадались случаи, когда пользователи определенных браузеров или старых версий приложений не попадали в когорту из-за технических ограничений. Если исключенная из рассмотрения аудитория не была случайной и в нее входят представители разных местоположений, технических навыков или социально-экономического статуса, важно оценить, насколько велика эта аудитория по сравнению с остальными и нужно ли внести какие-либо коррективы, чтобы включить их в окончательный анализ.

Другая возможная ситуация возникает тогда, когда система назначения вариантов несовершенна и сущности распределяются не случайным образом. Это довольно маловероятно для большинства современных инструментов для проведения экспериментов, но если такое произойдет, то это сделает недействительным весь эксперимент. Результаты, которые «слишком хороши, чтобы быть правдой», могут свидетельствовать о проблемах при назначении вариантов. Например, я сталкивалась со случаями, когда активно вовлеченные пользователи были случайно отнесены и к вариантной, и к контрольной группам из-за изменений в конфигурации эксперимента. Тщательное профилирование данных позволяет проверить, были ли сущности отнесены к нескольким вариантам или были ли пользователи с высокой или низкой вовлеченностью до начала эксперимента сгруппированы в одном варианте.

Проведение *A/A-тестирования* может помочь выявить недостатки в системе назначения вариантов. В этом тесте сущности объединяются в две когорты и сравниваются целевые метрики, как и в любом другом эксперименте. Однако никаких изменений в эксперимент не вносится, и обе когорты получают один контрольный вариант. Поскольку разницы в варинтах нет, мы не ожидаем существенных различий в целевых метриках. Если выяснится, что разница есть, необходимо провести дополнительное исследование, чтобы выявить и устранить эту проблему.

## Выбросы

Статистические тесты для анализа непрерывных целевых метрик опираются на их средние значения. В результате они чувствительны к необычно высоким или низким значениям выбросов. Я наблюдала эксперименты, в которых присутствие в варианте одного или двух покупателей, особенно много тратящих, давало этому варианту статистически значимое преимущество перед остальными. Без учета этих нескольких транжир результат был бы нейтральным или даже противоположным. В большинстве случаев нас больше интересует, оказывает ли эксперимент воздействие на группу обычных людей, и поэтому корректировка таких выбросов может сделать результаты эксперимента более значимыми.

Мы уже обсуждали выявление аномалий в *гл. 6*, и анализ экспериментов — это еще одна область, в которой можно применять эти методы. Выбросы могут быть опре-

делены либо при анализе результатов эксперимента, либо путем нахождения основного коэффициента до начала эксперимента. Выбросы можно удалить с помощью *винсоризации* (*см. гл. 6*), когда аномалии, превышающие пороговое значение, заменяются, например, на 95-й или 99-й процентиль. Это можно сделать в SQL, прежде чем переходить к остальному анализу эксперимента.

Другой способ обработки выбросов в непрерывных целевых метриках — преобразовать целевую метрику в бинарный результат. Например, вместо того чтобы сравнивать средние расходы, которые могут быть искажены из-за нескольких очень больших выбросов, сравните коэффициент покупок в контрольной и вариантной группах, а затем воспользуйтесь методом, описанным в *разд. 7.3* об экспериментах с бинарными результатами. Мы можем рассмотреть коэффициент конверсии в покупателя (совершившего покупки внутриигровой валюты) для пользователей, завершивших процесс обучения, для контрольной и вариантной групп из эксперимента `'Onboarding'`:

```
SELECT a.variant
,count(distinct a.user_id) as total_cohorted
,count(distinct b.user_id) as purchasers
,count(distinct b.user_id) * 100.0 / 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        10.00
variant 1  38280           4981        9.91
```

Из полученных результатов видно, что, несмотря на то, что в варианте 1 было больше пользователей, покупателей оказалось меньше. Процент пользователей, совершивших покупку в контрольной группе, составил 10%, по сравнению с 9.91% для варианта 1. Далее мы подставляем полученные данные в онлайн-калькулятор. Коэффициент конверсии для контрольной группы имеет более высокую статистическую значимость. В нашем случае, несмотря на то, что количество покупок было больше в контрольной группе, с практической точки зрения, мы можем согласиться с этим небольшим спадом для варианта 1, если считаем, что большее количество пользователей, прошедших процесс обучения, имеет другие преимущества. Например, большее количество игроков может повысить рейтинг игры, а игроки, которым она понравилась, расскажут о ней своим друзьям. Оба этих фактора могут спо-

собствовать росту известности и привлечению новых игроков, которые впоследствии могут стать покупателями.

Для целевой метрики также может быть установлено пороговое значение, и можно сравнивать долю сущностей, которые соответствуют этому пороговому значению. Например, целевой метрикой может быть прочтение не менее трех новостных сообщений или использование приложения не менее двух раз в неделю. Таким образом можно определить бесконечное количество метрик, поэтому нужно правильно оценивать, что важно и значимо для вашей организации.

## Метод временных рамок

Эксперименты часто проводятся в течение нескольких недель. Это означает, что люди, которые подключились к эксперименту раньше, имеют больше времени для выполнения действий, связанных с целевой метрикой. Чтобы контролировать это, мы можем применить *метод временных рамок* (time boxing) — установить фиксированное временное окно относительно даты поключения к эксперименту и рассматривать действия только внутри этого окна. Похожая концепция уже применялась в *гл. 4*.

Для экспериментов подходящий размер временного окна зависит от того, что вы измеряете. Оно может быть короче одного часа, если измеряется действие, которое обычно имеет немедленную реакцию, например клик на рекламе. Для измерения конверсии в покупку часто используют временной интервал от 1 до 7 дней. Поскольку всем сущностям в когортах надо дать полное время для выполнения действий, то более короткие временные окна позволяют анализировать эксперименты быстрее. Для оптимального окна необходимо найти баланс между быстрым получением результатов и реальной динамикой организации. Если клиенты обычно совершают покупки за несколько дней, используйте 7-дневное окно; если им часто требуется 20 и более дней, используйте 30-дневное окно.

В качестве примера мы можем повторно рассмотреть случай из раздела об экспериментах с непрерывными результатами, включив в него только покупки, сделанные в течение 7 дней после назначения варианта. Обратите внимание, что в качестве начальной точки временного окна важно использовать время, когда сущности был назначен вариант. В нашем примере добавится новое условие соединения, ограничивающее результаты покупками, которые сделаны в интервале `'7 days'`:

```
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
```

Средние значения очень похожи, и статистически они тоже существенно не отличаются друг от друга. В этом примере результаты метода временных рамок согласуются с результатами, когда временые рамки не использовались.

В нашем случае покупки происходят относительно редко. Для метрик, измеряющих частые события и те, которые накапливаются быстро, например просмотры страниц, клики, лайки и прочитанные статьи, использование временного интервала поможет избежать того, чтобы пользователи, раньше выбранные в когорту, выглядели значительно «лучше», чем те, кто был выбран позднее.

## Эксперименты с повторным воздействием

При рассмотрении онлайн-экспериментов большинство примеров относится к тому, что я люблю называть «одноразовыми» экспериментами, — пользователь сталкивается с воздействием один раз, реагирует на него и больше никогда не проходит этот путь. Классический пример — регистрация пользователя, которую он выполняет только один раз, и поэтому любые изменения в процессе регистрации коснутся только новых пользователей. Анализ таких экспериментов относительно прост.

Существует и другой тип экспериментов, который я называю «повторным воздействием», когда клиент сталкивается с изменениями много раз при использовании продукта или сервиса. В любом эксперименте с такими изменениями мы можем ожидать, что клиенты будут столкиваться с ними более одного раза. Изменения пользовательского интерфейса, например цвета, текста, размещения важной информации и ссылок, воздействуют на пользователей на протяжении всего времени работы с приложением. Маркетинговые программы, в которых клиентам регулярно отправляются электронные письма с напоминаниями или рекламными акциями, также обладают этим качеством многократного воздействия. Само электронное

письмо воздействует на получателя несколько раз — с помощью темы письма в почтовом ящике и с помощью содержания, если его открыли.

Оценка экспериментов с повторным воздействием сложнее, чем оценка экспериментов с однократным воздействием из-за эффекта новизны и регрессии к среднему. *Эффект новизны* — это тенденция к изменению поведения только потому, что что-то является новым, а не потому, что оно обязательно лучше. *Регрессия к среднему* — это тенденция возвращения к среднему уровню с течением времени. Например, изменение любой части пользовательского интерфейса приводит к увеличению числа людей, которые с ним взаимодействуют, будь то новый цвет кнопок, логотип или размещение функциональности. Сначала метрики выглядят хорошо, потому что количество переходов или вовлеченность повысились. Это эффект новизны. Но со временем пользователи привыкают к этим изменениям и, как правило, используют функциональность с частотой, возвращающейся к исходному уровню. Это и есть регрессия к среднему. Важный вопрос, на который необходимо ответить при проведении такого рода экспериментов, заключается в том, является ли новый уровень выше (или ниже) предыдущего. Одно из решений состоит в том, чтобы подождать, пока пройдет достаточно длительный период времени, в течение которого может появиться регрессия, прежде чем оценивать результаты. В некоторых случаях это займет несколько дней, в других — несколько недель или месяцев.

Когда изменений много или проводится серия экспериментов, например маркетинговая кампания по рассылке электронной и обычной почтой, оценить эффект от всей кампании целиком может быть непросто. Легко принять за успех действие, когда клиент, получивший определенный вариант электронного письма, покупает продукт, но как выяснить, что он не совершил бы эту покупку в любом случае? Один из способов — создать *глобальную контрольную группу*. Это группа клиентов, которым не отправляется никаких маркетинговых сообщений или изменений в продукте. Обратите внимание, что это отличается от простого сравнения с клиентами, которые специально отказались от маркетинговых сообщений, поскольку у них, как правило, уже сформировалась некоторая предвзятость. Формирование такой группы может быть сложным, но есть и другие методы оценки накопительного эффекта от маркетинговой кампании или изменений в продукте.

Другой способ — провести когортный анализ (*см. гл. 4*) вариантных групп. За группами можно следить в течение более длительного периода времени — от нескольких недель до нескольких месяцев. Можно рассчитать показатели удержания или накопительные итоги и проверить, есть ли разница между вариантами в долгосрочной перспективе.

Даже несмотря на различные трудности, которые могут возникнуть при проведении экспериментов, они по-прежнему являются лучшим способом проверить и доказать причинно-следственные связи при внесении изменений, начиная от маркетинговых сообщений и заканчивая взаимодействием с продуктом. Однако при анализе данных мы часто сталкиваемся с совсем неидеальными ситуациями, поэтому далее мы рассмотрим некоторые виды анализа, когда A/B-тестирование невозможно.

## 7.5. Альтернативные анализы, когда контролируемые эксперименты невозможны

Контролируемые эксперименты позволяют выйти за рамки корреляции и установить причинно-следственные связи. Однако существует ряд причин, по которым проведение эксперимента невозможно. Например, может быть неэтично предоставлять разное обслуживание разным группам, особенно в медицинских или образовательных учреждениях. Требования законодательства могут препятствовать проведению экспериментов в других областях, например в сфере финансовых услуг. Могут быть и практические причины, скажем, сложно обеспечить доступ к вариантному обслуживанию только для случайно набранной группы. Всегда стоит задуматься о том, все ли составляющие эксперимента, который нужно или можно провести, останутся в этических, правовых и практических рамках. Это может касаться, например, формулировок, расположения элементов дизайна и т. д.

Еще один случай, в котором эксперименты невозможны, — это когда изменение произошло в прошлом, и данные уже собраны. Отменить изменение, вернуться назад и провести эксперимент не представляется возможным. Иногда это происходит, потому что аналитик данных или специалист по данным не был доступен, чтобы вовремя проконсультировать по вопросам проведения экспериментов. Не раз мне приходилось работать в организациях, где меня просили разобраться в результатах изменений, которые было бы гораздо проще оценить, если бы существовала глобальная контрольная группа. В других случаях изменения произошли непреднамеренно. В качестве примера можно привести перебои в работе сайта, которые затронули некоторых или всех клиентов, ошибки в формах ввода, а также стихийные бедствия, такие как ураганы, землетрясения и лесные пожары.

Хотя выводы о причинно-следственных связях не столь убедительны в случаях, когда эксперимент не проводился, существует несколько квазиэкспериментальных методов анализа, которые можно использовать для извлечения пользы из таких данных. Они выполняются для групп, сформированных как можно ближе к «контрольным» и «экспериментальным» условиям, на основе имеющихся данных.

### Анализ «до и после»

В анализе «до и после» сравнивается одна и та же группа или схожие группы до и после изменений. Группа до изменения принимается в качестве контрольной, а группа после изменения — в качестве вариантной.

Анализ «до и после» лучше всего работает, когда есть четко определенное изменение, произошедшее в конкретную дату, так что группы «до» и «после» могут быть четко разделены. В этом типе анализа вам нужно будет выбрать, какие временные интервалы рассматривать до и после изменения, но они должны быть равны или почти равны. Например, если после изменения прошло две недели, сравните этот период с двумя неделями до изменения. Вы можете сравнить нескольких времен-

ных периодов, например одну неделю, две, три и четыре недели до и после изменения. Если результаты совпадают для всех этих периодов, вы можете быть более уверены в достоверности результатов, чем если бы они разнились.

Давайте рассмотрим пример. Представьте, что процесс регистрации в нашей мобильной игре включает в себя этап, на котором пользователь может установить флажок, указывающий на то, хочет ли он подписаться на рассылку новостей об игре по электронной почте. Этот флажок всегда был установлен по умолчанию, но теперь решено, что он должен быть снят по умолчанию. 27 января 2020 г. это изменение вышло с очередным обновлением игры, и мы хотели бы выяснить, оказало ли оно негативное влияние на коэффициент подписывающихся на рассылку. Для этого мы сравним две недели до изменения и две недели после него и посмотрим, изменился ли коэффициент подписавшихся. Мы могли бы использовать одну неделю или три недели в качестве временного периода, но две недели выбраны потому, что они достаточно длинные, чтобы сгладить некоторую изменчивость в зависимости от дня недели, и достаточно короткие, чтобы исключить другие факторы, которые могли бы повлиять на решение пользователей подписаться на рассылку.

