Графовые базы данных

Новые возможности для работы со связанными данными

Ян Робинсон, Джим Вебер, Эмиль Эифрем

O'REILLY®

Ян Робинсон, Джим Вебер, Эмиль Эифрем

Графовые базы данных

НОВЫЕ ВОЗМОЖНОСТИ ДЛЯ РАБОТЫ СО СВЯЗАННЫМИ ДАННЫМИ

Второе издание

Ian Robinson, Jim Webber & Emil Eifrem

Graph Databases

NEW OPPORTUNITIES FOR CONNECTED DATA

Second Edition

Beijing · Cambridge · Farnham · Köln · Sebastopol · Tokyo

O'REILLY®

Ян Робинсон, Джим Вебер, Эмиль Эифрем

Графовые базы данных

НОВЫЕ ВОЗМОЖНОСТИ ДЛЯ РАБОТЫ СО СВЯЗАННЫМИ ДАННЫМИ

Второе издание

Москва, 2016

УДК 004.6
ББК 32.972
Р58

Робинсон Ян, Вебер Джим, Эифрем Эмиль

Р58 Графовые базы данных: новые возможности для работы со связанными данными / пер. с англ. Р. Н. Рагимова; науч. ред. А. Н. Кисилев. – 2-е изд. – М.: ДМК Пресс, 2016. – 256 с.: ил.

ISBN 978-5-97060-201-0

Из книги вы узнаете, как проектировать и реализовывать приложения, основанные на графовых базах данных, привносящих мощь графов в широкий круг прикладных областей. Если вам необходимо уменьшить время выполнения запросов пользователей или создать базу данных, способную приспосабливаться под быстро развивающийся бизнес, эта книга продемонстрирует вам практическое применение графовых моделей.

Второе издание книги содержит обновленные примеры кода и схемы, соответствующие актуальному синтаксису графовой базы данных Neo4j, а также информацию о новом функционале Neo4j.

Издание предназначено для программистов, желающих изучить работу графовых баз данных и научиться максимально использовать их мощь в своей работе.

УДК 004.6
ББК 32.972

Authorized Russian translation of the English edition of Graph Databases, 2nd Edition. This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

Материал, изложенный в данной книге, многократно проверен. Но поскольку вероятность технических ошибок все равно существует, издательство не может гарантировать абсолютную точность и правильность приводимых сведений. В связи с этим издательство не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-1-491-93089-2 (анг.) © 2015 Neo Technology, Inc.
ISBN 978-5-97060-201-0 (рус.) © Оформление, перевод, ДМК Пресс, 2016

Содержание

Пролог **10**

Предисловие **13**

Глава 1. Введение **17**

Что такое граф? 17
Обзор областей применения графов 19
Графовые базы данных 20
Механизмы вычисления графов 22
Преимущества графовых баз данных 23
Производительность 24
Гибкость 24
Оперативность 25
Итоги 25

Глава 2. Варианты хранения взаимосвязанных данных **26**

Недостатки NOSQL-баз данных при работе со взаимосвязями 30
Взаимосвязи в графовых базах данных 35
Итоги 41

Глава 3. Моделирование данных графами **42**

Модели и задачи 42
Графовая модель со свойствами и метками 43
Графовые запросы: введение в Cypher 44
Философия языка Cypher 45
MATCH 47
RETURN 48
Другие фразы языка Cypher 48
Сравнение реляционного и графового моделирования 49
Реляционная модель системы управления 51
Графовое моделирование системы управления 56
Тестирование модели 58
Кросс-модели нескольких прикладных областей 60
Создание графа творчества Шекспира 64

Введение в запросы 66
Определение шаблонов для поиска 68
Ограничение совпадений 70
Обработка результатов 71
Цепочки в запросах 72
Распространенные просчеты при моделировании 73
Проблемы анализа источников электронных писем 73
Первый блин комом? 73
Со второго раза все получится 76
Эволюция прикладной области 79
Идентификация узлов и взаимосвязей 85
Как избежать антишаблонов 85
Итоги 86

Глава 4. Разработка приложений графовых баз данных ... 88

Моделирование данных 88
Описание модели с учетом потребностей приложения 89
Узлы представляют сущности, взаимосвязи формируют структуру 90
Подробные имена или свойства взаимосвязей 91
Моделирование фактов в виде узлов 92
Работа 92
Исполнение ролей 92
Электронная переписка 94
Рецензии 94
Представление комплексных типовых значений в виде узлов 94
Время 95
Хронологическое древо 95
Связанные списки 97
Управление версиями 98
Итеративная и поэтапная разработка 98
Архитектура приложений 100
Встроенная поддержка или сервер 100
Встроенная поддержка Neo4j 100
Серверный режим 102
Серверные расширения 103
Кластеризация 106
Репликация 106
Запись буфера с помощью очередей 107

Глобальные кластеры 107
Балансировка нагрузки 107
Отделение трафика чтения от трафика записи 107
Распределительный кэш 109
Чтение собственных записей 110
Тестирование 111
Разработка модели данных, основанная на тестировании 111
Пример: модель данных социальной сети и ее тестирование 112
Тестирование серверных расширений 116
Тестирование производительности 118
Тесты производительности запросов 119
Тесты производительности приложений 120
Тестирование с помощью репрезентативных данных 121
Планирование производственных мощностей 123
Критерии оптимизации 124
Производительность 124
Калькуляция затрат на увеличение производительности графовой базы данных 125
Варианты оптимизации производительности 125
Избыточность 127
Нагрузка 127
Импорт и массовая загрузка данных 128
Первоначальный импорт 128
Пакетный импорт 130
Итоги 133

Глава 5. Графы в реальном мире 134

Почему выбирают графовые базы данных 134
Типичные примеры использования 136
Социальные сети 136
Рекомендации 137
Геоинформационные системы 138
Управление справочными данными 139
Сети и управление центром обработки данных 139
Авторизация и контроль доступа (для коммуникаций) 141
Реальные примеры 142
Социальные рекомендации (Профессиональная социальная сеть) 142

Модель данных Talent.net 144
Выявление социальных взаимосвязей 145
Поиск коллег с определенными интересами 149
Добавление взаимосвязей WORKED_WITH 152
Авторизация и контроль доступа 155
Модель данных компании TeleGraph 156
Поиск доступных администратору ресурсов 159
Определение доступности ресурса администратору 160
Поиск администраторов по учетной записи 163
Геоинформационные системы и логистика 165
Модель данных Global Post 166
Расчет маршрута 169
Поиск кратчайшего маршрута с помощью Cypher 172
Реализация расчета маршрутов с помощью фреймворка Traversal 175
Итоги 180

Глава 6. Внутреннее устройство графовых баз данных 182

Нативная обработка графов 182
Нативное хранилище графов 186
Программные интерфейсы 192
Программный интерфейс ядра 193
Базовый интерфейс 193
Фреймворк Traversal 194
Нефункциональные характеристики 196
Транзакции 197
Восстанавливаемость 199
Доступность 200
Масштабирование 202
Мощность 203
Задержки 203
Производительность 204
Итоги 206

Глава 7. Интеллектуальный анализ с помощью теории графов 207

Поиск в глубину и ширину 207
Поиск маршрутов с помощью алгоритма Дейкстры 209
Алгоритм A* 217

Теория графов и прогнозное моделирование 218
Триадические замыкания 219
Структурный баланс 221
Локальные перемычки 226
Итоги 228

Приложение А. Обзор NOSQL-баз данных 229
Движение NOSQL 229
ACID или BASE 231
Секторы NOSQL 233
Хранилища документов 233
Хранилища пар ключ-значение 236
Семейства столбцов 239
Запросы или обработка в агрегированных хранилищах 242
Графовые базы данных 243
Графы со свойствами 244
Гиперграфы 245
Триплеты 246

Предметный указатель 249

Об авторах 254

Заключение 255

Пролог

Вездесущие графы, или Рождение известных нам графовых баз данных

Это было в 1999 году, мы работали по 23 часа в сутки. По крайней мере, чувствовали себя именно так. Каждый день приносил очередную новость о сумасшедшей идее, только что получившей финансирование в миллионы долларов. У наших конкурентов были сотни инженеров, а наша команда разработчиков состояла всего из 20 человек. Этого было явно недостаточно, причем 10 из наших инженеров большую часть времени проводили в борьбе с реляционной базой данных.

Нам потребовалось время, чтобы выяснить, почему так происходит. После тщательного анализа архитектуры данных, хранимых нашим приложением управления контентом, выяснилось, что программа должна не просто управлять массой отдельных изолированных *дискретных* элементов данных, но и учитывать *связи* между ними. Несмотря на то что наши дискретные данные легко размещались в таблицах реляционной базы данных, с хранением связей между ними возникли сложности, а запросы, извлекающие их, выполнялись чрезвычайно медленно.

Просто от отчаяния я вместе с Йоханом и Питером, соучредителями компании Neo, начал экспериментировать с другими моделями представления данных, в частности основанными на графах. Нас увлекла идея замены табличной семантической модели данных графо-ориентированной, которая должна была значительно облегчить навигацию по связанным данным. Мы поняли, что, вооружившись графовой моделью, наша команда перестанет тратить половину своего времени на борьбу с базой данных.

Конечно, мы понимали, что не являемся первопроходцами. Теория графов существует уже в течение почти 300 лет, и примеры ее применения в целом ряде разнообразных математических задач широко известны. Естественно, должны быть и базы данных, основанные на графах!

Ну, мы и прочесали с помощью AltaVista[1] весь молодой Интернет и ничего не нашли. Через несколько месяцев бесплодных поисков мы

---

[1] Это будет шоком для молодых читателей, но в истории человечества было время, когда Google не существовал. В те далекие времена землей правили динозавры и поисковые системы AltaVista, Lycos и Excite в основном использовались для поиска электронных порталов, предлагающих через Интернет приобрести корм для домашних животных.

(смело) приступили к созданию с нуля базы данных, которая изначально предназначалась для работы с графами. В нашем представлении она должна была сохранить все проверенные временем функции реляционной базы данных (транзакции, ACID, триггеры и т. д.), но при этом использовать модель данных XXI века. Так родился проект Neo, а вместе с ним графовые базы данных, в том виде, в котором они известны сегодня.

В первом десятилетии нового тысячелетия возникло несколько изменивших мир коммерческих проектов, в том числе Google, Facebook и Twitter. Им присуща одна общая особенность: они сделали ставку на графовую модель представления взаимосвязей между данными, сделав ее основой своего бизнеса. И вот, через 15 лет после появления, графовая модель используется повсеместно.

Facebook, например, базируется на идее, что, несмотря на значимость дискретной информации о людях, их именах, профессиях и т. д., еще большую ценность представляют сведения о *взаимосвязях* между ними. Основатель Facebook Марк Цукерберг (Mark Zuckerberg) построил свою империю, основываясь на понимании моделирования этих взаимосвязей с помощью *социального графа*.

Точно так же основатели компании Google – Ларри Пейдж (Larry Page) и Сергей Брин (Sergey Brin) – придумали, как хранить и обрабатывать не только отдельные веб-документы, но и как связать их между собой. Google создала *граф Интернета*, что и позволило ей стать самой впечатляющей компанией последнего десятилетия.

Сегодня графы успешно используются не только гигантами Интернета. Одна из самых больших в мире логистических компаний использует графовую базу данных для прокладки в режиме реального времени маршрутов отправлений, ведущая авиакомпания использует графы для управления метаданными медиаконтента, и финансовые структуры топ-уровня переводят всю свою инфраструктуру на Neo4j. Практически никому не известные несколько лет назад, графовые базы данных в настоящее время широко применяются в таких областях, как здравоохранение, розничная торговля, газонефтедобыча, средства массовой информации, разработка игр и др., ускоряя развитие каждой из них.

Реализация этих идей потребовала разработки инструментов нового типа, основанных на технологиях управления базами данных, учитывающих взаимосвязи между данными и позволяющих мыслить категориями графов, а это именно те инструменты, о которых мы мечтали, мучась с реляционной базой данных в 1999 году.

Я надеюсь, что эта книга станет вашим путеводителем по прекрасному развивающемуся миру графовых технологий и вдохновит вас на использование графовой базы данных в вашем следующем проекте, чтобы вы тоже смогли ощутить необычайную мощь графов.

Удачи!

Эмиль Эифрем (*Emil Eifrem*),
соучредитель Neo4j и генеральный директор Neo Technology
Менло-Парк, Калифорния
Май 2013 года

Предисловие

Сегодня решения, основанные на графовых базах данных, очень важны для ведения успешного бизнеса: управление, учитывающее сложные и динамичные отношения тесно связанных между собой данных, предоставляет преимущество в конкуренции. Везде, где требуется учитывать взаимосвязи между клиентами, абонентами телефонного центра или сети обработки данных, организаторами представлений и зрителями или же генами и белками, способность воспринимать и анализировать огромные графы тесно взаимосвязанных данных будет играть определяющую роль для компаний, которые превзойдут своих конкурентов в течение следующего десятилетия.

Для данных любого объема и значения графовые базы данных являются лучшим способом представления и извлечения взаимосвязанных данных. Взаимосвязи между данными представляют собой данные, описание и оценка которых нужны нам для понимания, каким образом связаны между собой составляющие элементы. Для достижения такого понимания потребуются идентификация и квалификация взаимосвязей между элементами данных.

Многие крупные корпорации, осознав эту необходимость некоторое время назад, начали создавать собственные запатентованные технологии обработки графов, но мы живем в эпоху, когда доступ к технологиям становится все более демократичным. Сегодня общедоступные графовые базы данных являются реальностью, позволяя основной массе пользователей испытать преимущества работы с взаимосвязанными данными без необходимости инвестировать в разработку собственной графовой инфраструктуры.

Примечательным является возрождение интереса к отображению данных посредством графов, притом что теория графов сама по себе не нова. Теория графов была впервые описана Эйлером в XVIII веке и до сих пор активно исследуется и улучшается математиками, социологами, антропологами и другими специалистами-практиками. Тем не менее только в последние несколько лет теория графов и представление с помощью графов стали применяться к управлению информацией. Графовые базы данных помогли решению важных проблем в области социальных сетей, управления нормативно-справочной информацией, картографии, экспертных рекомендаций и многих других. Такое повышенное внимание к графовым базам данных обусловлено двумя факторами: огромным коммерческим успехом таких компаний, как Facebook, Google и Twitter, бизнес-модели которых основываются

на собственных запатентованных технологиях графов, и появлением общедоступных графовых баз данных.

О втором издании книги

Первое издание этой книги была написано, когда Neo4j 2.0 находился в стадии активного развития и еще не были окончательно закреплены формы меток, индексов и ограничений целостности. Сейчас версия 2.x Neo4j уже просуществовала часть своего жизненного цикла (на момент написания книги доступна версия 2.2, и скоро выйдет версия 2.3) и мы уже с уверенностью можем включить в текст книги новые элементы графовой модели.

Для второго издания этой книги мы пересмотрели все примеры на языке запросов Cypher, чтобы привести их в соответствие с современным синтаксисом языка. Мы добавили метки для запросов и схем, включили пояснения декларативной индексации и дополнительных ограничений целостности Cypher. В другом месте мы добавили дополнительные правила моделирования, переработали описание внутренних свойств Neo4j в соответствии с изменениями его внутренней архитектуры и обновили тестовые примеры, применив в них новейшие инструменты.

О чем эта книга

Цель этой книги – познакомить разработчиков, специалистов по базам данных и лиц, принимающих решение о выборе технологий, с графами и графовыми базами данных, в основном ориентируясь на практическое применение этих технологий. Прочтя эту книгу, вы научитесь применять графовые базы данных на практике. Мы покажем, как графовая модель «формирует» данные и как мы извлекаем, обосновываем, понимаем и *воздействуем на* данные с помощью графовых баз данных. Мы рассмотрим все виды проблем, которые легко решаются при применении графовых баз данных, сопроводив пояснения практическими примерами, и опишем проектирование и реализацию решений, основывающихся на графовых базах данных.