Варианты назначаются в SQL-запросе с помощью оператора `CASE`: пользователей из таблицы `game_users`, которые были созданы за две недели до изменения, обозначаем как `'pre'`, а пользователей, созданных после изменения, — как `'post'`. Далее мы подсчитываем с помощью `count` количество пользователей в каждой группе. Затем мы подсчитываем количество пользователей, подписавшихся на рассылку, для этого выполняем `LEFT JOIN` с таблицей `game_actions`, учитывая только действия `'email_optin'`. Затем мы делим эти значения на размер группы, чтобы найти процент подписавшихся пользователей. Мне нравится включать количество дней для проверки результатов, хотя это значение не понадобится в остальной части анализа:

```
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) * 100.0 /
 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     58.75      14
post     27617     11220     40.63      14
```

Многие базы данных, в том числе и Postgres, распознают даты, введенные как строки, например `'2020-01-13'`. Если ваша база данных так не умеет, преобразуйте строку в дату, используя один из этих методов:

```
cast('2020-01-13' as date)
date('2020-01-13')
'2020-01-13'::date
```

В данном случае мы видим, что пользователи, прошедшие регистрацию до изменения, имеют гораздо более высокий коэффициент подписавшихся на рассылку — 58.75%, по сравнению с 40.63% после изменений. Подставив эти значения в онлайн-калькулятор, можно убедиться, что показатель для группы «до» статистически значимей, чем для группы «после». В данном примере компания, выпустившая игру, мало что может сделать, поскольку изменения были вызваны требованиями законодательства. Дальнейшие тесты помогут определить, повлияет ли предоставление образца письма или другой информации о рассылке на то, чтобы большее количество новых игроков подписывалось на рассылку, если это является целью бизнеса.

При проведении анализа «до и после» следует помнить, что другие факторы, помимо того изменения, который вы пытаетесь проанализировать, могут вызвать увеличение или уменьшение метрики. Внешние события, сезонность, маркетинговые акции и прочее могут кардинально изменить ситуацию и отношение клиентов даже в течение нескольких недель. В результате этот вид анализа не так хорош для доказательства причинно-следственных связей, как настоящий эксперимент. Тем не менее, иногда это один из немногих доступных видов анализа, и он может привести к появлению рабочих гипотез, которые можно будет проверить и уточнить в будущих контролируемых экспериментах.

## Анализ естественных экспериментов

*Естественный эксперимент* проводится, когда сущности получают различные воздействия в результате некоторого процесса, приближенного к случайному. Одна группа получает обычный или контрольный вариант, а другая — некоторые изменения, которые могут иметь положительный или отрицательный эффект. Обычно это происходит произвольным образом, например когда возникает ошибка в программном обеспечении или когда из нескольких мест событие случается толко в одном. Чтобы этот тип анализа был достоверным, мы должны четко знать, какие

сущности подверглись воздействию. Кроме того, контрольная группа должна быть максимально похожа на группу, подвергшуюся воздействию.

SQL можно использовать для формирования групп, определения их размеров и успешных событий в случае бинарных результатов или для вычисления среднего значения, стандартного отклонения и размера аудитории в случае непрерывных результатов. Статистическую достоверность можно проверить в онлайн-калькуляторе, как и для любого другого эксперимента.

В качестве примера с набором данных о мобильной игре представьте, что в течение всего периода времени пользователям в Канаде на странице покупки виртуальной валюты при первом просмотре случайно выводилось другое предложение: к количеству виртуальных монет в каждом пакете добавлялся дополнительный нолик. То есть, например, вместо 10 монет пользователь видел 100 игровых монет, вместо 100 монет — 1000, и т. д. Вопрос, на который мы хотели бы найти ответ, звучит так: был ли коэффициент конверсии в покупателя у канадцев выше, чем у других пользователей. Вместо того чтобы сравнивать со всей базой пользователей, мы будем сравнивать только с пользователями в США. Эти страны близки географически, большинство пользователей в этих двух странах говорят на одном языке, и предположим, что для этого примера мы провели другой анализ, показавший, что их поведение схоже, а поведение пользователей в других странах отличается настолько, что их можно исключить.

Для выполнения анализа мы формируем варианты на основе любой отличительной характеристики — в данном случае это будет поле `country` из таблицы `game_users`, но иногда вам потребуется написать более сложный SQL, в зависимости от набора данных. Количества пользователей в группах и тех, кто сделал покупку в игре, рассчитываются теми же способами, что и ранее:

```
SELECT a.country
,count(distinct a.user_id) as total_cohorted
,count(distinct b.user_id) as purchasers
,count(distinct b.user_id) * 100.0 / 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        24.83
United States  45012           4958        11.01
```

Доля пользователей в Канаде, совершивших покупку, значительно выше — 24.83%, по сравнению с 11.01% в США. Подстановка этих значений в онлайн-калькулятор подтверждает, что коэффициент конверсии для Канады статистически более значим при доверительном интервале 95%.

Самое сложное в анализе естественного эксперимента — найти сопоставимую группу и доказать, что эти две группы достаточно похожи, чтобы подтвердить выводы статистического теста. Хотя практически невозможно доказать отсутствие других мешающих факторов, тщательное сравнение демографических характеристик и поведения групп придает результатам достоверности. Поскольку естественный эксперимент не является настоящим случайным экспериментом, доказательства причинно-следственной связи будут слабее, и это нужно отметить при публиковании результатов такого анализа.

## Анализ популяции около порогового значения

В некоторых случаях существует пороговое значение, которое приводит к тому, что какие-то сущности получают некое воздействие, а какие-то нет. Например, определенный средний балл может дать студентам право на получение стипендии или определенный уровень дохода может дать семьям право на субсидированное медицинское обслуживание, или высокий показатель оттока может побудить торгового представителя связаться с клиентами. В таких случаях мы можем использовать идею о том, что сущности по обе стороны от порогового значения, скорее всего, очень похожи друг на друга. Поэтому вместо того, чтобы сравнивать все группы, получившие и не получившие воздействие, мы можем сравнить только тех, кто был близок к пороговому значению как с положительной, так и с отрицательной стороны. Формальное название этого метода — *разрывный регрессионный дизайн* (regression discontinuity design, RDD).

Чтобы провести такой анализ, мы можем сформировать варианты, разбив данные около порогового значения, аналогично тому, как мы делали в анализе «до и после». К сожалению, не существует жесткого правила относительно того, насколько широкими должны быть диапазоны значений по обе стороны от порогового значения. Группы должны быть схожими по размеру и достаточно большими, чтобы обеспечить статистическую значимость результатам анализа. Один из способов — провести анализ несколько раз с разными диапазонами. Например, можно проанализировать различия между вариантной и контрольной группами, когда в каждой группе есть сущности, попадающие в пределы 5%, 7.5% и 10% от порогового значения. Если выводы, полученные в этих анализах, совпадают, значит, достоверность этих выводов более чем убедительна. Если же они не совпадают, то результаты можно считать неубедительными.

Как и в случае с другими видами неэкспериментальных анализов, результаты RDD следует рассматривать как менее убедительное доказательство причинно-следственной связи. Также следует уделить пристальное внимание потенциально влияющим

факторам. Например, если с клиентами с высоким риском оттока связываются сразу по нескольким каналам или им предоставляется специальная удерживающая скидка в дополнение к звонку торгового представителя, данные могут быть потенциально искажены этими дополнительными воздействиями.

## 7.6. Заключение

Анализ экспериментов — это обширная область, которая часто включает в себя различные виды анализов, рассмотренные в предыдущих главах этой книги, от когортного анализа до выявления аномалий. Профилирование данных может пригодиться для отслеживания возникающих проблем. Когда случайные эксперименты невозможны, доступны другие методы, а SQL можно использовать для создания условных контрольных и вариантных групп. В следующей главе мы перейдем к построению сложных наборов данных для анализа — теме, объединяющей все, что мы рассматривали до сих пор.

ГЛАВА 8

# Создание сложных наборов данных

В *гл. 3—7* мы рассмотрели множество способов, в которых используется SQL для анализа данных в базах. В дополнение к этим случаям иногда цель SQL-запроса состоит в том, чтобы сформировать новый набор данных, который будет и специфичным, и в то же время достаточно универсальным, чтобы в дальнейшем его можно было использовать для выполнения различных анализов. Результирующий набор данных может быть сохранен в таблицу базы данных, в текстовый файл или передан в инструмент бизнес-аналитики. При этом сам SQL-код может быть простым, с применением лишь нескольких фильтров или агрегирований. Однако чаще код и логика, необходимые для получения требуемого набора данных, очень сложны. Кроме того, такой код, вероятно, будет обновляться со временем, поскольку заинтересованные лица будут требовать дополнительные данные или вычисления. Тогда структура кода SQL, его производительность и удобство обслуживания становятся критически важными, чего нельзя сказать о единоразовых анализах.

В этой главе я расскажу о принципах упорядочивания кода, чтобы его было легко передавать и обновлять. Затем поговорим о том, когда логику следует реализовывать в SQL-запросе, а когда следует задуматься о переходе к постоянным таблицам с использованием процесса ETL. Далее я расскажу о способах хранения промежуточных результатов — о подзапросах, временных таблицах и общих табличных выражениях, и о том, как использовать их в коде. В заключение я рассмотрю способы уменьшения размера набора данных, и мы поговорим об обеспечении конфиденциальности данных и скрытии персональной информации.

## 8.1. SQL для сложных наборов данных

Почти все наборы данных, подготовленные для дальнейшего анализа, уже содержат определенную логику. Логика может варьироваться от относительно простой, например обычное соединение таблиц или фильтры в разделе `WHERE`, до сложных вычислений, которые выполняют агрегирование, категоризацию, разбор текста или запускают оконные функции над секциями данных. При создании наборов данных для дальнейшего анализа выбор того, где именно реализовывать логику — в SQL-запросе или на более раннем этапе в ETL, или позднее в другом инструменте, это зачастую не столько наука, сколько искусство. Удобство использования, производительность и доступная помощь разработчиков — все эти факторы влияют на ре-

шение. Как правило, не существует единственно правильного варианта, но чем дольше вы работаете с SQL-запросами, тем больше у вас опыта и уверенности.

## Преимущества использования SQL

SQL — очень гибкий язык. Надеюсь, в предыдущих главах я убедила вас, что с помощью SQL можно решать самые разнообразные задачи подготовки и анализа данных. Эта гибкость и является основным преимуществом использования SQL при разработке сложных наборов данных.

На начальных этапах работы с набором данных вы можете выполнить множество разных запросов. Работа часто начинается с нескольких запросов в рамках профилирования, чтобы изучить данные. Затем следует пошаговое построение сложного запроса, с промежуточными проверками преобразований и агрегирований, чтобы быть уверенными в правильности возвращаемых результатов. Это может перемежаться с повторным профилированием, когда получаемые значения начинают отличаться от наших ожиданий. Выполнение запроса и проверка результатов происходят быстро и позволяют проводить быстрые итерации. Сложные наборы данных могут быть построены путем объединения нескольких подзапросов, отвечающих на конкретные вопросы, с помощью `JOIN` или `UNION`.

Помимо того, что SQL полагается на качество и актуальность данных в таблицах, у него мало других зависимостей. Запросы выполняются по требованию и не зависят от разработчиков или релиза. Они могут быть встроены в инструменты бизнес-аналитики (BI) или в код на языке R или Python аналитиком или специалистом по данным самостоятельно, без обращения в техническую поддержку. Если заинтересованным лицам потребуется добавить в результирующий набор другой атрибут или агрегирование, изменения могут быть внесены достаточно быстро.

При работе над новым анализом, когда ожидается, что логика и набор результатов могут часто меняться, идеальным решением будет сохранение логики в самом коде SQL. Кроме того, когда запрос выполняется быстро и данные быстро возвращаются заинтересованным сторонам, необходимости переносить логику куда-либо еще может никогда и не возникнуть.

## Перенос логики в ETL

В некоторых случаях лучшим решением будет перенести часть логики в процесс ETL, чем реализовывать всю логику в SQL-запросе, особенно если в вашей организации уже есть хранилище данных (data warehouse) или озеро данных (data lake). Две основные причины для использования ETL — это его производительность и прозрачность.

Производительность SQL-запросов зависит от сложности логики, размера запрашиваемых таблиц и вычислительных ресурсов базы данных. Хотя многие запросы выполняются быстро, особенно на новых базах данных и аппаратных средствах, вы рано или поздно столкнетесь с написанием запросов, которые содержат сложные

вычисления или соединения JOIN с большими таблицами, или любую другую конструкцию, приводящую к увеличению времени выполнения до нескольких минут и более. Аналитик или специалист по данным может согласиться ждать ответа на такой запрос. Но большинство пользователей данных привыкли к быстрому времени отклика веб-сайтов и будут расстроены, если им придется ждать результатов дольше нескольких секунд.

ETL выполняется «за кулисами» в запланированное время и записывает результаты в таблицы. Поэтому он может работать 30 секунд, 15 минут или час, и конечных пользователей это не коснется. Расписание часто бывает ежедневным, но может быть установлен и более короткий интервал. Пользователи могут запрашивать таблицу с результатами напрямую, без использования JOIN или другой сложной логики, и таким образом быстро получать ответ на свой запрос.

Хорошим примером того, когда лучше выбрать ETL, чем хранить всю логику в запросе SQL, является таблица ежедневных снимков (снэпшотов). Во многих организациях хранение ежедневных снимков клиентов, заказов или других сущностей используется для аналитических целей. Мы можем подсчитать для клиентов общее количество заказов или посещений на сегодняшний день, текущий статус продаж и другие атрибуты, которые меняются или накапливаются. Мы уже видели, как создавать ежедневные временные ряды, в том числе для дней, когда сущность отсутствовала, при обсуждении анализа временных рядов в *гл. 3* и когортного анализа в *гл. 4*. На уровне отдельных сущностей и на длительном временном периоде такие запросы могут быть очень медленными. Кроме того, такие атрибуты, как текущий статус, могут быть изменены в исходной таблице, поэтому создание ежедневных снимков может быть единственным способом сохранения истории операций. Разработка ETL и хранение ежедневных снимков в таблице, как правило, стоят затраченных усилий.

Прозрачность — вторая причина переноса логики в ETL. Часто SQL-запросы хранятся на локальном компьютере каждого сотрудника или встроены в отчеты. Другим коллегам иногда трудно понять логику, заложенную в запросе, не говоря уже о том, чтобы разобраться в ней и проверить на наличие ошибок. Перенос логики в ETL и хранение кода ETL в репозитории, например GitHub, упрощает сотрудникам организации поиск, проверку и обновление кода. Большинство репозиториев, используемых командами разработчиков, также хранят историю изменений — дополнительное преимущество, позволяющее видеть, когда была добавлена или изменена конкретная строка запроса.

Несмотря на то, что есть много веских причин для переноса логики в ETL, у этого подхода существуют свои недостатки. Один из них заключается в том, что свежие результаты будут недоступны до тех пор, пока задание ETL не будет выполнено и не обновит данные, даже если в исходную таблицу новые данные уже поступили. Это можно обойти, если для новых записей продолжать использовать SQL-запрос на необработанных данных, но ограничив его выполнение на небольшом временном окне, чтобы результаты возвращались быстро. При желании этот запрос можно объединить с запросом к таблице ETL, используя подзапросы или UNION. Еще одним

недостатком размещения логики в ETL является то, что ее сложнее изменить. Обновления или исправления ошибок в ETL часто приходится передавать инженеру по данным, а код тестировать, сохранять в репозитории и выпускать с очередным релизом хранилища данных. По этой причине я обычно предпочитаю подождать, пока мои SQL-запросы не пройдут период быстрых итераций и пока полученные наборы данных не будут проверены и не начнут использоваться организацией, прежде чем переносить их в ETL. Конечно, усложнение процесса изменения кода и дополнительные проверки кода — отличный способ обеспечить согласованность и качество данных.

**Представления как альтернатива ETL**

Если вам нужны возможности ETL — многократное использование кода и его прозрачность, но при этом не требуется хранить промежуточные результаты и, следовательно, занимать лишнюю память, хорошим вариантом может стать использование представлений (view) базы данных. *Представление* — это, по сути, сохраненный запрос с псевдонимом, на который можно ссылаться так же, как и на любую другую таблицу базы данных. Сам запрос может быть простым или сложным и включать в себя соединения таблиц, фильтры и другие конструкции SQL.

Представления можно использовать для того, чтобы быть уверенными, что все, кто запрашивает данные, используют одни и те же записи из базового запроса, например всегда исключают тестовые транзакции. Представления могут скрывать от пользователей сложности основной логики, что полезно для начинающих или временных пользователей базы данных. Представления также можно использовать для обеспечения дополнительного уровня безопасности, ограничивая доступ к определенным строкам или столбцам таблиц. Например, представление может быть создано для скрытия персональных данных, таких как адреса электронной почты, но при этом позволять пользователям базы данных просматривать другие допустимые атрибуты клиентов.

Однако у представлений есть несколько недостатков. Они являются объектами базы данных и поэтому требуют прав доступа для создания и изменения. Представления не хранят данные, поэтому каждый раз, когда выполняется запрос с использованием представления, база данных должна обратиться к исходной таблице или таблицам, чтобы получить данные. Поэтому они не заменяют ETL, который создает таблицу с предварительно вычисленными данными.

Большинство основных баз данных имеют также *материализованные представления* (materialized view), которые похожи на обычные представления, но хранят возвращаемые данные в таблице. Перед тем как планировать создание и обновление материализованных представлений, лучше всего проконсультироваться с опытным администратором базы данных, поскольку при их использовании есть нюансы, касающиеся производительности, но выходящие за рамки этой книги.

## Перенос логики в другие инструменты

Код SQL и результаты запроса, получаемые в редакторе запросов, часто являются лишь частью анализа. После этого результаты вставляются в отчеты, выводятся в виде таблиц и графиков или подвергаются дальнейшей обработке с помощью различных инструментов — от электронных таблиц и BI-инструментов до систем, в которых применяется статистика или машинное обучение. Помимо принятия решения о перемещении логики на шаг раньше в ETL, мы также должны определиться, перемещать ли логику на шаг позже в другие инструменты. Ключевыми факторами при принятии решения здесь являются производительность и специфика каждого конкретного случая.

Любой инструмент имеет свои сильные и слабые стороны. Электронные таблицы очень гибкие, но не способны обрабатывать большое количество строк или выполнять сложные вычисления по многим строкам. Базы данных определенно имеют над ними преимущество в производительности, поэтому лучше выполнять как можно больше вычислений в базе данных и передавать в электронную таблицу меньший набор данных.

Инструменты бизнес-аналитики обладают различными возможностями, поэтому важно понимать, как программа справляется с вычислениями и как будут использоваться данные. Некоторые BI-инструменты могут кэшировать данные (сохранять локальную копию) в оптимизированном виде, что ускоряет вычисления. Другие инструменты посылают новый запрос каждый раз, когда поле добавляется или удаляется из отчета, и поэтому лучше использовать вычислительную мощность базы данных. Определенные вычисления, такие как `count distinct` и `median`, требуют детализированных данных на уровне сущностей. Если невозможно заранее предусмотреть все вариации необходимых вычислений, может потребоваться передача большего и более подробного набора данных, чем предполагалось сначала. Кроме того, если целью является создание набора данных, позволяющего исследовать данные и делать различные срезы, лучше иметь более детализированную информацию. Поиск оптимального сочетания SQL, ETL и BI-инструментов может потребовать нескольких итераций.

Если целью является проведение статистического анализа или машинного обучения на наборе данных с использованием таких языков, как R или Python, лучше работать с подробными данными. Оба эти языка могут выполнять те же задачи, что и SQL, например агрегирование и разбор текста. Лучше выполнять как можно больше расчетов на SQL, чтобы использовать вычислительные мощности базы данных. Но сложное итерирование тоже может являться важной частью процесса. Выбор того, выполнять ли вычисления на SQL или на другом языке, может также зависеть от вашего уровня владения каждым из них. Те, кто хорошо знает SQL, могут предпочесть выполнять больше вычислений в базе данных, в то время как те, кто лучше владеет R или Python, могут выполнять больше вычислений на этих языках.

Хотя существует несколько правил для принятия решения о том, куда поместить логику, я настоятельно рекомендую вам следовать одному из них: избегайте ручных действий. Очень легко открыть набор данных в виде электронной таблицы или в текстовом редакторе, внести небольшие изменения, сохранить и двигаться дальше. Но когда вам нужно провести итерацию или когда поступают новые данные, легко забыть об этом ручном шаге или выполнить его не в той последовательности. По моему опыту, не существует такого понятия, как действительно «одноразовый» запрос. Если возможно, поместите всю логику в код.

SQL — удобный и невероятно гибкий инструмент. Он также хорошо вписывается в рабочий процесс анализа и в ваш набор инструментов. Решение о том, где разместить вычисления, может потребовать выполнения нескольких проб и ошибок, пока вы перебираете возможные комбинации SQL, ETL и других инструментов. Чем больше у вас будет опыта работы со всеми доступными комбинациями, тем быстрее вы сможете оценивать варианты, а также эффективнее повышать производительность и гибкость своей работы.

## 8.2. Упорядочивание кода

В SQL есть несколько правил форматирования, которые нужно применять к запросам. Разделы в запросе должны располагаться в правильном порядке: после `SELECT` следует `FROM`, и, например, `GROUP BY` не может располагаться перед `WHERE`. Некоторые ключевые слова, такие как `SELECT` и `FROM`, зарезервированы, т. е. их нельзя использовать в качестве имен полей, таблиц или псевдонимов. Однако, в отличие от некоторых языков, переносы на новые строки, пробельные символы (кроме пробелов, которые разделяют слова) и регистр букв не важны и игнорируются базой данных. Любой из запросов в этой книге можно написать в одну строчку, одними заглавными или строчными буквами, за исключением строк, заключенных в кавычки. В результате форматирование кода целиком зависит от того, кто пишет запрос. К счастью, у нас есть несколько официальных и неофициальных инструментов для упорядочивания кода, от комментариев и «косметического» форматирования до параметров хранения файлов с SQL-кодом.

### Комментарии

В большинстве языков программирования есть способ указать, что блок текста должен рассматриваться как комментарий к коду и игнорироваться во время выполнения. В SQL для этого есть два способа. Первый — использовать два символа дефиса, что превращает весь последующий текст в данной строке в комментарий:

```
-- Это комментарий
```

Второй способ — использовать символы косой черты и звездочки (/*) для обозначения начала блока комментария, который может занимать несколько строк, а затем звездочку и косую черту (*/) для обозначения конца блока комментария:

```
/*
Это блок комментария
с несколькими строками
*/
```

Многие редакторы SQL изменяют стиль отображения текста комментариев, выделяя его серым или любым другим цветом, чтобы их было легче заметить.

Комментирование кода — это хорошая практика, но, по общему наблюдению, редко кто делает это на регулярной основе. Как правило, SQL пишется очень быстро, особенно во время исследования или профилирования данных, и мы не ожидаем, что этот код останется с нами надолго. Код с излишними комментариями может быть таким же трудным для чтения, как и код без комментариев. И все мы слишком самоуверены, думая, что раз мы сами написали код, то всегда сможем вспомнить, что и зачем он делает. Однако каждый, кто унаследовал длинный запрос, написанный коллегой, или не возвращался к своему запросу несколько месяцев, а затем понадобилось его обновить, знает, что разбор кода может быть утомительным и отнимать много времени.

Чтобы сбалансировать плюсы и минусы комментирования кода, я стараюсь следовать нескольким правилам. Во-первых, добавляйте комментарий везде, где используемое значение имеет неочевидный смысл. Многие исходные системы записывают разные атрибуты в виде целых чисел, и их значения легко забыть. Если оставить комментарий, то смысл сразу будет понятен, и код будет легко изменить при необходимости:

```
WHERE status in (1,2)    -- 1 - активный, 2 - ожидающий
```

Во-вторых, добавляйте комментарии о любых неочевидных вычислениях или преобразованиях. Это может быть что угодно, о чем может не знать человек, не потративший много время на профилирование набора данных, от ошибок ввода данных до возможных выбросов:

```
case when status = 'Live' then 'Active'
     else status end
     /* раньше для клиентов мы использовали статус Live,
     но в 2020 г. перешли на статус Active */
```

Третье правило, которому я стараюсь следовать при комментировании кода, — это оставлять пояснения, когда запрос содержит несколько подзапросов. Короткий комментарий о том, что вычисляет каждый подзапрос, позволяет легко найти соответствующий фрагмент в этом длинном запросе, когда вы возвращаетесь к этому коду позднее для проверки качества или редактирования:

```
SELECT...
FROM
```

```
( -- найти дату регистрации каждого клиента
    SELECT ...
    FROM ...
) a
JOIN
( -- найти все товары для каждого клиента
    SELECT ...
    FROM ...
) b on a.field = b.field
...
;
```

Хорошее комментирование требует практики и определенной дисциплины, но это стоит делать для большинства запросов, длина которых превышает несколько строк кода. Комментарии также можно использовать для добавления общей информации о запросе, такой как назначение, автор, дата создания и т. д. Будьте добры к своим коллегам и к будущему себе и оставляйте полезные комментарии в своем коде.

## Регистр, отступы, круглые скобки и другие приемы форматирования

Форматирование, и особенно последовательное форматирование, является хорошим способом сохранить код SQL читаемым и поддерживаемым. Базы данных игнорируют регистр и пробельные символы (пробелы, табуляции и новые строки) в SQL, поэтому мы можем использовать их в своих интересах для выделения кода в более читаемые блоки. Круглые скобки могут как контролировать порядок выполнения, о чем мы поговорим позднее, так и визуально группировать элементы вычислений.

Слова, написанные заглавными буквами, выделяются на фоне остальных, что может подтвердить каждый, кто когда-нибудь получал электронное письмо с темой, набранной заглавными буквами. Я предпочитаю использовать заглавные буквы только для основных разделов запроса: `SELECT`, `FROM`, `JOIN`, `WHERE` и т. д. Это особенно удобно в длинных или сложных запросах, когда можно быстро заметить, где заканчивается раздел `SELECT` и начинается раздел `FROM`, что экономит много времени.