Соглашения

В этой книге приняты следующие типографские соглашения:

Курсив

Выделение новых терминов, URL-адресов, адресов электронной почты, имен файлов и расширений файлов.

Моноширинный шрифт

Используется для оформления программного кода и фрагментов программного кода внутри абзацев, таких как имена переменных и функций, типы данных, переменные окружения, операторы и ключевые слова.

Жирный моноширинный шрифт

Команды или другой текст, который должен быть введен пользователем.

Наклонный моноширинный шрифт

Текст, который должен быть заменен пользовательскими значениями или значениями, определяемыми контекстом.

Так обозначаются советы, предложения и примечания общего характера.

Так обозначаются предупреждения и предостережения.

Использование примеров кода

Сопроводительные материалы (примеры кода, упражнения и т. д.) можно загрузить со страницы https://github.com/iansrobinson/graph-databases-use-cases.

Данная книга призвана оказать вам помощь в решении ваших задач. Вы можете свободно использовать примеры программного кода из этой книги в своих приложениях и в документации. Вам не нужно обращаться в издательство за разрешением, если вы не собираетесь воспроизводить существенные части программного кода. Например, если вы разрабатываете программу и используете в ней несколько отрывков программного кода из книги, вам не нужно обращаться за разрешением. Однако в случае продажи или распространения компакт-дисков с примерами из этой книги вам необходимо получить разрешение от издательства O'Reilly. Если вы отвечаете на вопросы, цитируя данную книгу или примеры из нее, получения разрешения не требуется. Но при включении существенных объемов программного кода примеров из этой книги в вашу документацию необходимо получить разрешение издательства.

Мы приветствуем, но не требуем добавлять ссылку на первоисточник при цитировании. Под ссылкой на первоисточник мы подразумеваем указание авторов, издательства и ISBN. Например: «Graph Databases by Ian Robinson, Jim Webber, and Emil Eifrem (O'Reilly). Copyright 2015 Neo Technology, Inc., 978-1-491-93089-2».

За получением разрешения на использование значительных объемов программного кода примеров из этой книги обращайтесь по адресу permissions@oreilly.com.

Как связаться с нами

С вопросами и предложениями, касающимися этой книги, обращайтесь в издательство:

O'Reilly Media, Inc.
1005 Gravenstein Highway North Sebastopol, CA 95472
800-998-9938 (США или Канада)
707-829-0515 (международный и местный)
707-829-0104 (факс)

Список опечаток, файлы с примерами и другую дополнительную информацию вы найдете на странице книги http://bit.ly/modern_php.

Свои пожелания и вопросы технического характера отправляйте по адресу bookquestions@oreilly.com.

Ищите нас в Facebook: http://facebook.com/oreilly.

Следуйте за нами в Twitter: http://twitter.com/oreillymedia.

Смотрите нас на YouTube: http://www.youtube.com/oreillymedia.

Благодарности

Мы хотели бы поблагодарить наших технических рецензентов: Майкла Хангера (Michael Hunger), Колина Джека (Colin Jack), Марка Нидхама (Mark Needham) и Прамода Садалага (Pramod Sadalage).

Выражаем признательность и благодарность редактору первого издания Натану Джепсону (Nathan Jepson).

Наши коллеги из Neo Technology внесли свой вклад в написание этой книги, потратив на это массу времени и сил. В частности, спасибо Андерсу Наврозу (Anders Nawroth) за его неоценимую помощь, Андресу Тейлору (Andrés Taylor) за помощь во всем, что касается Cypher, и Филиппу Расли (Philip Rathle) за его полезные советы.

Большое спасибо всем членам Neo4j-сообщества за огромный вклад в разработку графовой базы данных.

Отдельное спасибо нашим семьям за их любовь и поддержку: Лотти, Тигер, Эллиот, Кэт, Билли, Мадлен и Нооми.

Это второе издание стало возможным благодаря работе Кристины Эскалант (Cristina Escalante) и Майкла Хангера (Michael Hunger). Спасибо вам обоим за вашу неоценимую помощь.

Глава 1

Введение

Большая часть книги посвящена графовым моделям данных, но она не рассказывает о теории графов[1]. Для использования графовых баз данных не требуются глубокие теоретические познания, достаточно наличия общих представлений о графах. Давайте освежим наши сведения о графах.

Что такое граф?

Формально граф – это набор *вершин* и *ребер*, или, говоря более простым языком, набор *узлов* и *взаимосвязей* между ними. В графах объекты представлены узлами, а способы, которыми эти объекты соединены между собой, – взаимосвязями. Эта универсальная и выразительная структура позволяет моделировать всевозможные сценарии, от постройки космической ракеты до строительства системы дорог, от поставок продуктов питания до историй болезни населения, и многое другое.

Графы повсюду

Графы чрезвычайно полезны при анализе самых разных наборов данных в таких областях, как наука, государственное управление и бизнес. Реальный мир, в отличие от основанной на шаблонах устаревшей модели реляционных баз данных, разнообразен и взаимосвязан: в одних местах равномерно и упорядочено, в других случайно и нерегулярно. После освоения графов вы начнете замечать их присутствие повсюду. Гартнер (Gartner, http://www.gartner.com/id=2081316), например, выделяет в мире бизнеса пять видов графов: социальный, целевой, потребления, интересов и мобильности – и утверждает, что способность использовать эти графы обеспечивает «устойчивое преимущество в конкурентной среде».

---

[1] Информацию о теории графов можно найти в книгах Ричарда Дж. Труде (Richard J. Trudeau), «Introduction To Graph Theory» (Dover, 1993) и Гарри Чартранд (Gary Chartrand), «Introductory Graph Theory» (Dover, 1985). О применении теории графов для анализа сложных событий и поведения можно узнать из книги Дэвида Исли (David Easley) и Джон Клейнберг (Jon Kleinberg) «Networks, Crowds, and Markets: Reasoning about a Highly Connected World» (Cambridge University Press, 2010).

Например, данные в Twitter легко представить в виде графа. На рис. 1.1 изображена небольшая сеть пользователей Twitter. Каждый узел помечен как «пользователь», с указанием его роли в сети. Узлы соединены взаимосвязями, определяющими дополнительный семантический контекст, а именно: Билли следует за Гарри, а Гарри, в свою очередь, следует за Билли. Рут и Гарри так же следуют друг за другом, но, к сожалению, хотя Рут следует с Билли, Билли (пока) не ответил взаимностью.

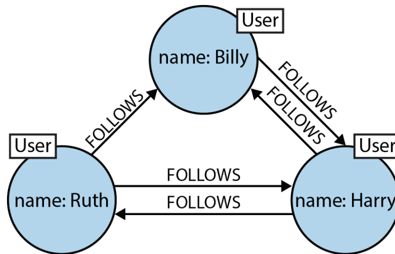


Рис. 1.1 ❖ Малый социальный граф

Конечно же, реальный граф Twitter в сотни миллионов раз больше, чем изображенный на рис. 1.1, но работает по тому же принципу. На рис. 1.2 мы расширили граф, включив в него сообщения, опубликованные Рут.

Несмотря на свою простоту, рис. 1.2 демонстрирует выразительную мощь графового моделирования. Из него легко понять, что Рут опубликовала последовательность сообщений. Ее самое последнее сообщение можно найти по взаимосвязи, помеченной как ТЕКУЩЕЕ. Далее следуют сообщения Рут, помеченные как ПРЕДЫДУЩЕЕ.

Модель графов с метками и свойствами

При обсуждении рис. 1.2 мы неформально представили самую популярную графовую модель – *графовую модель с метками и свойствами* (в приложении A более подробно будут рассмотрены альтернативные графовые модели). Графовая модель с метками имеет следующие характеристики:

- содержит узлы и взаимосвязи;
- у узлов есть свойства (пары ключ-значение);
- узлы должны быть помечены одной или более метками;
- взаимосвязи имеют имя и направление, для них всегда определен начальный и конечный узлы;
- у взаимосвязей также имеются свойства.