Пробельные символы — это еще один важный способ организовать и визуально упростить поиск фрагментов запроса, а также понять, какие фрагменты логически связаны друг с другом. Любой SQL-запрос можно записать в одну строку в редакторе кода, но в большинстве случаев это приведет к необходимости прокручивать длинную строку влево и вправо. Мне нравится начинать каждый раздел (`SELECT`, `FROM` и т. д.) с новой строки, что, наряду с использованием заглавных букв, помогает мне отслеживать, где он начинается и заканчивается. Кроме того, я считаю, что размещение каждого агрегирования, как и каждой длинной функции, в отдельной строке помогает в упорядочивании кода. Разделение оператора `CASE` с более чем двумя ус-

ловиями WHEN на несколько строк также является хорошим способом легко видеть и отслеживать, что происходит в коде. В качестве примера мы можем вернуть тип type и магнитуду mag землетрясений, разобрать текстовое поле place и подсчитать количество записей в таблице earthquakes с определенной фильтрацией в разделе 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
...                 ...  ...         ...
```

Отступы — еще один способ для визуального упорядочивания кода. Одним из таких примеров является добавление пробелов или табуляции для выравнивания условий WHEN в операторе CASE. Вы также видели подзапросы с отступами во многих примерах этой книги. Это визуально выделяет подзапрос, а если в запросе есть несколько уровней вложенных подзапросов, то это упрощает понимание порядка, в котором они будут выполняться, и того, какие из них равноправны с точки зрения уровня вложенности:

```
SELECT...
FROM
(
    SELECT...
    FROM
    (
        SELECT...
        FROM...
    ) a
    JOIN
```

```
    (
        SELECT...
        FROM
    ) b on...
) a
...
;
```

Можно сделать любое другое форматирование, и запрос вернет те же результаты. Кто пишет на SQL в течение длительного времени, как правило, имеет свои собственные предпочтения в форматировании. Однако четкое и последовательное форматирование всегда значительно упрощает создание, сопровождение и совместное использование SQL-кода.

Многие редакторы SQL-запросов предоставляют свое форматирование и выделение цветом. Обычно ключевые слова подсвечиваются другим цветом, что визуально выделяет их в тексте запроса. Такие визуальные подсказки значительно упрощают и разработку, и проверку SQL-запросов. Если вы все время писали SQL в редакторе запросов, попробуйте открыть файл **sql** в обычном текстовом редакторе, чтобы увидеть разницу. На рис. 8.1 показан пример кода в редакторе SQL-запросов, а на рис. 8.2 показано, как тот же код выглядит в обычном текстовом редакторе.

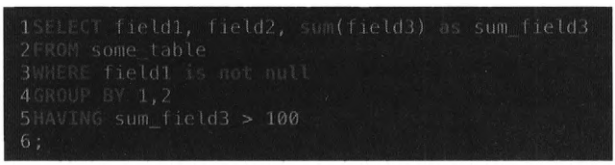


**Рис. 8.1.** Пример подсвечивания ключевых слов в редакторе SQL-запросов DBVisualizer

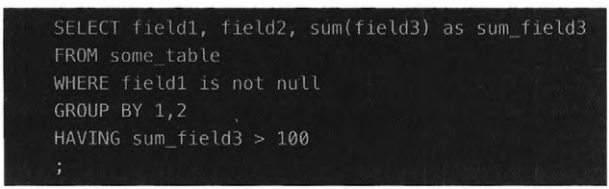


**Рис. 8.2.** Пример кода в обычном текстовом редакторе Atom

Форматирование кода не является обязательным для баз данных, но это очень хорошая практика. Последовательное использование отступов, заглавных букв и других возможностей форматирования в значительной степени способствует сохранению читаемости вашего кода, что облегчает его совместное использование и сопровождение.

### Хранение кода

После того как вы добавили комментарии и отформатировали SQL-код, неплохо было бы сохранить его где-нибудь на случай, если вам понадобится использовать или сослаться на него позднее.

Многие аналитики и специалисты по данным часто работают с десктопной версией SQL-редакторов. Редакторы SQL удобны тем, что они обычно позволяют просматривать схемы базы данных наряду с окном редактирования кода. Файлы сохраняются с расширением **sql**, и эти текстовые файлы можно открывать и изменять в любом текстовом редакторе. Файлы можно сохранить локально на вашем компьютере или в облачных сервисах для хранения файлов.

Поскольку файлы SQL являются текстовыми, их удобно хранить в репозиториях систем контроля версий, таких как GitHub. Использование такого репозитория обеспечивает возможность резервного копирования и облегчает обмен файлами с другими пользователями. Репозитории также отслеживают историю изменений файлов, что очень полезно, если нужно выяснить, когда было внесено то или иное изменение, или если нужно хранить историю изменений по нормативным требованиям. Главный недостаток GitHub и аналогичных инструментов заключается в том, что они обычно не являются обязательными в процессе анализа. Вы должны сами не забывать периодически обновлять свой код в репозитории, и, как в случае с любым другим ручным действием, этот шаг легко пропустить.

## 8.3. Контроль над порядком вычислений

При создании сложных наборов данных мы сталкиваемся с двумя связанными проблемами: правильная логика и хорошая производительность запросов. Логика должна быть правильной, иначе результаты будут бесполезными. Производительности запросов для целей анализа, в отличие от транзакционных систем, обычно хватает и она «достаточно хороша». Запросы, которые не возвращают результаты в течение долгого времени, являются проблемными, но разница между 30 секундами и минутой ожидания не имеет большого значения. В SQL часто существует более одного способа написать запрос, который возвращает нужные результаты. И мы, конечно, можем это использовать как для обеспечения правильной логики, так и для настройки производительности длительных запросов. Существует три основных способа управления порядком вычислений промежуточных результатов в SQL: подзапросы, временные таблицы и общие табличные выражения (common table expressions, CTE). Прежде чем мы поговорим о них, давайте рассмотрим порядок выполнения операций в SQL. В конце раздела я представлю расширения для группировки полей, которые в некоторых случаях могут заменить многочисленные объединения запросов через `UNION`.

## Порядок выполнения операций SQL

База данных преобразует SQL-запрос в набор операций, которые будут выполнены для возврата требуемых данных. Хотя вам не обязательно разбираться, как именно это работает, чтобы писать хороший SQL для анализа, но знать порядок, в котором база данных будет выполнять эти операции, очень полезно (а иногда и необходимо для отладки неожиданных результатов).

Многие современные базы данных имеют встроенные умные оптимизаторы запросов, которые анализируют различные части запроса, чтобы выработать наиболее эффективный план его выполнения. Хотя они могут анализировать части запроса в своем порядке, отличном от рассматриваемого далее, и меньше нуждают в оптимизации запроса со стороны человека, но промежуточные результаты будут вычисляться именно в том порядке, который приведен здесь.

Общий порядок выполнения показан в табл. 8.1. SQL-запрос обычно включает в себя только часть этих возможных операций, поэтому его фактическое выполнение будет содержать только те шаги, которые имеют к нему отношение.

**Таблица 8.1.** Порядок выполнения операций SQL-запроса

| Номер шага | Операция |
|---|---|
| 1 | `FROM`, включая `JOIN` и условия `ON` |
| 2 | `WHERE` |
| 3 | `GROUP BY`, включая агрегирование |
| 4 | `HAVING` |
| 5 | Оконные функции |
| 6 | `SELECT` |
| 7 | `DISTINCT` |
| 8 | `UNION` |
| 9 | `ORDER BY` |
| 10 | `LIMIT` и `OFFSET` |

Сначала определяются таблицы в разделе `FROM`, а также во всех `JOIN`. Если в разделе `FROM` есть подзапросы, они выполняются до того, как процесс перейдет к следующему шагу. В `JOIN` с условием `ON` указывается способ соединения таблиц, который также может фильтровать набор результатов.

Раздел FROM всегда выполняется первым, за одним исключением: когда запрос его не содержит. В большинстве баз данных можно выполнять запросы, используя только SELECT, как было показано в некоторых примерах этой книги. Запрос только с разделом SELECT может возвращать системную информацию, например текущую дату или версию базы данных. Он также может выполнять функции даты/времени, математические, текстовые и другие функции для константных значений. Несмотря на то, что в окончательном анализе такие запросы вряд ли будут использоваться, они очень удобны для тестирования функций или быстрого выполнения итераций со сложными вычислениями.

Далее выполняется раздел WHERE, чтобы определить, какие записи должны быть включены в последующие вычисления. Обратите внимание, что WHERE является одним из первых шагов и поэтому не может включать в себя результаты вычислений, которые определяются на более поздних шагах.

Затем выполняются раздел GROUP BY и соответствующие агрегирования, такие как count, sum и avg. Как и следовало ожидать, GROUP BY будет применяться только к тем записям, которые остались в таблицах FROM после всех JOIN и фильтрации WHERE.

Следующим выполняется раздел HAVING. Поскольку он следует за GROUP BY, то может выполнять фильтрацию агрегированных значений, полученных на предыдущем шаге. Единственный альтернативный способ фильтрации по агрегированным значениям — поместить агрегирование в подзапрос и применить фильтр во внешнем запросе. Например, мы хотим найти все штаты в наборе данных о законодателях из *гл. 4*, в которых было не менее тысячи сроков законодателей, и упорядочим количество сроков по убыванию:

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

На следующем шаге выполняются оконные функции, если они есть в запросе. Заметьте, что поскольку агрегирования уже были рассчитаны на предыдущих шагах, их можно использовать в определении оконных функций. Например, мы можем

рассчитать как количество сроков законодателей для каждого штата, так и среднее количество сроков по всем штатам в одном запросе:

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

Агрегирование также можно использовать в предложении OVER, как показано в следующем запросе, который ранжирует штаты в порядке убывания количества сроков:

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

И, наконец, в последнюю очередь выполняется раздел SELECT. Это немного противоречит здравому смыслу, поскольку агрегирования и оконные функции указаны непосредственно в разделе SELECT. Так как база данных уже позаботилась о вычислениях этих функций, то их результаты доступны для дальнейшей обработки или просто для вывода. Например, агрегирование может находиться в операторе CASE или к нему могут применяться функции даты/времени, математические или текстовые функции, если результаты агрегирования имеют соответствующий тип данных.

Функции агрегирования `sum`, `count` и `avg` возвращают числовые значения. Но функции `min` и `max` возвращают тот же тип данных, что и поле, к которому они применяются, и используют присущую этому типу данных упорядоченность. Например, `min` и `max` для поля даты возвращают самую раннюю и самую позднюю календарные даты, а `min` и `max` для текстового поля используют алфавитный порядок для определения результата.

После `SELECT` выполняется предложение `DISTINCT`, если оно указано в запросе. Это означает, что сначала находятся все строки, а затем происходит дедупликация.

Далее выполняется `UNION` или `UNION ALL`. До этого момента каждый запрос, входящий в `UNION`, выполняется независимо. На данном шаге результаты запросов собираются вместе. Это означает, что запросы могут выполнять свои вычисления совершенно по-разному или из разных наборов данных. Все, что требует `UNION`, — одинаковое количество столбцов и совместимые типы данных в этих столбцах.

Раздел `ORDER BY` — предпоследний шаг при выполнении запроса. Это означает, что ему доступны результаты всех предыдущих вычислений для их сортировки. Единственное ограничение заключается в том, что если в запросе используется `DISTINCT`, раздел `ORDER BY` не может содержать какие-либо поля, которые не возвращаются в `SELECT`. В обычном запросе (без `DISTINCT`) вполне возможно упорядочивание результатов по полю, которое не указано в запросе.

Предложения `LIMIT` и `OFFSET` выполняются на последнем шаге при выполнении запроса. Это гарантирует, что ограничение возвращаемых результатов будет применяться к уже полностью вычисленному набору данных. Это также означает, что `LIMIT` имеет несколько ограниченное влияние на объем работы, проделываемой базой данных для возвращения результатов. Это, наверное, сильнее всего заметно, когда запрос содержит большое значение смещения `OFFSET`. Для того чтобы сделать `OFFSET`, скажем, на три миллиона записей, базе данных все равно придется вычислить весь набор результатов, определить, где находится три миллиона плюс одна запись, а затем вернуть количество записей, указанное в `LIMIT`. Это не означает, что `LIMIT` бесполезен. Ограничение возвращаемых результатов помогает проверить вычисления, не перегружая сеть или ваш локальный компьютер лишними данными. Кроме того, использование `LIMIT` как можно ранее, например в подзапросе, может значительно сократить объем работы базы данных при создании сложного запроса.

Теперь, когда мы хорошо понимаем порядок, в котором базы данных выполняют запросы и вычисления, перейдем к некоторым способам управления этими операциями в контексте более крупного, сложного запроса: подзапросы, временные таблицы и CTE.

## Подзапросы

Подзапросы обычно являются первым способом, с помощью которого мы учимся управлять порядком вычислений в SQL, т. е. выполнять вычисления, которые не могут быть реализованы с помощью одного обычного запроса. Подзапросы универ-

сальны и помогают разбить длинные запросы на более мелкие логические блоки со своими отдельными целями.

Подзапрос должен быть заключен в круглые скобки — как и в математике, где круглые скобки также выделяют некоторую часть уравнения, которая должна быть выполнена в первую очередь. Внутри круглых скобок находится отдельный запрос, который выполняется до внешнего запроса. Если подзапрос находится в разделе FROM, то внешний запрос может обращаться к результатам подзапроса, как к любой другой таблице. В этой книге мы уже рассмотрели множество примеров с подзапросами.

Исключением из таких автономных подзапросов является специальный тип, называемый *латеральным подзапросом*, который может обращаться к результатам предыдущих элементов раздела FROM. Вместо JOIN используются запятая и ключевое слово LATERAL, и не нужно указывать никакого условия ON. И тогда внутри латерального подзапроса можно ссылаться на столбцы предыдущего подзапроса. В качестве примера представьте, что мы хотим проанализировать предыдущее членство в партии для ныне работающих законодателей. Мы можем найти первый год, когда они были членом другой партии, и проверить, насколько часто это встречается, если сгруппировать их по текущей партии. В первом подзапросе a мы находим всех нынешних законодателей. Во втором латеральном подзапросе c мы используем результаты первого подзапроса, чтобы вернуть начало term_start самого первого срока, когда партия отличается от текущей партии:

```
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
```

Как оказалось, это довольно редкое явление. Только трое нынешних законодателей сменили партию, и нельзя выделить никакую партию, в которой было бы больше всего «перебежчиков». Есть и другие способы получить такой же результат, например изменив латеральный подзапрос на `JOIN` и переместив его условие из `WHERE` в `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
;
```

Если вторая таблица очень большая, то добавление фильтрации по значению, возвращенному в первом подзапросе, может ускорить выполнение. По моему опыту, `LATERAL` редко используется, а значит, такой синтаксис будет менее понятен, чем любой другой, поэтому его стоит приберечь для задач, которые нельзя будет эффективно решить другими способами.

Подзапросы обеспечивают большую гибкость и контроль над порядком вычислений. Однако очень сложная последовательность вычислений в середине длинного запроса может стать трудной для чтения и поддержки. А в каких-то случаях производительность подзапросов может оказаться слишком низкой, или запрос вообще не будет возвращать результатов. К счастью, в SQL есть некоторые дополнительные возможности, которые могут помочь в таких ситуациях: временные таблицы и общие табличные выражения.

## Временные таблицы

*Временная таблица* создается так же, как и обычная таблица, но с одним отличием: она доступна только в течение текущей сессии подключения к базе данных. Временные таблицы удобно использовать, когда вам для работы нужен только небольшой фрагмент очень большой таблицы, т. к. на маленьких таблицах запросы выполняются гораздо быстрее. Они также могут пригодиться, когда вы хотите использовать какой-то промежуточный результат в нескольких запросах. Поскольку временная таблица является отдельной таблицей, к ней можно обращаться много раз в течение одной сессии. Еще один полезный случай — работа с некоторыми базами данных, такими как Redshift или Vertica, которые распределяют данные по узлам. Вставка данных во временную таблицу позволяет сложить вместе данные из других таблиц, чтобы потом с помощью `JOIN` использовать их в последующих запросах. У временных таблиц есть два основных недостатка. Во-первых, для работы с ними требуются привилегии на запись данных, что может быть запрещено по соображениям безопасности. Во-вторых, некоторые BI-инструменты, такие как Tableau и Metabase, позволяют создавать набор данных только с помощью одного оператора SQL[1], в то время как временная таблица требует как минимум два: оператор для создания и добавления данных во временную таблицу и сам запрос.

Для создания временной таблицы можно использовать оператор `CREATE`, за которым следует ключевое слово `TEMPORARY` и имя, которое вы хотите присвоить таблице. Затем для заполнения таблицы данными можно использовать оператор `INSERT`. Например, вы можете создать временную таблицу со списком штатов, для которых избирались законодатели:

```
CREATE temporary table temp_states
(
state varchar primary key
)
;

INSERT into temp_states
SELECT distinct state
FROM legislators_terms
;
```

Первый оператор создает временную таблицу, а второй оператор заполняет ее значениями из запроса. Обратите внимание, что при определении таблицы мы должны указать тип данных для всех ее столбцов (в данном случае столбец `state` имеет тип `varchar`), и при желании можно использовать другие элементы определения таблицы, например задать первичный ключ. Хорошей практикой считается добавление к именам временных таблиц приставки `"temp_"` или `"tmp_"` (от англ. *temporary* — вре-

[1] Хотя в случае с Tableau вы можете обойти это с помощью Initial SQL.

менный), чтобы было сразу понятно, что в запросе используется временная таблица, но это не является строго обязательным.

Более быстрый и простой способ создания временной таблицы — это оператор CREATE as SELECT:

```
CREATE temporary table temp_states
as
SELECT distinct state
FROM legislators_terms
;
```

В этом случае база данных автоматически определяет типы столбцов на основе данных, возвращаемых оператором SELECT, и первичный ключ не задается. Если вам не нужен контроль над созданием временных таблиц для управления производительностью, то этот второй способ отлично вам подойдет.

После того как вы заполнили временную таблицу данными, если вам понадобится пересобрать ее во время той же сессии, вам придется удалить эту таблицу с помощью DROP и заново создать ее или удалить из нее все данные с помощью TRUNCATE. Также можно использовать отключение и повторное подключение к базе данных.

## Общие табличные выражения

Общие табличные выражения (common table expressions, CTE) — это относительно новая конструкция языка SQL, которая появилась во многих крупных базах данных только в начале 2000-х гг. В течение многих лет я писала запросы SQL без них, используя только подзапросы и временные таблицы. Однако с тех пор, как я узнала о них несколько лет назад, они все больше и больше мне нравятся.

Можно считать, что *общее табличное выражение* — это подзапрос, который достали из основного запроса и поместили в самое начало выполнения. Оно возвращает временный набор результатов, который затем может быть использован как угодно в последующем запросе. Запрос может содержать несколько CTE, и CTE может использовать результаты предыдущих CTE для выполнения сложных вычислений.

CTE особенно удобен в случае, когда его результат используется несколько раз в остальной части запроса. Альтернативный способ — определить один и тот же подзапрос несколько раз — будет и медленнее (поскольку базе данных необходимо выполнить один и тот же запрос несколько раз) и чреват ошибками. При изменении логики можно забыть обновить какой-то из этих одинаковых подзапросов, что, конечно, приведет к ошибке в конечном результате. Поскольку CTE является частью запроса, для его использования не требуется специальных разрешений базы данных. Общие табличные выражения также могут быть хорошим способом упорядочивания кода в отдельные блоки и уменьшения вложенности подзапросов.

Основной недостаток CTE заключается в том, что он определяется в самом начале запроса, отдельно от того места, где он упоминается. Это может усложнить чтение

очень длинного запроса, поскольку необходимо будет прокрутить запрос в начало, чтобы проверить определение CTE, а затем снова вернуться к тому месту, где оно используется, чтобы понять, что происходит. Здесь может помочь хороший комментарий. Вторая проблема заключается в том, что CTE усложняет выполнение длинных запросов по частям. Чтобы проверить промежуточные результаты длинного запроса, можно просто выделить и выполнить любой подзапрос в редакторе SQL-запросов. Однако если в этом длинном запросе используется CTE, необходимо сначала закомментировать весь промежуточный код.

Чтобы создать CTE, мы применяем ключевое слово `WITH` в начале основного запроса, за которым следует название CTE, а затем в круглых скобках запрос, который он обозначает. Например, мы можем создать CTE, который рассчитывает начало первого срока для каждого законодателя, а затем использовать этот результат при дальнейших расчетах когорт, как мы делали в примерах *гл. 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 period
,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
;

period  cohort_retained
------  ---------------
0.0     12518
1.0     3600
2.0     3619
...     ...
```

Результат этого запроса совпадает с результатом альтернативного способа с использованием подзапроса, рассмотренного в *разд. 4.3*. В одном запросе можно указать несколько CTE через запятую:

```
WITH first_cte as
(
    SELECT...
),
```

```
second_cte as
(
    SELECT...
)
SELECT...
;
```

CTE — это удобный способ контролировать порядок вычислений, повышать производительность выполнения запросов и упорядочивать код SQL. Общие табличные выражения очень просто использовать, если вы знакомы с синтаксисом, и они доступны в большинстве основных баз данных. В SQL очень часто существуют несколько способов выполнить то или иное действие, а с помощью CTE ваши навыки SQL станут более гибкими.

## Расширения для группировки

Хотя эта тема не касается напрямую управления порядком выполнения, но это удобный способ избежать использования `UNION` и выполнить всю работу в одном операторе запроса. Во многих базах данных в разделе `GROUP BY` есть специальный синтаксис, включающий расширения `GROUPING SETS`, `CUBE` и `ROLLUP` (хотя в Redshift их совсем нет, а в MySQL есть только `ROLLUP`). Они полезны, когда набор данных должен содержать совокупные итоги для различных комбинаций атрибутов.

| * | 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 |

**Рис. 8.3.** Фрагмент таблицы `videogame_sales`

Для примеров в этом разделе мы будем использовать набор данных о продажах видеоигр, доступный на сайте Kaggle[2]. Набор данных в формате CSV и SQL для загрузки есть в репозитории GitHub[3] для этой книги. Название таблицы — `videogame_sales`. Таблица содержит такие атрибуты игр, как название, платформа, год, жанр и издатель. Также приведены данные о продажах для Северной Америки

[2] **https://www.kaggle.com/datasets/gregorut/videogamesales**

[3] **https://github.com/cathytanimura/sql_book/tree/master/Chapter 8: Creating Complex Data Sets**

(na_sales), Евросоюза (eu_sales), Японии (jp_sales), других стран (other_sales) и всего мира (global_sales). На рис. 8.3 показан пример данных из таблицы.

Предположим, нам нужно найти суммарные глобальные продажи global_sales по платформе, жанру и издателю — по каждому полю отдельное агрегирование (а не по комбинации этих трех полей) — и вывести результаты в одном запросе. Этого можно достичь с помощью объединения UNION трех запросов. Обратите внимание, что каждый запрос должен содержать заполнители null для неучаствующих в группировке полей:

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

Этого же результата можно достичь с помощью более компактного запроса, используя GROUPING SETS. В разделе GROUP BY за GROUPING SETS следует список группировок для вычисления. Предыдущий запрос можно заменить на:

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

В круглых скобках после GROUPING SETS указывается список столбцов, разделенных запятыми, а кроме того там могут быть пустые позиции. Например, кроме группировки по platform, genre и publisher, мы можем рассчитать общие глобальные продажи без какой-либо группировки, добавив в список просто пару круглых скобок. Давайте также очистим вывод, заменив все null-значения на 'All' с помощью функции coalesce:

```
SELECT coalesce(platform,'All') as platform
,coalesce(genre,'All') as genre
,coalesce(publisher,'All') as publisher
,sum(global_sales) as na_sales
FROM videogame_sales
GROUP BY grouping sets ((), platform, genre, publisher)
ORDER BY 1,2,3
;
```

```
platform  genre      publisher        global_sales
--------  ------     ---------        ------------
All       All        All              8920.44
2600      All        All              97.08
3DO       All        All              0.10
...       ...        ...              ...
```

```
All       Action     All              1751.18
All       Adventure  All              239.04
...       ...        ...              ...
All       All        10TACLE Studios  0.11
All       All        1C Company       0.10
...       ...        ...              ...
```

Если мы хотим вычислить не только отдельные итоги по каждому значению платформы, жанра и издателя, но и по всем комбинациям платформы и жанра, платформы и издателя, жанра и издателя, то мы можем указать эти комбинации в круглых скобках в `GROUPING SETS`. Или же мы можем использовать удобное расширение `CUBE`, которое сделает все это за нас:

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

Третье расширение `ROLLUP` возвращает промежуточные итоги по комбинациям, определяемым порядком следования полей в круглых скобках, а не все возможные комбинации этих полей. Таким образом, предыдущий запрос со следующей группировкой:

```
GROUP BY rollup (platform, genre, publisher)
```

возвращает агрегирование для комбинаций полей:

```
platform, genre, publisher
platform, genre
platform
```

При этом запрос не будет возвращать агрегирования для комбинаций:

```
platform, publisher
genre, publisher
genre
publisher
```

Хотя можно сформировать такой же результат с помощью UNION, использование GROUPING SETS, CUBE и ROLLUP позволяет значительно сократить объем кода и время, когда необходимо агрегирование на нескольких уровнях, поскольку эти расширения приводят к меньшему количеству строк кода и меньшей загрузке базы данных. Однажды я написала запрос длиной около сотни строк с использованием UNION, который был необходим для вывода динамической графики на веб-сайте и для которого нужно было предварительно рассчитать все возможные комбинации фильтров. Для проверки его качества был проделан огромный объем работы, а обновления проходили еще тяжелее. Использование GROUPING SETS и CTE могло бы значительно облегчить и написание кода, и его сопровождение.

## 8.4. Управление размером набора данных и проблемы конфиденциальности

После того как мы выстроили правильную логику нашего SQL-запроса, упорядочили код и проверили его эффективность, мы часто сталкиваемся с другой проблемой — объемом результирующих данных. Хранение данных становится все дешевле, что приводит к тому, что организации хранят все бóльшие массивы данных. Вычислительная мощность также постоянно растет, что позволяет нам обрабатывать все эти данные разными сложными способами, которые мы рассматривали в предыдущих главах. Однако узкие места все равно возникают — либо в системах последующей обработки результатов, таких как BI-инструменты, либо в пропускной способности каналов связи при передаче больших массивов данных между системами. Кроме того, серьезной проблемой является конфиденциальность данных, которая влияет на то, как мы работаем с чувствительными данными. Поэтому я расскажу о некоторых способах ограничения размера наборов данных, а также о сохранении конфиденциальности данных.

### Частичная выборка с помощью остатка от деления

Один из способов уменьшить размер возвращаемых результатов — сделать выборку исходных данных. *Частичная выборка* означает извлечение только небольшого подмножества точек данных или наблюдений. Это целесообразно, когда набор данных достаточно велик и подмножество является репрезентативным для всей совокупности. Например, вы можете получить частичную выборку трафика веб-сайта и сохранить при этом большую часть полезной информации. При выборке необходи-

мо определиться с двумя моментами. Первый — это размер выборки, при котором достигается правильный баланс между уменьшением размера набора данных и сохранением достаточного количества важной информации. Выборка может включать 10%, 1% или 0.1% всех точек данных, в зависимости от начального объема. Во-вторых, нужно определиться с сущностью, для которой будет проводиться выборка. Мы можем взять выборку 1% *посещений* сайта, но если цель анализа — понять, как пользователи перемещаются по сайту, то лучше сделать выборку 1% *посетителей* сайта, чтобы сохранить все полезные точки данных в выборке.

Самый распространенный способ выборки — отфильтровать результаты запроса в разделе `WHERE`, применив какую-нибудь функцию к идентификатору (ID) сущности. Многие поля ID представляют собой целые числа. Для таких идентификаторов самый быстрый способ добиться нужного результата — использовать остаток от целочисленного деления одного числа на другое. Например, 10 при делении на 3 дает 3 с остатком 1. В SQL есть два альтернативных способа найти остаток от деления: с помощью оператора % или функции `mod`:

```
SELECT 123456 % 100 as mod_100;

mod_100
-------
56

SELECT mod(123456,100) as mod_100;

mod_100
-------
56
```

Оба варианта возвращают один и тот же результат — `56`, который равен двум последним цифрам исходного значения `123456`. Чтобы создать выборку из 1% набора данных, поместите любое из этих выражений в раздел `WHERE` и проверяйте его на равенство любому целому числу, например 7:

```
SELECT user_id, ...
FROM table
WHERE user_id % 100 = 7
;
```

Остаток от деления на 100 дает выборку в 1%, тогда как остаток от деления на 1000 дает выборку в 0.1%, а остаток от деления на 10 — выборку в 10%. Хотя выборка по остатку от деления на целое, кратное 10, является стандартной, но можно делить на любое целое число.

Выборка для идентификаторов, которые состоят из букв и цифр, не так проста, как для числовых идентификаторов. Можно использовать функции разбора строк для получения нескольких первых или последних символов, и уже к ним применить

фильтры. Например, мы можем выбрать только идентификаторы, заканчивающиеся на букву 'b', получив символ в конце строки с помощью функции right:

```
SELECT user_id, ...
FROM table
WHERE right(user_id,1) = 'b'
;
```

Предполагая, что последним символом может быть любая заглавная или строчная буква английского алфавита или любая цифра, мы получаем в выборке примерно в 1.6% (1/62) набора данных. Чтобы получить бóльшую выборку, укажите в фильтре несколько допустимых значений:

```
SELECT user_id, ...
FROM table
WHERE right(user_id,1) in ('b','f','m')
;
```

Чтобы создать выборку меньшего размера, включите в фильтр несколько последних символов идентификатора:

```
SELECT user_id, ...
FROM table
WHERE right(user_id,2) = 'c3'
;
```

**При формировании выборки нужно убедиться, что функция, которую вы используете, действительно создает случайную или близкую к случайной выборку данных. На одном из моих предыдущих проектов мы обнаружили, что у определенных типов пользователей идентификаторы чаще всего имели конкретные комбинации последних двух цифр. В этом случае использование функции mod для создания выборки в 1% приводило к заметному смещению результатов. Буквенно-цифровые идентификаторы, как правило, имеют общие шаблоны в начале или конце строки, и это можно выявить заранее при профилировании данных.**

Выборка — это простой способ уменьшения размера набора данных на порядки. Она позволяет ускорить вычисления SQL-запросов и сделать конечный результат более компактным, что облегчает его перенос в другой инструмент. Однако иногда потеря детализации при выборке недопустима, и тогда применяются другие методы.

## Уменьшение размерности

Количество различных комбинаций значений атрибутов — *размерность* — сильно влияет на количество записей в наборе данных. Чтобы понять это, давайте рассмотрим простой пример. Представьте, что у нас есть поле с 10 различными значениями, и мы подсчитываем количество записей с помощью функции count и выполняем

группировку GROUP BY по этому полю. Запрос вернет 10 записей. Теперь добавим в запрос второе поле, тоже с 10 различными значениями, подсчитаем количество записей и выполним группировку по этим двум полям. Запрос вернет уже 100 записей. Если мы добавим третье поле с 10 значениями, то получим 1000 записей. Даже если не все возможные комбинации значений трех полей встречаются в запрашиваемой таблице, очевидно, что добавление дополнительных полей в запрос может значительно увеличить размер результирующего набора данных.

При выполнении анализа мы, как правило, можем контролировать количество возвращаемых полей и фильтровать необходимые значения, чтобы в итоге получить управляемый результат. Однако при подготовке наборов данных для дальнейшего анализа в других инструментах часто требуется обеспечить достаточную гибкость и, следовательно, включить множество различных атрибутов и вычислений. Чтобы сохранить как можно больше детализации, но при этом управлять общим размером набора данных, мы можем использовать несколько способов группировки.

Изменение уровня детализации даты и времени является самым простым методом уменьшения объема данных. Поговорите со своими заинтересованными лицами и уточните, действительно ли им нужны, например, ежедневные данные — возможно, им будет вполне достаточно еженедельных или ежемесячных агрегирований. Группировка данных по месяцам и по дням недели может быть хорошим решением, чтобы обеспечить видимость разных шаблонов поведений, характерных для будних дней и для выходных. Ограничение возвращаемого временного промежутка — это тоже один из способов, но он может затруднить изучение долгосрочных тенденций. Я сталкивалась с вариантом, когда команда по работе с данными предоставляла два набора данных: один с агрегированием по месяцам за несколько лет, а второй набор включал в себя те же атрибуты, но с данными по дням или даже по часам за гораздо более короткий промежуток времени.

Текстовые поля — еще одно место, которое стоит проверить на предмет возможной экономии объема данных. Различия в написании или в регистре символов могут привести к появлению гораздо бóльшего количества лишних значений, чем это требуется. Применение функций, рассмотренных в *гл. 5*, таких как `lower`, `trim` или `initcap`, стандартизирует текстовые значения и обычно делает данные более полезными для заинтересованных сторон. Для более точечных корректировок можно использовать функцию `replace` или оператор `CASE`, например для исправления орфографических ошибок или замены старого значения на новое.

Иногда только несколько значений из длинного списка интересны для анализа, поэтому более эффективным будет сохранение детализации только для них, а остальные значения можно сгруппировать вместе. Я часто сталкивалась с этим при работе с географическими регионами. В мире насчитывается более двухсот стран, но, как правило, только в нескольких странах есть достаточное количество клиентов или других точек данных, чтобы имело смысл делать по ним отдельную отчетность. Набор данных о законодателях, использованный в *гл. 4*, содержит 59 значений штата, что включает 50 штатов и остальные территории США, где есть представители. Например, мы хотим создать набор данных с детализацией по пяти штатам с наи-

большим населением (в настоящее время это Калифорния, Техас, Флорида, Нью-Йорк и Пенсильвания), а остальные штаты сгруппировать в категорию 'Other' (другие) с помощью оператора 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
```

Запрос возвращает только 6 строк, а не 59, что представляет собой значительное уменьшение количества записей. Чтобы сделать список более динамичным, мы можем сначала ранжировать значения в подзапросе, в данном случае по уникальным значениям id_bioguide (идентификатор законодателя), а затем вернуть первые пять штатов и 'Other' для остальных штатов:

```
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
```

В этом втором списке несколько штатов изменились. Если мы продолжим обновлять набор данных свежими точками данных, динамический запрос гарантирует, что результат всегда будет отображать пять актуальных верхних значений.

Размерность также может быть уменьшена с помощью преобразования значений данных во флаги. Флаги обычно являются двоичными (т. е. имеют только два значения). Для флагов можно использовать или логический тип BOOLEAN со значениями TRUE и FALSE, или 1 и 0, «Да» и «Нет», или любую другую пару значимых строк. Флаги удобны, когда важно, достигнуто ли пороговое значение, а детализация за его пределами уже не так интересна. Например, мы можем захотеть узнать, совершил ли посетитель сайта покупку или нет, но подробности о точном количестве покупок менее важны.

Законодатели в нашем наборе данных занимали разное количество сроков. Однако вместо точного количества сроков в результирующем наборе нам, возможно, достаточно будет информации о том, занимал ли законодатель более одного срока, для этого можно добавить новый флаг:

```
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
```

Примерно в 2 раза больше законодателей проработало не менее двух сроков по сравнению с теми, кто проработал всего один срок. В сочетании с другими полями

в наборе данных это преобразование может привести к значительному сокращению размера набора результатов.

Иногда простого индикатора TRUE/FALSE или «Да»/«Нет» недостаточно, чтобы передать необходимые нюансы. В этом случае числовые данные могут быть разбиты на несколько групп для сохранения дополнительной детализации. Это выполняется с помощью оператора CASE, а возвращаемое значение может быть числом или строкой.

Например, мы можем включить в наш запрос не только информацию о том, переизбирался ли законодатель на второй срок, но и выделить тех, кто переизбирался 10 и более сроков:

```
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
```

Здесь мы сократили множество различных значений до трех групп, сохранив при этом информацию о законодателях с одним сроком, о тех, кто переизбирался несколько раз, и тех, кто исключительно хорош на своем посту. Такие группировки или деления встречаются во многих сферах. Как и в случаях со всеми остальными преобразованиями, вам может потребоваться выполнить несколько проб и ошибок, чтобы найти точные пороговые значения, наиболее значимые для заинтересованных лиц. Нахождение правильного баланса между детализацией и агрегированием может значительно уменьшить размер результирующего набора данных и, следовательно, ускорить время передачи данных и последующую работу в другом инструменте.

## Персональные данные и конфиденциальность

Конфиденциальность данных — один из самых важных вопросов, с которыми сталкиваются специалисты по данным сегодня. Большие наборы данных с множеством атрибутов позволяют проводить более надежный анализ с подробными выводами и рекомендациями. Однако, когда данные касаются отдельных людей, нам необходимо помнить как об этических, так и о нормативных аспектах сбора и использования таких данных. Нормативные документы о конфиденциальности данных пациентов, студентов и клиентов финансовых служб существуют уже много лет. В последние годы также вступили в силу законы, регулирующие права потребителей на конфиденциальность данных. На территории Российской Федерации действует федеральный закон «О персональных данных» № 152-ФЗ. Общий регламент по защите данных (General Data Protection Regulation, GDPR) принят Европейским союзом. Аналогичными нормативными актами являются Калифорнийский закон о конфиденциальности потребителей (California Consumer Privacy Act, CCPA), Австралийские принципы конфиденциальности (Australian Privacy Principles) и Бразильский общий закон о защите данных (Brazil's General Data Protection Law, LGPD).

Эти и другие нормативные акты касаются обработки, хранения и (в ряде случаев) удаления *персональных данных* (ПД, personally identifiable information, PII). Некоторые категории ПД очевидны: имя, адрес, электронная почта, дата рождения и номер социального страхования. ПД также включают в себя медицинские показатели, такие как пульс, кровяное давление и диагнозы. Информация о местоположении, например координаты GPS, также считается ПД, поскольку по нескольким точкам координат GPS может однозначно идентифицировать человека. Например, по GPS координатам моего дома и школы моих детей можно однозначно идентифицировать кого-то из членов моей семьи. Третья точка GPS в моем офисе может однозначно идентифицировать меня. Вам, как специалисту по работе с данными, стоит ознакомиться с тем, что охватывают эти нормативные акты, и обсудить с юристами вашей организации, которые располагают самой свежей информацией, как эти документы могут повлиять на вашу работу.

Лучшей практикой при анализе данных, содержащих в себя ПД, считается исключение этих самых ПД из результирующего набора. Этого можно добиться с помощью агрегирования данных, замены или хеширования значений.

Для большинства анализов целью является поиск тенденций и закономерностей. Очень часто необходим простой подсчет клиентов и усреднение их поведения, а не выяснение персональных данных. При агрегировании ПД обычно удаляются, однако следует помнить, что по комбинации атрибутов, для которой количество клиентов равно 1, потенциально можно идентифицировать конкретного человека. Такие точки данных можно рассматривать как выбросы и удалять из результатов, чтобы сохранить высокую степень конфиденциальности.

Если индивидуальные данные необходимы по каким-либо причинам, например для поиска отдельных пользователей в последующем инструменте, мы можем заменить

опасные данные случайными альтернативными значениями, для которых будет сохраняться уникальность. Оконная функция `row_number` может быть использована для присвоения нового значения каждому пользователю в таблице:

```
SELECT email
,row_number() over (order by ...)
FROM users
;
```

В этом случае проблема заключается в том, чтобы выбрать поле, которое можно поместить в `ORDER BY`, чтобы упорядочивание в оконной функции было достаточно случайным и мы могли бы считать полученный идентификатор пользователя анонимным.

Еще одним способом является хэширование значений. Хеширование принимает входное значение и с помощью алгоритма создает новое выходное значение. При этом каждое входное значение всегда будет давать один и тот же результат, что делает хэширование хорошим способом сохранения уникальности и сокрытия конфиденциальных данных. Для генерации хэшированного значения может быть использована функция `md5`:

```
SELECT md5('my info');


md5
--------------------------------
0fb1d3f29f5dd1b7cabbad56cf043d1a
```

Функция `md5` хэширует входные значения, но не шифрует их, и поэтому ее можно обратить, чтобы получить исходное значение. Для очень чувствительных данных вам следует поговорить с администратором базы данных, чтобы действительно зашифровать такие данные.

Всегда лучше отказаться от использования персональных данных в результатах SQL-запросов, если это возможно, поскольку тем самым вы препятствуете их распространению по разным системам и файлам. Замена или хэширование значений является еще одним отличным способом защиты ПД. Вы также можете освоить безопасные методы обмена данными, например, разработать защищенную передачу данных непосредственно между базой данных и системой электронной почты, чтобы избежать записи адресов электронной почты в файл. При должном соблюдении осторожности и при сотрудничестве с коллегами из технического и юридического отделов можно добиться качественного анализа при сохранении конфиденциальности персональных данных.

## 8.5. Заключение

При выполнении любого анализа необходимо принять ряд дополнительных решений, касающихся упорядочивания кода, управления сложностью, оптимизации производительности и обеспечения конфиденциальности выходных данных. В этой главе мы обсудили несколько способов и стратегий, а также специальный синтаксис SQL, который может помочь в решении ваших задач. Постарайтесь не зацикливаться на этих решениях и не беспокоиться о том, что без уверенного знания всех этих тем вы не сможете стать эффективным аналитиком или специалистом по данным. Не все эти методы применяются в каждом анализе, и часто существуют другие, альтернативные способы выполнения работы. Чем дольше вы занимаетесь анализом данных с помощью SQL, тем больше вероятность того, что вы столкнетесь с ситуациями, в которых пригодится один или несколько этих методов.

ГЛАВА 9

# Комбинирование методов анализа и полезные ресурсы

На протяжении всей книги мы видели, что SQL является достаточно гибким и мощным языком для решения целого ряда задач анализа данных. SQL может решать разнообразные общие задачи — от простого профилирования данных до временных рядов, текстового анализа и выявления аномалий. Кроме того, можно комбинировать методы и функции в любом операторе SQL для проведения анализа экспериментов и построения сложных наборов данных. Хотя с помощью языка SQL нельзя выполнить все задачи, он хорошо вписывается в набор инструментов для анализа.

В этой последней главе я расскажу о нескольких дополнительных видах анализа, при выполнении которых можно комбинировать различные методы SQL, рассмотренные в предыдущих главах книги. В заключение я расскажу о некоторых полезных ресурсах, которыми вы можете воспользоваться для продолжения освоения анализа данных или для более глубокого изучения конкретных тем.

## 9.1. Анализ воронки продаж

Воронка продаж состоит из последовательности шагов, которые необходимо выполнить для достижения определенной цели. Конечной целью может быть регистрация в сервисе, совершение покупки или получение сертификата об окончании курса. Например, шаги воронки продаж на сайте могут включать в себя нажатие кнопки «Добавить в корзину», заполнение информации о доставке, ввод данных кредитной карты и, наконец, нажатие кнопки «Оформить заказ».