Большинство считает, что графовая модель со свойствами является простой и интуитивно понятной. Но, несмотря на свою простоту, именно она используется в подавляющем большинстве случаев при применении графов для отображения данных.

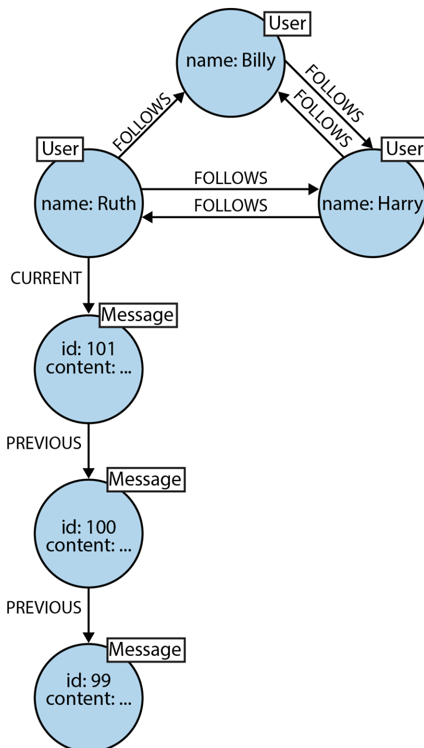


Рис. 1.2 ❖ Опубликованные сообщения

Обзор областей применения графов

В последние годы появилось множество проектов и готовых решений для управления, обработки и анализа графов. Огромное их количество затрудняет мониторинг инструментов и различий между ними, даже для тех, кто активно ими пользуется. Этот раздел содержит обобщенную их классификацию, чтобы помочь сформировать представление об областях применения графов.

С высоты птичьего полета области применения графов можно разделить на две части:

1) *технологии организации транзакционных графовых хранилищ, как правило, используемых в режиме реального времени непосредственно из приложения*. Эти технологии называются *графовыми базами данных*, и им в этой книге будет уделено основное

внимание. Они являются аналогом «нормальной» обработки транзакций в масштабе реального времени (On-Line Transaction Processing, OLTP) реляционных баз данных.

2) *технологии автономного анализа графов, обычно реализующие серии пакетных шагов*. Эти технологии часто называют *механизмами вычисления графов* (*graph compute engine*). Их можно отнести к той же категории, что и другие приемы анализа объемных данных, например интеллектуальный анализ данных и аналитическая обработка данных в реальном времени (On-Line Analytical Processing, OLAP).

Существует и еще один способ классификации применения графов, основанный на видах графовых моделей. Имеются три доминирующие графовые модели: графы со свойствами, триплексы схем описания ресурсов (Resource Description Framework, RDF) и гиперграфы. Они подробно рассмотрены в приложении А. Большая часть популярных графовых баз данных, представленных на рынке, использует графовые модели со свойствами, поэтому именно эта модель и будет использована в остальной части этой книги.

Графовые базы данных

Система управления графовыми базами данных (далее *графовые базы данных*) поддерживает методы создания (Create), чтения (Read), изменения (Update) и удаления (Delete) (CRUD), основанные на графовой модели данных. Графовые базы данных, как правило, поддерживают систему транзакций реального времени (OLTP). Соответственно, они оптимизированы для выполнения транзакций и спроектированы с учетом транзакционной целостности и оперативности.

Имеются две особенности графовых баз данных, которые необходимо учитывать при рассмотрении применяемой ими технологии:

1) **принцип хранения.** Некоторые графовые базы данных используют *специализированные хранилища графов*, предназначенные и оптимизированные для хранения и обработки именно графов. Но такую технологию хранения используют не все графовые базы данных. Некоторые сериализуют графы и размещают их в реляционной, объектно-ориентированной или какой-то другой базе данных или хранилище;
2) **порядок обработки.** Некоторые определения требуют, чтобы графовая база данных использовала *смежность без индесов*

(*index-free adjacency*), т. е. физическое соединение узлов друг с другом[1]. Мы придерживаемся более широких взглядов: любую базу данных, которая с точки зрения пользователя *ведет* себя как графовая (т. е. предоставляет графовую модель данных через CRUD-операции), квалифицируется как графовая база данных. Но мы признаем значительные преимущества в производительности при использовании смежности без индексов и, следовательно, используем термин *специализированная обработка графов* для графовых баз данных, использующих смежность без индексов.

Важно отметить, что принципы специализированного хранения графов и специализированной обработки графов не хороши и не плохи – они просто являются классическими инженерными компромиссами. Преимущество специализированного хранения графов – в том, что оно предусматривает стек, специально разработанный для повышения производительности и масштабируемости. Преимуществом неспециализированного хранения графов является использование зрелых неграфовых интерфейсов (например, MySQL), особенности которых хорошо знакомы разработчикам. Специализированная обработка графов (смежность без индексов) демонстрирует лучшую производительность обхода, но некоторые запросы приводят к использованию больших объемов памяти.

Взаимосвязи в графовой модели данных являются гражданами первого сорта. Здесь к ним относятся не так, как в других системах управления базами данных, где для отображения взаимосвязей применяются такие механизмы, как внешние ключи или внешние операции, например MapReduce. Собирая абстракции узлов и взаимосвязей в связанные структуры, графовая база данных позволяет строить модели любой сложности, лучше всего отражающие предметную область. Полученные модели проще и в то же время нагляднее, чем те, что создаются с помощью традиционных реляционных баз данных или других NOSQL-хранилищ.

На рис. 1.3 приведен графический обзор некоторых графовых баз данных из представленных сегодня на рынке, основанных на разных моделях хранения и обработки.

---

[1] Более подробную информацию можно найти в статье Родригеса Марко А. (Rodriguez Marko A.), Питера Неубауэр (Peter Neubauer) «The Graph Traversal Pattern». 2011 (http://arxiv.org/abs/1004.1001), в сборнике «Graph Data Management: Techniques and Applications», под ред. Шерифа Сарка (Sherif Sakr), Эрика Пардеде (Eric Pardede), 29–46. Hershey, PA: IGI Global.

Рис. 1.3 ❖ Обзор графовых баз данных

Механизмы вычисления графов

Механизмы вычисления графов позволяют выполнять глобальные графовые вычислительные алгоритмы для больших наборов данных. Они предназначены для решения таких задач, как идентификация кластеров данных или получение ответов на такие вопросы, как: «Сколько всего взаимосвязей, сколько их в среднем, полна ли социальная сеть?»

Из-за своей направленности на глобальные запросы механизмы вычисления графов, как правило, оптимизированы для сканирования и пакетной обработки больших объемов информации, и в этом отношении они похожи на другие технологии пакетного анализа, такие как интеллектуальный анализ данных (data mining) или аналитическая обработка в реальном времени (OLAP), используемые в реляционном мире. Некоторые механизмы вычисления включают в себя и средства хранения графов, а другие (большинство) заботятся только об обработке данных, получаемых из внешнего источника, а затем возвращают результаты для сохранения в другом месте.

Рисунок 1.4 иллюстрирует типовую архитектуру развертывания механизмов вычисления графов. Она включает в себя систему записи (System of Record, SOR) базы данных со свойствами OLTP (например, MySQL, Oracle или Neo4j), которая обслуживает запросы и отвечает на запросы, поступающие от приложений (и в конечном счете от пользователей). Периодически задания на извлечение, преобразование и загрузку данных (Extract, Transform, Load, ETL) перемещают

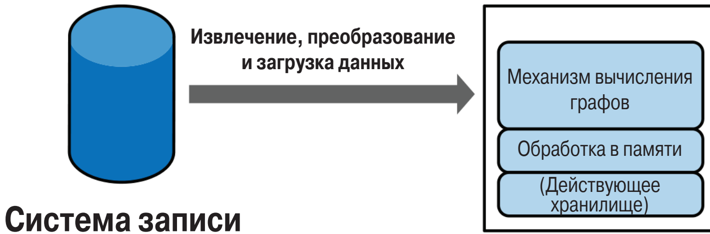


Рис. 1.4 ❖ Укрупненная схема типичной среды движков расчетов графов

данные из системы записи базы данных в механизм вычисления графов для выполнения автономных запросов и анализа.

Существуют разные типы механизмов вычисления графов. Наиболее известными из них являются *одномашинные* (*in-memory/single machin)*, такие как Cassovary (https://github.com/twitter/cassovary), и *распределенные*, такие как Pegasus (http://www.cs.cmu.edu/~pegasus/) и Giraph (http://giraph.apache.org/). В основе большинства распределенных механизмов вычисления графов лежит идея, изложенная в статье «Pregel: a system for large-scale graph processing», опубликованной на сайте Google (http://dl.acm.org/citation.cfm?id=1807184), которая описывает движок Google для классификации страниц.

Эта книга посвящена только графовым базам данных

В предыдущем разделе был приведен общий обзор областей применения графов. Остальная часть этой книги будет посвящена исключительно графовым базам данных. Далее речь пойдет о *концепции* графовых баз данных. При необходимости их рассмотрение будет иллюстрироваться примерами, взятыми из собственного опыта разработки решений с использованием графовой модели с метками и свойствами, и базы данных Neo4j. Независимо от того, какая графовая модель или база данных использована в примерах, иллюстрируемые ими идеи применимы и к другим графовым базам данных.

Преимущества графовых баз данных

Несмотря на то что практически все можно представить в виде графа, мы живем в прагматичном мире бюджетов, проектов с жесткими графиками, корпоративных стандартов и коммерческих правил. Предоставляемый графовыми базами данных новый мощный метод моделирования данных сам по себе не является достаточным основанием для замены устоявшихся и понятных платформ обработки данных – от

этого должна быть незамедлительная и весьма значительная практическая польза. Для графовых баз данных такой мотивацией может послужить применение ее в тех случаях и к таким моделям данных, когда при переходе на графовую модель будет достигнуто увеличение производительности на один и более порядков. Вместе с выигрышем в производительности графовые базы данных предоставляют чрезвычайно гибкую модель данных и способ развертывания, соответствующий современным способам развертывания программного обеспечения.

Производительность

Одной из веских причин выбора графовой базы данных является большой прирост производительности при работе со взаимосвязанными данными, по сравнению с реляционными базами данных и NOSQL-хранилищами. В отличие от реляционных баз данных, где учет взаимосвязей интенсивно ухудшает производительность запросов на больших наборах данных, производительность графовых баз данных остается неизменной с увеличением объема хранимых данных. Это связано с тем, что запросы локализуются в определенной части графа. В результате время выполнения каждого запроса зависит от размера части графа, которую требуется обойти для удовлетворения запроса, а не от общего размера графа.

Гибкость

Разработчикам и проектировщикам необходимо организовать взаимосвязи между данными, согласно требованиям области применения, структура данных должна соответствовать изменяющимся потребностям, а не навязываться заранее и оставаться неизменной. В графовых базах данных эта задача легко решается. Как мы увидим в главе 3, графовая модель данных отражает и охватывает потребности бизнеса таким образом, что может *изменяться со скоростью изменения самого бизнеса*.

Присущая графам возможность расширения означает, что можно добавлять новые виды взаимосвязей, новые узлы, новые метки и новые подграфы в существующую структуру, не нарушив при этом существующих запросов и функционала приложения. Это положительно влияет на производительность разработки и снижает риски для проекта. Благодаря гибкости графовой модели не требуется предварительно моделировать задачу в мельчайших подробностях, что очень неудобно из-за быстро меняющихся бизнес-требований. Способность графов к расширению также позволяет уменьшить ко-

личество миграций, что снижает нагрузку при обслуживании данных и уменьшает риск потери данных.

Оперативность

Модель данных должна не отставать от прочих составных частей приложения и использовать технологии, соответствующие современным итерационным методам развертывания программного обеспечения. Современные графовые базы данных оснащены всем необходимым для разработки и системного обслуживания. В частности, встроенная графовая модель данных, лишенная схем, в сочетании со встроенным программным интерфейсом (API) и языком запросов позволяет эффективно вести разработку приложений.

В то же время благодаря отсутствию схемы графовые базы данных не предполагают наличия ориентированных на схемы механизмов контроля данных, которые широко применяются в реляционном мире. Но в этом нет ничего страшного, здесь они заменены гораздо более удобными и действенными видами контроля. Как мы увидим в главе 4, контроль выполняется в программной форме, с помощью тестов для моделей данных и запросов, а также с помощью определения бизнес-правил, основанных на графе. Сейчас такая методика уже не вызывает сомнений: разработка с помощью графовых баз данных полностью соответствует современным методикам гибкой и надежной разработки программного обеспечения, что позволяет разработке приложений с использованием графовых баз данных не отставать от бизнес-среды.

Итоги

В этой главе мы рассмотрели графовую модель со свойствами, представляющую собой простой, но удобный инструмент для работы со взаимосвязанными данными. Графовая модель со свойствами хорошо моделирует области ее применения, а графовые базы данных облегчают разработку приложений, которые реализуют графовые модели.

В следующей главе мы сравним несколько различных технологий обработки взаимосвязанных данных, начнем с реляционных баз данных, затем перейдем к агрегированным NOSQL-хранилищам и закончим графовыми базами данных. Обсудив их, мы узнаем, почему графы и графовые базы данных являются лучшим средством для моделирования, хранения и выборки взаимосвязанных данных. Затем, в последующих главах, будут описаны проектирование и реализация решений, основывающихся на графовых базах данных.

Глава 2

Варианты хранения взаимосвязанных данных

Мы живем во взаимосвязанном мире. Чтобы преуспевать и развиваться, мы должны осознавать и воздействовать на окружающие нас взаимосвязи.

Как современные технологии решают проблему взаимосвязи данных? В этой главе мы рассмотрим, как реляционные базы данных и агрегированные NOSQL-хранилища работают с графами и взаимосвязанными данными, и сравним их производительность с производительностью графовых баз данных. Те читатели, которых заинтересует тема NOSQL, найдут в приложении А описание четырех основных типов NOSQL-баз данных.

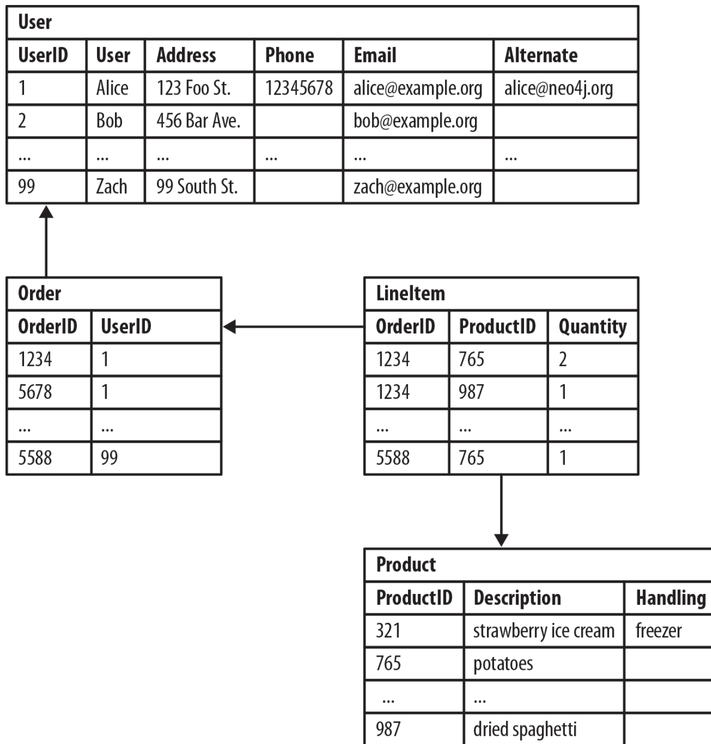
Недостатки реляционных баз данных при работе со взаимосвязями

В течение нескольких десятилетий разработчики пытались приспособить реляционные базы данных для работы со взаимосвязанными слабоструктурированными наборами данных. Но так как реляционные базы данных изначально предназначены для обработки бланков форм и таблиц, с чем они прекрасно справляются, попытки моделирования взаимосвязей реального мира оканчивались неудачей. Как это ни странно, дела со взаимосвязями у реляционных баз данных обстоят плохо.