*Анализ воронки продаж* сочетает в себе методы анализа временных рядов, рассмотренного в *гл. 3*, и когортного анализа, рассмотренного в *гл. 4*. Данные для анализа воронки продаж берутся из временного ряда событий, хотя в этом случае события представляют собой разные действия в реальном мире, а не являются одним и тем же повторяющимся действием. Измерение показателя удержания от шага к шагу является ключевой целью анализа воронки продаж, хотя в данном контексте часто используется термин *конверсия*. Как правило, сущности отсеиваются на всех этапах

процесса, и если построить график их количества на каждом этапе, то он по форме будет напоминать воронку — отсюда и название.

Этот вид анализа используется для выявления мест, вызывающих противоречия, трудности или путаницу. Этапы, на которых большое количество пользователей отсеивается или застревает, являются возможными кандидатами для проведения оптимизации. Например, процесс оформления заказа, в котором информация о кредитной карте запрашивается до того, как показывается общая сумма с учетом доставки, может оттолкнуть некоторых потенциальных покупателей. Отображение общей суммы заказа перед вводом кредитной карты может привести к большему проценту завершений покупки. Такие изменения часто становятся предметом анализа экспериментов, о котором мы говорили в *гл. 7*. Воронки продаж также помогают отслеживать непредвиденные внешние события. Например, изменение коэффициента завершенных покупок может соответствовать хорошей (или плохой) маркетинговой кампании или изменениям в ценообразовании у конкурента.

Первым шагом анализа воронки продаж является определение базовой совокупности всех пользователей, клиентов или других сущностей, у которых была возможность участвовать в процессе. Затем нужно собрать набор данных о завершении каждого интересующего нас этапа, включая и конечную цель. Здесь часто используется один или несколько `LEFT JOIN`, чтобы работать со всей базовой совокупностью пользователей, а не только с теми, кто дошел до последнего шага. Затем нужно подсчитать количество пользователей на каждом шаге и разделить эти значения на общее количество пользователей. Существуют два способа построения запроса, в зависимости от того, все ли шаги являются обязательными для пользователей.

Если все шаги в воронке продаж обязательные (или если вы хотите найти пользователей, которые выполнили все шаги), присоединяйте каждую последующую таблицу к предыдущей с помощью `LEFT JOIN`:

```
SELECT count(a.user_id) as all_users
,count(b.user_id) as step_one_users
,count(b.user_id) / count(a.user_id) as pct_step_one
,count(c.user_id) as step_two_users
,count(c.user_id) / count(b.user_id) as pct_one_to_two
FROM users a
LEFT JOIN step_one b on a.user_id = b.user_id
LEFT JOIN step_two c on b.user_id = c.user_id
;
```

Если пользователи могут пропускать шаги или если вы хотите предусмотреть такую возможность, присоединяйте каждую последующую таблицу к первой таблице, содержащей всех пользователей, и вычисляйте доли от этой начальной группы:

```
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(a.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
;
```

Здесь очень тонкое различие, но на него нужно обратить внимание и адаптировать к вашему конкретному случаю. Рассмотрите возможность добавления временных рамок, чтобы учитывать только тех пользователей, которые завершили действие в течение определенного временного периода, если пользователи могут снова зайти в воронку продаж после длительного отсутствия. Анализ воронки продаж может также включать в себя дополнительные срезы, например по когортам или по другому атрибуту сущности, чтобы упростить сравнение и генерировать новые гипотезы о том, почему воронка продаж работает так, как работает.

## 9.2. Отток, отставшие и анализ разрывов

Отток уже упоминался в *гл. 4*, поскольку он, по сути, противоположен удержанию. Как правило, организации хотят или должны выбрать конкретное определение *оттока*, чтобы можно было непосредственно измерять его. В некоторых случаях у нас есть указанная в договоре дата окончания взаимодействия, как, например, в случае с программным обеспечением B2B. Но очень часто отток — более расплывчатое понятие, и тогда лучше подойдет определение, основанное на временном периоде. Даже при наличии даты окончания в договоре оценка того, как давно клиент перестал пользоваться продуктом, может заранее предупредить о скором расторжении договора. Определение оттока также может быть применено к конкретным продуктам или характеристикам, даже если клиент не уходит полностью.

Показатель оттока, основанный на временном периоде, подсчитывает клиентов, которые не покупали продукт или не взаимодействовали с организацией в течение заданного периода времени — обычно от 30 дней до года. Выбираемый интервал во многом зависит от вида бизнеса и от типичных шаблонов поведения. Чтобы выбрать подходящее определение оттока, вы можете использовать *анализ разрывов* для нахождения типичных интервалов между покупками или пользования сервисом. Для проведения анализа разрыва вам понадобятся временной ряд действий или событий, оконная функция `lag` и некоторые функции для дат.

Например, мы можем рассчитать типичные интервалы между сроками полномочий представителей, используя набор данных о законодателях из *гл. 4*. Мы проигнорируем тот факт, что политики часто уходят в отставку из-за результатов выборов, а не добровольно. В остальном же этот набор данных имеет подходящую структуру для такого типа анализа. Сначала мы найдем среднее значение интервала. Для этого мы создадим подзапрос, который для каждого законодателя и для каждого срока

вычислит временной интервал между start_date и start_date предыдущего срока, а затем вычислим среднее значение интервала во внешнем запросе. Дату start_date предыдущего срока можно найти с помощью оконной функции lag, а разрыв как временной интервал рассчитывается с помощью функции 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
```

Как и следовало ожидать, среднее значение близко к двум годам, что вполне логично, поскольку продолжительность срока для этой должности составляет 2 года. Мы также можем построить распределение разрывов, чтобы выбрать реалистичный порог оттока. Для этого мы преобразуем интервал в месяцы:

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

Если функция `date_part` не поддерживается вашей базой данных, можно использовать функцию `extract` (*см. разд. 3.1*). По этим результатам можно построить график, показанный на рис. 9.1. Поскольку у распределения есть длинный хвост справа, то этот график увеличен, чтобы показать диапазон, в который попадает большинство разрывов. Самый часто встречающийся разрыв составляет 24 месяца, но вплоть до 32 месяцев количество интервалов достигает нескольких сотен случаев. При значениях разрыва в 47 и 48 месяцев наблюдается еще один небольшой скачок до 100 и более случаев. Учитывая среднее значение и это распределение, я бы, вероятно, установила порог оттока в 36 или 48 месяцев и считала бы, что любой представитель, который не был переизбран в течение этого интервала, «выбыл».

После определения порога оттока вы можете отслеживать клиентскую базу с помощью анализа «времени с момента последнего обращения». Это может относиться к последней сделанной покупке, последнему платежу, последнему открытию приложения или любой другой метрике, основанной на времени и значимой для организации. Для такого расчета вам необходимо иметь набор данных, содержащий самую последнюю дату или временную метку для каждого клиента. Если вы работаете с временным рядом, сначала найдите самую последнюю временную метку

для каждого клиента в подзапросе. Затем примените функции для дат, чтобы найти время, прошедшее с этой последней даты до текущей даты или до последней даты в наборе данных, если с момента сбора данных прошло некоторое время.

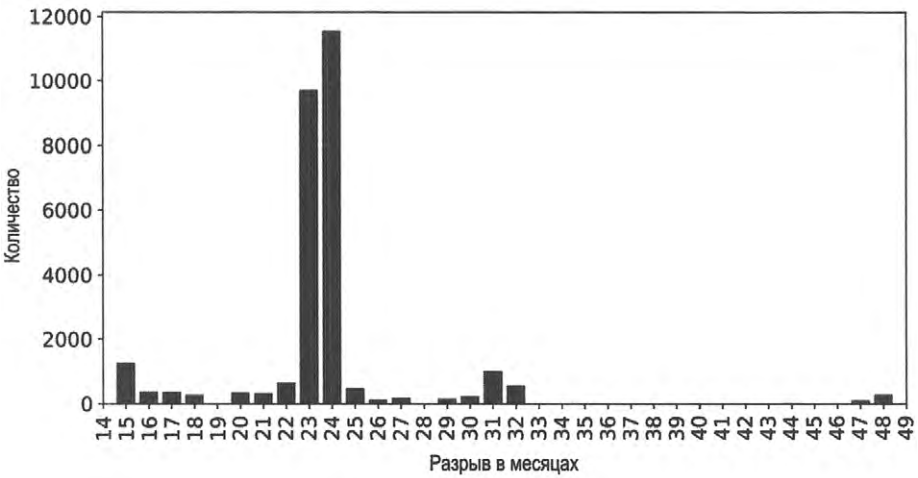


**Рис. 9.1.** Распределение разрыва между датами начала сроков, показан диапазон от 10 до 59 месяцев

Например, для законодателей мы можем получить распределение лет, прошедших с момента их последнего избрания. В подзапросе вычислим самую позднюю начальную дату с помощью функции `max`, а затем найдем временной интервал, прошедшие с тех пор с помощью функции `age`. В данном случае я использую максимальную дату в наборе данных — 19 мая 2020 г. В актуальном наборе данных замените эту дату на `current_date` или на эквивалентное выражение. Внешний запрос возвращает количество лет в интервале с помощью функции `date_part` и подсчитывает количество законодателей:

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

Связанное понятие «отставший» (lapsed) часто используется для обозначения промежуточного статуса клиента между полностью активными и ушедшими клиентами. Таких клиентов также называют «спящими». Отставший клиент имеет более высокий риск попасть в отток, потому что мы не видели его некоторое время, но все еще есть хорошая вероятность его возвращения, основанная на нашем с ним прошлом взаимодействии. В сфере потребительских услуг я видела, что «отставшим» считается клиент, не пользующийся услугой от 7 до 30 дней, а «ушедшим» — более 30 дней. Компании часто экспериментируют с реактивацией отставших пользователей, используя различные тактики — от электронных рассылок до работы службы поддержки. Чтобы найти всех таких клиентов, сначала нужно найти их «время с момента последнего обращения», а затем присвоить им статус с помощью оператора CASE, используя соответствующее количество дней или месяцев. Например, мы можем сгруппировать представителей в зависимости от того, как давно они были избраны:

```
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
```

Этот набор данных содержит сроки полномочий законодателей за более чем двухсотлетний период, поэтому, конечно, многие из людей уже умерли, а некоторые все еще живы, но вышли на пенсию. Конечно, в сфере бизнеса мы надеемся, что количество ушедших клиентов не будет так сильно превышать количество наших активных клиентов, и мы хотим знать больше об отставших клиентах.

Многие организации очень беспокоятся об оттоке клиентов, поскольку привлечь их, как правило, дороже, чем удержать. Чтобы больше узнать о клиентах в каждом статусе или о промежутках времени с момента последнего обращения, такие анализы могут быть дополнительно выполнены в разрезе любого атрибута, доступного в наборе данных.

## 9.3. Анализ потребительской корзины

У меня трое детей, и когда я иду в продуктовый магазин, моя корзина (или, чаще всего, тележка) быстро наполняется продуктами, чтобы нам хватило на неделю. Обычно там есть молоко, яйца и хлеб, а другие продукты могут меняться в зависимости от того, какой сейчас время года, ходят ли дети в школу или они на каникулах, или мы планируем приготовить особое блюдо. *Анализ потребительской корзины* получил свое название из-за практики анализа товаров, которые потребители покупают одновременно, чтобы найти закономерности, которые можно использовать для маркетинга, размещения на полках магазина или принятия других стратегических решений. Целью анализа потребительской корзины может быть поиск групп товаров, покупаемых вместе. Он также может быть сфокусирован на конкретном продукте: когда кто-то покупает мороженое, что еще он покупает?

Хотя анализ потребительской корзины изначально был ориентирован на товары, купленные вместе в рамках одной транзакции, его концепция может быть расширена несколькими способами. Розничный или интернет-магазин может интересоваться корзиной с товарами, которые покупатель приобретает в течение всей своей жизни. Услуги и аксессуары также можно анализировать подобным образом. Услуги, которые обычно покупаются вместе, могут быть объединены в новое предложение, например, когда туристические сайты делают специальное предложение на пакет из перелета, отеля и аренды автомобиля. Аксессуары, которые могут приме-

няться вместе с товаром, можно размещать рядом на одной странице или использовать для составления рекомендаций. Анализ потребительской корзины также может помочь с определением групп клиентов или сегментов, которые затем будут использоваться в других видах анализа.

Чтобы найти наиболее распространенную потребительскую корзину, учитывая все товары в корзине, мы можем использовать текстовую функцию `string_agg` (или эквивалентную, в зависимости от типа базы данных, *см. гл. 5*). Представьте, например, что у нас есть таблица покупок `purchases`, в которой есть по строке для каждого товара `product`, купленного клиентом `customer_id`. Сначала в подзапросе используем функцию `string_agg`, чтобы составить список товаров, купленных каждым клиентом. Затем выполним `GROUP BY` по этим спискам и подсчитаем количество клиентов:

```
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
;
```

Этот метод хорошо работает, когда у вас есть относительно небольшое количество возможных товаров. Другой вариант анализа — найти пары товаров, купленных вместе. Для этого нужно сделать self-JOIN таблицы `purchases` с самой собой, используя идентификатор `customer_id`. Второе условие в `JOIN` решает проблему дублирования записей, которые отличаются только порядком. Например, если клиент купил яблоки и бананы, то без второго условия результат будет включать две записи `'яблоки, бананы'` и `'бананы, яблоки'`. Условие `b.product > a.product` гарантирует возвращение только одной из этих записей, а также отфильтровывает пары, которые товар составляет сам с собой:

```
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
;
```

Этот запрос можно расширить до трех и более товаров, добавив дополнительные JOIN. Чтобы включить в рассмотрение корзины, содержащие только один товар, замените JOIN на LEFT JOIN.

При проведении анализа потребительской корзины обычно возникают несколько проблем. Первая — это производительность, особенно при наличии длинного каталога продуктов, услуг или аксессуаров. В результате выполнение запроса в базе данных может стать медленным, особенно если целью является поиск групп из трех или более товаров и, следовательно, SQL содержит три или более самосоединения. Можно использовать дополнительную фильтрацию таблиц с помощью WHERE, чтобы исключить редко покупаемые товары перед выполнением JOIN. Еще одна проблема возникает, когда некоторые товары покупают так часто, что они «топят» все остальные группы товаров. Например, молоко покупают так часто, что группы с ним и любым другим товаром всегда занимают первые места в анализе потребительской корзины. Такие результаты хотя и являются точными, но могут быть бесполезными с практической точки зрения. В этом случае перед выполнением JOIN следует полностью исключить наиболее часто встречающиеся товары, опять же с помощью WHERE. Кроме того, это повысит производительность запроса за счет уменьшения набора данных.

Последняя проблема анализа потребительской корзины — это «самоисполняющееся пророчество». Товары, которые случайно появились вместе в анализе потребительской корзины, могут затем продаваться вместе, увеличивая тем самым частоту их совместных покупок. Это, в свою очередь, подкрепляет аргументы в пользу их дальнейшего совместного продвижения, что приводит к еще большему количеству совместных покупок и т. д. У товаров, которые лучше подходят друг другу, не было никаких шансов просто потому, что они не появились в первоначальном анализе и не участвовали в продвижении. Знаменитая корреляция пива и подгузников — лишь один из примеров. Разные методы машинного обучения и крупные интернет-компании пытались решить эту проблему, и в этом анализе еще много интересных направлений для развития.

## 9.4. Полезные ресурсы

Чтобы заниматься анализом данных профессионально (или даже как хобби), необходимо сочетание технического мастерства, знания предметной области, любознательности и коммуникативных навыков. Я хотела бы поделиться с вами моими любимыми ресурсами, чтобы вы могли продолжить изучение этой темы — и получать новые знания, и практиковаться на реальных наборах данных.

## Книги и блоги

Хотя при написании этой книги я предполагала, что у вас уже есть опыт работы с SQL, хочется упомянуть два хороших ресурса для изучения основ SQL или для повышения квалификации:

- книгу *Бена Форта* «SQL за 10 минут», 5-е изд., М.: Диалектика-Вильямс, 2021 (*Forta, Ben.* Sams Teach Yourself SQL in 10 Minutes a Day. 5th ed. Hoboken, NJ: Sams, 2020);
- учебник по SQL с интерактивным интерфейсом запросов, удобным для отработки навыков, от компании Mode: **mode.com/sql-tutorial**.

Не существует единого общего стиля оформления SQL-кода, но вы можете найти что-то полезное для себя в статьях SQL Style Guide[1] (Руководство по стилю SQL) и Modern SQL Style Guide[2] (Современное руководство по стилю SQL). Обратите внимание, что их стили не совпадают с тем, который был использован в этой книге, и даже отличаются друг от друга. Я считаю, что стиль, который одновременно последователен и удобен для чтения, очень важен при написании кода.

Ваш подход к анализу и оформлению результатов зачастую так же значим, как и код, который вы пишете. Есть две хорошие книги для развития этих навыков:

- *Дуглас Хаббард* «Как измерить все, что угодно. Оценка стоимости нематериального в бизнесе», М.: Олимп-Бизнес, 2009 (*Hubbard, Douglas W.* How to Measure Anything: Finding the Value of "Intangibles" in Business. 2nd ed. Hoboken, NJ: Wiley, 2010);
- *Даниэль Канеман* «Думай медленно... решай быстро», М.: АСТ, 2014 (*Kahneman, Daniel.* Thinking, Fast and Slow. New York: Farrar, Straus and Giroux, 2011).

Блог Towards Data Science (**towardsdatascience.com**) — отличный источник статей по различным темам анализа. Хотя многие статьи в нем посвящены языку программирования Python, подходы и методы часто можно адаптировать к SQL.

Забавный взгляд на корреляцию и причинно-следственные связи можно найти в статье Тайлера Вигена «Ложные корреляции» (**tylervigen.com/spurious-correlations**).

Регулярные выражения могут казаться сложными. Если вы хотите разобраться с ними или научиться решать сложные задачи, которые не разбирались в этой книге, хорошим ресурсом будет:

- *Бен Форта* «Изучаем регулярные выражения», М.: Вильямс, 2019 (*Forta, Ben.* Learning Regular Expressions. Boston: Addison-Wesley, 2018).

Проведение контролируемых экспериментов имеет долгую историю и затрагивает многие области естественных и социальных наук. Но по сравнению со статистикой анализ онлайн-экспериментов все еще относительно молод. Многие учебники по

---

[1] **https://www.sqlstyle.guide/**

[2] **https://gist.github.com/mattmc3/38a85e6a4ca1093816c08d4815fbebfb**

статистике знакомят с темой, но разбирают примеры на очень маленьких выборках, поэтому в них не рассматриваются уникальные особенности и проблемы онлайн-тестирования. Приведу пару хороших книг, в которых обсуждаются онлайн-эксперименты:

- *Георгий Георгиев* «Статистические методы в онлайновом A/B-тестировании» (*Georgiev, Georgi Z.* Statistical Methods in Online A/B Testing. Sofia, Bulgaria: selfpublished, 2019).
- *Рон Кохави, Диана Тан, Я Сюй* «Доверительное A/B-тестирование. Практическое руководство по контролируемым экспериментам», М.: ДМК-Пресс, 2021 (*Kohavi, Ron, Diane Tang, Ya Xu.* Trustworthy Online Controlled Experiments: A Practical Guide to A/B Testing. Cambridge, UK: Cambridge University Press, 2020).

На сайте Awesome A/B Tools (**www.evanmiller.org/ab-testing**) Эвана Миллера есть онлайн-калькуляторы для экспериментов с бинарными и непрерывными результатами, а также несколько других тестов, которые могут быть полезны для проведения экспериментов, выходящих за рамки этой книги.

## Наборы данных

Лучший способ изучить и улучшить свои навыки SQL — это применить их к реальным данным. Если вы работаете и имеете доступ к базе данных в своей организации, это хорошая возможность для начала, поскольку у вас, вероятно, уже есть представление о том, как генерируются эти данные и что они означают. Однако существует множество интересных открытых наборов данных, которые вы можете проанализировать, и они охватывают широкий спектр тем. Далее перечислены несколько сайтов, с которых стоит начать:

- Data Is Plural (**www.data-is-plural.com**) — это информационный бюллетень новых и интересных наборов данных, а архив Data Is Plural (**dataset-finder.netlify.app**) — кладезь разных наборов данных с возможностью поиска.
- FiveThirtyEight (**fivethirtyeight.com**) — это сайт журналистов, которые освещают политику, спорт и науку через призму данных. Наборы данных, на которых основаны их истории, находятся в репозитории FiveThirtyEight GitHub (**github.com/fivethirtyeight/data**).
- Gapminder (**www.gapminder.org/data**) — шведский фонд, который ежегодно публикует данные по многим показателям человеческого и экономического развития, включая данные Всемирного банка.
- Организация Объединенных Наций публикует ряд статистических данных. Департамент по экономическим и социальным вопросам ООН (Department of Economic and Social Affairs) публикует данные о динамике численности населения (**population.un.org/wpp/Download/Standard/Population**) в относительно простом для использования формате.

- На сайте Kaggle проводятся конкурсы по анализу данных, а также имеется библиотека наборов данных (**www.kaggle.com/datasets**), которые можно скачать даже вне официальных конкурсов.
- Многие правительства на всех уровнях, от национальных до местных, приняли движение за открытость данных и публикуют различные статистические данные. Data.gov (**www.data.gov/open-gov**) содержит список сайтов как в США, так и по всему миру, которые могут стать хорошей отправной точкой.

## 9.5. Заключение

Я надеюсь, что вы нашли полезными методы и код, приведенные в этой книге. Я считаю, что важно иметь хорошую базу при использовании инструментов анализа, и в SQL есть много полезных функций и выражений, которые могут сделать ваш анализ более быстрым и точным. Однако развитие навыков анализа — это не только изучение новых модных методик или нового языка программирования. Хороший анализ заключается в том, чтобы задавать правильные вопросы, разбираться в данных и предметной области, применять правильные методы анализа для получения качественных и надежных выводов и, наконец, оформлять результаты анализа таким образом, чтобы они были понятны бизнесу и способствовали принятию решений. Даже после 20 лет работы с SQL мне все еще интересно находить новые способы его применения и анализировать с его помощью новые наборы данных в поиске новых открытий, которые терпеливо ждут своего часа.

# Об авторе

Кэти Танимура одержима идеей обеспечить людей и организации всеми данными, которые необходимы им для достижения успеха. Более 20 лет она занимается анализом данных в самых разных отраслях — от финансов до программного обеспечения B2B и потребительских сервисов. У нее большой опыт выполнения анализа данных с помощью языка SQL в большинстве основных проприетарных баз данных и баз данных с открытым исходным кодом. Она создавала и управляла командами специалистов по работе с данными и их инфраструктурой в нескольких ведущих технологических компаниях. Кэти также часто выступает на крупных конференциях по таким темам, как формирование культуры данных, разработка приложений, управляемая данными, и комплексный анализ данных.

# Об обложке

Животное на обложке книги «SQL для анализа данных» — зеленая цисса (лат. *Cissa chinensis*), которую часто называют обыкновенной зеленой сорокой. Это птица ярко-зеленого цвета, относящаяся к семейству вороновых. Обитающие в низменных вечнозеленых и бамбуковых лесах северо-восточной Индии, центрального Таиланда, Малайзии, Суматры и северо-западного Борнео, птицы этого вида шумные и очень общительные. В дикой природе их можно узнать по оперению нефритового цвета, который контрастирует с красным клювом и черной полосой вдоль глаз. У них также есть хвост с белым кончиком и красноватые крылья.

Шумных зеленых сорок можно узнать по пронзительным крикам, за которыми следует глухой и решительный звук «чуп». Их также часто трудно заметить, потому что они перелетают с дерева на дерево в средних и верхних ярусах леса. Свои гнезда они строят на деревьях, крупных кустарниках и в сплетениях различных вьющихся лиан. Иногда их называют охотничьими циссами, потому что они в основном плотоядны и питаются различными беспозвоночными, а также молодыми птицами, яйцами, мелкими рептилиями и млекопитающими.

Зеленые сороки известны своей способностью менять окраску. Хотя в природе они нефритово-зеленые, в неволе они могут стать бирюзовыми. Свою зеленую окраску они получают благодаря особой структуре перьев, которая создает голубой цвет за счет преломления света, и каротиноидов — желтых, оранжевых и красных пигментов, получаемых из рациона. Длительное воздействие прямого солнечного света разрушает каротиноиды, поэтому птица приобретает бирюзовый цвет.

Зеленая сорока имеет очень большой ареал обитания, и хотя популяция, по-видимому, снижается, сокращение происходит не настолько быстро, чтобы перевести вид в категорию уязвимых. Их текущий природоохранный статус — «Вызывающие наименьшие опасения».

Иллюстрация на обложке выполнена Карен Монтгомери на основе черно-белой гравюры из «Английской циклопедии» (English Cyclopedia).

# Предметный указатель

## Символы

## A



## B



## C



## D



## E



## F



## G



## H



## I

## J



## L



## M



## N



## O



## P



## R



## S



## U

## К



## Л



## М



## Н



## О

## П



## Р



## С

## Т



## У



## Ф

O'REILLY®

# SQL для анализа данных

С ростом объема информации, вычислительных мощностей и облачных хранилищ SQL стал незаменимым инструментом для аналитиков, исследователей и специалистов по обработке данных. Эта книга показывает новые способы улучшения навыков работы с SQL, решения сложных практических задач и максимального использования SQL в рабочих процессах. Вы узнаете, как применять SQL для анализа различных типов данных, выявления аномалий, обработки результатов экспериментов и создания сложных наборов данных. Вы научитесь комбинировать методы SQL для более быстрого достижения целей с помощью простого и понятного кода.

- **Узнайте о ключевых шагах по подготовке данных к анализу**
- **Выполняйте анализ временных рядов**
- **Используйте когортный анализ для изучения изменений групп с течением времени**
- **Научитесь применять мощные функции SQL для анализа текста**
- **Выявляйте отклонения и аномалии в данных**
- **Установите причинно-следственную связь с помощью анализа экспериментов и результатов А/Б-тестирования**
- **Применяйте SQL для оценки эффективности воронки продаж**

**Кэти Танимура** более 20 лет занимается анализом данных в самых разных отраслях, от финансов до программного обеспечения и сферы потребительских услуг. Кэти управляла командами специалистов по анализу данных в нескольких ведущих технологических компаниях. Имеет богатый опыт работы со стандартом SQL, включая наиболее популярные проприетарные базы данных и базы данных с открытым исходным кодом.

«Я не могу сосчитать, сколько раз воскликнул "Ого!", читая эту книгу, несмотря на мой 20-летний опыт в сфере анализа данных и работы с различными реализациями SQL. Я обязательно куплю по одному экземпляру книги для каждого члена моей команды».

**— Стюарт Ким-Браун**, доктор философии, руководитель проектов в области продуктов B2C и SaaS

«Так здорово наконец-то увидеть книгу по SQL, написанную специально для тех, кто занимается анализом данных. Любому начинающему аналитику или специалисту по исследованию данных будет полезно прочитать о различных методах анализа, которые могут быть реализованы с помощью SQL. Подробные примеры и код делают эту книгу как отличным пособием для начинающих, так и справочником для профессионалов».

**— Дэн Вурхис**, руководитель отдела аналитики компании Zillow

ISBN 978-5-9775-0958-9

9 785977 509589

191036, Санкт-Петербург,
Гончарная ул., 20
Тел.: (812) 717-10-50,
339-54-17, 339-54-28
E-mail: mail@bhv.ru
Internet: www.bhv.ru

bhv®