Реляционные базы данных действительно включают механизм взаимосвязей, но применяется он только на этапе моделирования, как средство объединения таблиц. Обсуждая взаимосвязанные данные

в предыдущей главе, мы упомянули, что часто требуется устранять неоднозначность семантики взаимосвязей, связывающих объекты, а также квалифицировать их вес или силу. Реляционные базы данных не предусматривают ничего такого. И что самое плохое, при увеличении объема данных общая структура набора данных становится все более и боле сложной и все менее формализованной, реляционная модель обременяется многочисленными соединениями таблиц, разреженными записями и требует увеличения объема логики для проверки отсутствия значений. Рост связности приводит в реляционном мире к увеличению соединений, которые снижают производительность и затрудняют внесение в базу данных обновлений, связанных с изменением потребностей бизнеса.

На рис. 2.1 изображена реляционная схема хранения заказов в ориентированном на клиентов, транзакционном приложении.



User

UserID	User	Address	Phone	Email	Alternate
1	Alice	123 Foo St.	12345678	alice@example.org	alice@neo4j.org
2	Bob	456 Bar Ave.		bob@example.org	
...	...	...	...	...	...
99	Zach	99 South St.		zach@example.org	

Order

OrderID	UserID
1234	1
5678	1
...	...
5588	99

LineItem

OrderID	ProductID	Quantity
1234	765	2
1234	987	1
...	...	...
5588	765	1

Product

ProductID	Description	Handling
321	strawberry ice cream	freezer
765	potatoes	
...	...	
987	dried spaghetti	

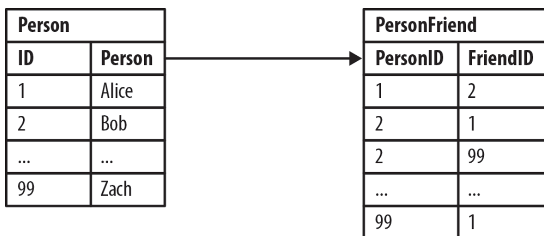
Рис. 2.1 ❖ Семантика взаимосвязей скрыта в реляционной базе данных

Область применения оказывает большое влияние на структуру этой схемы, делая часть запросов простыми, а другую часть – сложными:

- соединения таблиц добавляет сложности, они смешивают бизнес-данные с внешними ключами;
- внешние ключи сдерживают доработку базы данных и увеличивают расходы на ее обслуживание;
- разреженные таблицы с незаполненными колонками требуют дополнительного контроля в коде, несмотря на присутствие соответствующего описания в схеме;
- чтобы выяснить, что же клиент купил, потребуется достаточно *медленно выполняющееся* соединение;
- обратные запросы выполняются еще медленнее. Запрос «Какие товары этот клиент купил?» выполняется относительно быстро, по сравнению с запросом «Какие клиенты купили этот товар?», а этот запрос является базовым для системы рекомендации товаров. Можно использовать индексацию, но даже с индексацией такие обратные запросы, как «Какие клиенты купили этот и тот товар?», выполняются чрезвычайно медленно при увеличении уровня рекурсии.

Реляционные базы данных с трудом справляются со взаимосвязями. Чтобы разобраться в причинах медленного выполнения запросов для извлечения связанных данных в реляционных базах данных, рассмотрим некоторые простые и не очень запросы к социальной сети.

На рис. 2.2 приведена простое соединение таблиц для регистрации дружеских отношений.



Person

ID	Person
1	Alice
2	Bob
...	...
99	Zach

PersonFriend

PersonID	FriendID
1	2
2	1
2	99
...	...
99	1

Рис. 2.2 ❖ Моделирование связей «друзья» и «друзья друзей» в реляционной базе данных

Получить ответ на вопрос «С кем дружит Боб?» достаточно просто, текст необходимого для этого запроса показан в примере 2.1.

Пример 2.1 ❖ Друзья Боба

```
SELECT p1.Person
FROM Person p1 JOIN PersonFriend
  ON PersonFriend.FriendID = p1.ID
JOIN Person p2
  ON PersonFriend.PersonID = p2.ID
WHERE p2.Person = 'Bob'
```

Основываясь на данных нашего примера, получаем ответ: Алиса (`Alice`) и Зак (`Zach`). Запрос несложный и выполняется достаточно быстро, так как количество просматриваемых строк ограничено фильтром `WHERE p2.Person = 'Bob'`.

На дружбу не всегда отвечают взаимностью, поэтому в примере 2.2 создан обратный запрос, отвечающий на вопрос «Кто дружит с Бобом?».

Пример 2.2 ❖ Кто дружит с Бобом?

```
SELECT p1.Person
FROM Person p1 JOIN PersonFriend
  ON PersonFriend.PersonID = p1.ID
JOIN Person p2
  ON PersonFriend.FriendID = p2.ID
WHERE p2.Person = 'Bob'
```

Ответ на этот запрос: Алиса (`Alice`), к сожалению Зак (`Zach`), не считает Боба (`Bob`) своим другом. Обратный запрос по-прежнему прост в реализации, но база данных выполняет его медленнее, так как теперь необходимо просмотреть все строки таблицы `PersonFriend`.

Мы можем добавить индекс, но дорогостоящий уровень косвенности от этого не исчезнет. Проблем станет еще больше, когда мы захотим узнать «Кто друзья моих друзей?». Для учета иерархии в SQL используются рекурсивные соединения, усложняющие синтаксис и увеличивающие время выполнения запроса, что и демонстрирует пример 2.3. (Некоторые реляционные базы данных пытаются подсластить эту горькую пилюлю, например в Oracle имеется функция `CONNECT BY`, которая упрощает текст запроса, но сложность его выполнения оставляет без изменений.)

Пример 2.3 ❖ Друзья друзей Алисы

```
SELECT p1.Person AS PERSON, p2.Person AS FRIEND_OF_FRIEND
FROM PersonFriend pf1 JOIN Person p1
  ON pf1.PersonID = p1.ID
```

```
JOIN PersonFriend pf2
  ON pf2.PersonID = pf1.FriendID
JOIN Person p2
  ON pf2.FriendID = p2.ID
WHERE p1.Person = 'Alice' AND pf2.FriendID <> p1.ID
```

Этот запрос выполняется медленно, несмотря на то что предназначен только для выборки друзей Алисы и не отслеживает более дальних связей Алисы в социальной сети. Если продолжить углубляться в социальную сеть, запросы будут становиться все более сложными и медленными. Хотя еще можно за разумное время получить ответ на вопрос «Кто друзья друзей моих друзей?», но запросы, охватывающие четыре, пять или шесть уровней дружбы, будут выполняться чрезвычайно медленно из-за сложности вычислений и затрат на рекурсивное соединение таблиц.

Моделирование и извлечение взаимосвязей являются действиями, чуждыми структуре реляционной базы данных. Кроме упомянутой выше сложности и медлительности запросов, существуют и другие проблемы. Схема оказывается слишком жесткой и хрупкой. Для придания ей гибкости мы создаем разреженные таблицы со столбцами, не имеющими значений, и код для обработки исключений, и все потому, что не существует единой схемы для хранения всех необходимых данных. При этом теряется гибкость и пропадает даже подобие единства данных. Схема становится хрупкой, она рассыпается при попытках внести в нее изменения, необходимые для развития приложения.

Недостатки NOSQL-баз данных при работе со взаимосвязями

Большинство NOSQL-баз данных, основанных на парах ключ-значение, документах или семействах столбцов, хранит наборы не связанных между собой пар ключ-значение, документов или семейств столбцов. Это затрудняет их использование для хранения взаимосвязанных данных и графов.

Одним из популярных методов включения взаимосвязей в такие хранилища является добавление агрегирующих идентификаторов в поля, принадлежащие другим агрегатам, фактически являющихся внешними ключами. Но это требует агрегирования на уровне приложения, что приводит к потере производительности.

Рассматривая схему хранилища с агрегированной моделью, подобную приведенной на рис. 2.3, можно решить, что в ней отслеживаются

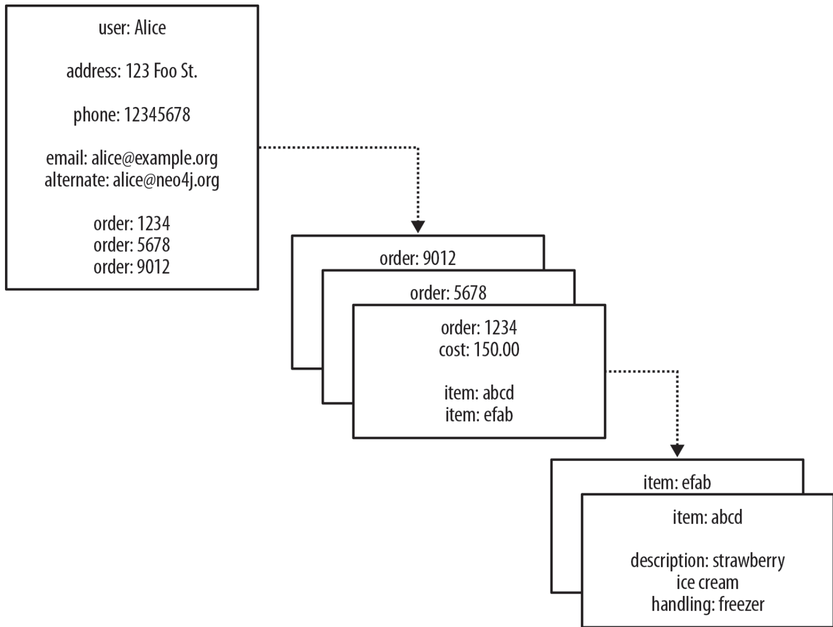


Рис. 2.3 ❖ Отражение взаимосвязей в агрегированном хранилище

взаимосвязи. Видя ссылку на заказ 1234 в записи пользователя Алиса, можно сделать вывод, что существует связь между пользователем Алисой и заказом 1234. Это дает ложную надежду, что с помощью пар ключ-значение можно моделировать графы.

По рис. 2.3 можно заключить, что некоторые значения свойств действительно являются ссылками на внешние агрегаты, расположенные в другом месте базы данных. Но такие включения в навигационную структуру не проходят бесследно, потому что взаимосвязи *между* агрегатами не являются главными в этой модели данных, большинство агрегированных хранилищ ориентировано только на *внутреннее* агрегирование в форме вложенных отображений. Таким образом, забота о поддержании взаимосвязей в этих плоских разорванных структурах данных ложится на приложение. Разработчики должны гарантировать, что приложение изменит или удалит эти внешние ссылки с изменением прочих данных. Если этого не произойдет, хранилище станет накапливать ведущие в никуда ссылки, которые несут опасность разрушения данных и отрицательно влияют на производительность запросов.

Ссылки и переходы

Хранилище пар ключ-значение Riak позволяет каждому из хранимых значений придать метаданные взаимосвязей. Все ссылки односторонние, направлены от одного хранимого значения к другому. Riak поддерживает любое количество таких *переходов* (*walked* в терминологии Riak) по взаимосвязям, что делает модель в некотором смысле взаимосвязанной. Но эта взаимосвязь поддерживается переходами с помощью скрытого отображения. В отличие от графовых баз данных, это связывание подходит для программирования только простых графовых структур, а не всех графовых алгоритмов.

В этой схеме имеется один слабый пункт. Из-за отсутствия идентификаторов, связывающих «точки» в обратном направлении (естественно, ведь внешние агрегатные «ссылки» не рефлексивны), нет возможности выполнять другие интересные запросы к базе данных. Например, для структуры, показанной на рис. 2.3, извлечение списка клиентов, купивших конкретный товар, возможно, с целью формирования рекомендаций, основанных на клиентском профиле, является очень медленной операцией. Чтобы получить ответ на подобный вопрос, скорее всего, потребуется экспортировать набор данных и обрабатывать его в какой-либо внешней вычислительной инфраструктуре, такой как Hadoop, где можно добиться результата с помощью полного перебора. Кроме того, можно ретроспективно добавить обратные внешние агрегатные ссылки, а затем выполнить запрос. В любом случае, результаты будут латентными.

Это наталкивает на мысль, что агрегатные хранилища в отношении взаимосвязанных данных функционально эквивалентны графовым базам данных. Но это не так. Агрегатные хранилища не поддерживают согласованность взаимосвязанных данных и так называемую *смежность без индексов*, которая позволяет элементам включать в себя прямые ссылки на своих соседей. Как результат для обработки взаимосвязанных данных агрегатные хранилища вынуждены использовать внешние латентные методы создания и излечения связей данных.

Рассмотрим некоторые проявления этих ограничений. На рис. 2.4 приведена структура небольшой социальной сети, реализованной с помощью агрегированного хранилища документов.

С помощью этой структуры легко выбрать непосредственных друзей пользователя, если, конечно, приложение с должным прилежанием относится к идентификаторам, хранящимся в свойстве «друзья», и они соответствуют идентификаторам записей в базе данных. В этом случае мы просто обойдем непосредственных друзей по их иденти-

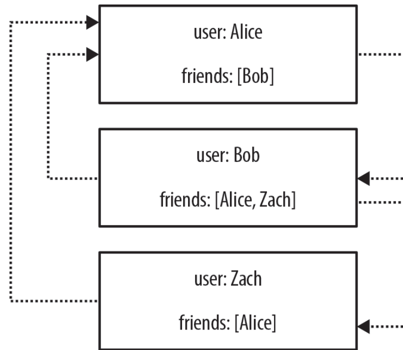


Рис. 2.4 ❖ Небольшая социальная сеть, моделируемая в агрегатном хранилище

фикаторам, что потребует выполнить несколько операций поиска по индексу (по одному для каждого друга), а не занимающего массу времени сканирования всего набора данных. Сделав это, мы определим, например, что Боб считает Алису и Зака друзьями.

Но на дружбу не всегда отвечают взаимностью. А если заменить вопрос «С кем дружит Боб?» на вопрос «Кто дружит с Бобом?». Этот вопрос окажется более сложным, чтобы ответить на него, потребуется полный перебор всего набора данных с просмотром полей «друзья», на предмет содержания в них идентификатора «Боб».

О-нотация и полный перебор

О-нотация является кратким способом выразить, как изменится производительность алгоритма при изменении размера набора данных. Алгоритм $O(1)$ имеет постоянную во времени производительность, т. е. выполнение алгоритма занимает постоянное время независимо от размера набора данных. Алгоритм $O(n)$ имеет линейную зависимость производительности, т. е. при удвоении размера набора данных удваивается и время его выполнения. Алгоритм $O(\log n)$ имеет логарифмическую зависимость производительности, т. е. когда размер набора данных удваивается, время выполнения алгоритма увеличивается на фиксированную величину. Относительное увеличение производительности может наблюдаться при начальном состоянии набора данных, но производительность быстро падает с увеличением размера набора данных. Алгоритм $O(\log n)$ является самым затратным по времени из всех, рассматриваемых в этой книге. В алгоритме $O(m \log n)$ с удвоением объема набора данных время выполнения удваивается и увеличивается на некоторую дополнительную величину, пропорциональную числу элементов в наборе данных.

Полный перебор всех данных представляется в терминах затрат времени как $O(n)$, потому что просматриваются все n агрегатов хранилища данных. Он становится слишком затратным при переборе большинства наборов

данных вполне разумных размеров, поэтому предпочтение обычно отдается алгоритму *O*(log *n*), потому что он отбрасывает половину или больше потенциальной нагрузки в каждой итерации.
Графовые базы данных обеспечивают постоянную производительность для таких же запросов. Здесь необходимо найти узел в графе, представляющий Боба, а затем отследить все входящие *дружеские* взаимосвязи; эти взаимосвязи приведут к узлам, которые представляют людей, считающих Боба своим другом. При этом затрат будет гораздо меньше, чем при полном переборе, потому что просматривается намного меньше членов социальной сети, т. е. просматриваются только члены сети, связанные с Бобом. Конечно, если все члены сети дружат с Бобом, необходимо будет перебрать весь набор данных.

Чтобы избежать необходимости обрабатывать весь набор данных, можно денормализовать модель хранения, добавив обратные ссылки. Добавление второго свойства, которое можно назвать, например «друг по», каждому пользователю, позволит перечислить входящие дружеские взаимосвязи, связанные с пользователем. Но ничто не дается даром. Для начала мы должны понести первоначальные и текущие затраты, связанные с увеличением времени записи, плюс увеличение размеров дискового пространства для хранения дополнительных метаданных. Кроме того, *переход* (*traversing*) по ссылкам останется затратным, потому что каждый переход потребует просмотра индекса. Это связано с тем, что агрегатные хранилища не имеют ни малейшего представления о местоположении, в отличие от графовых баз данных, которые обеспечивают смежность без индексов через реальные взаимосвязи. Реализуя графовую структуру в не предназначенном для этого хранилище, мы получаем некоторые преимущества частичной связности, но при существенных затратах.

Затраты значительно увеличиваются с увеличением числа переходов. Поиск друзей выполняется просто, но представьте, что требуется получить в реальном времени друзей друзей или друзей друзей друзей. Реализовать это в базе данных такого типа сложно, потому что переход по сфабрикованным взаимосвязям является весьма затратным процессом. Это не только ограничивает возможность расширения социальной сети, но и делает невозможным предоставление нужных рекомендаций, ведет к пропуску ошибок в центре обработки данных и позволяет вести мошенническую торговую деятельность через сеть. Многие системы стараются поддерживать обработку, внешне похожую на графовую, но она неизбежно становится пакетной, потому что не может обеспечить нужную скорость для взаимодействия в реальном времени, необходимого пользователям.

Взаимосвязи в графовых базах данных

Предыдущие примеры имели дело с *неявно* связанными данными. Как пользователи мы отслеживали семантические зависимости между объектами, но модели данных и сами базы данных не видели этих связей. Чтобы компенсировать этот недостаток, приложениям требуется создать сеть изолированных данных, а затем смириться с медленным выполнением запросов и латентно выполнять запись в денормализированных хранилищах.

В идеале необходимо добиться получения общей связанной картины, содержащей в том числе связи между элементами. В отличие от уже просмотренных хранилищ, в мире графов взаимосвязанные данные *хранятся в виде взаимосвязанных данных*. Там, где существуют связи в прикладной области, существуют и связи в данных. Рассмотрим пример социальной сети, приведенный на рис. 2.5.

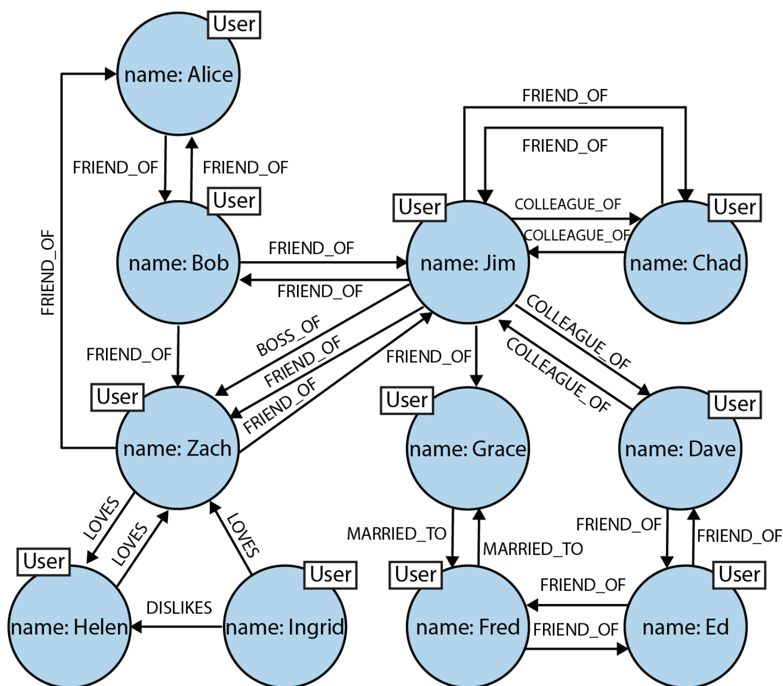


Рис. 2.5 ❖ Простота моделирования дружеских, служебных, рабочих и (невзаимных) любовных связей с помощью графа

В этой социальной сети, как и во многих других реальных примерах связанных данных, связи между объектами не проявляют единообразия, прикладная область является переменно-структурированной. Социальная сеть является популярным примером тесно связанной, переменно-структурированной сети, которая плохо моделируется с помощью жестких схем или схем, разделенных на изолированные агрегаты. Наша простая сеть друзей разрослась (потенциальных друзей могут соединять связи, содержащие до шести сегментов) и обогатилась новыми членами. Гибкость графовой модели позволила добавить новые *узлы* и новые виды *взаимосвязей* без ущерба для существующей сети или миграции данных, оригинальные объекты и их отношения остались нетронутыми.

Граф предоставляет гораздо более полную картину сети. Можно отследить, кто кого любит (и кто кому отвечает тем же). Можно увидеть, кто является кому коллегой и кто их начальник. Можно увидеть, кто занят, потому что женат или замужем; можно также проследить антипатии в нашей социальной сети с помощью взаимосвязей «не любит». Имея в своем распоряжении такой граф, рассмотрим преимущества графовых баз данных при работе со взаимосвязанными данными.

Метки в графе

Обычно узлы в сетях классифицируют согласно ролям, которые они играют. Некоторые узлы, например, могут представлять пользователей, а другие – заказы или товары. В Neo4j метки используются для определения ролей узлов в графе. Поскольку узел может исполнять несколько ролей, Neo4j позволяет узлу иметь несколько меток.

Таким образом, с помощью меток можно группировать узлы. Можно, например, обойти в базе данных все узлы с метками «Пользователь». (Метки также обеспечивают ловушки для декларативной индексации узлов, подробнее мы рассмотрим это позже.) Далее метки часто будут использоваться в примерах. Если узел представляет пользователя, ему присваивается метка `User`, если узел представляет заказ, ему присваивается метка `Order` и т. д. Синтаксис будет рассмотрен в следующей главе.

Взаимосвязи в графе естественным образом образуют маршруты. Выполнение запросов, или перемещение, связано со следованием по маршрутам. Из-за своей, изначально ориентированной на маршруты модели данных большинство базирующихся на маршрутах операций графовой базы данных выполняется с очень высокой скоростью, что делает их чрезвычайно эффективными. В своей книге «*Neo4j in Action*» (http://www.manning.com/partner/) Партнер (Partner) и Вукотич (Vukotic) приводят полученные экспериментальным путем

сравнительные данные о быстродействии реляционных хранилищ и Neo4j. Сравнение показывает, что графовая база данных (в данном случае Neo4j и фреймворк Traversal) существенно быстрее при работе со связанными данными, чем реляционное хранилище.

Эксперимент Партнера и Вукотича основывался на поиске друзей друзей в социальной сети на глубине до пяти уровней. Результаты для социальной сети, объединяющей 1 000 000 человек, каждый из которых имеет около 50 друзей, убедительно свидетельствуют, что графовые базы данных являются лучшим выбором для связанных данных, как это демонстрирует табл. 2.1.

Таблица 2.1. Сравнение эффективности поиска друзей в реляционной базе данных и в Neo4j

Глубина	Время выполнения в реляционной базе данных (в секундах)	Время выполнения в Neo4j (в секундах)	Количество возвращенных записей
2	0,016	0,01	~2500
3	30,267	0,168	~110 000
4	1543,505	1,359	~600 000
5	Не завершено	2,132	~800 000

На глубине двух уровней (друзья друзей) обе базы данных показали достаточно хорошее время, чтобы использовать их в онлайн-системе. Хотя запрос в Neo4j занимает лишь две трети времени выполнения аналогичного запроса в реляционной базе данных, конечный пользователь не заметит разницу в миллисекундах. Но с увеличением глубины запросов до трех уровней (друзья друзей друзей) становится ясно, что реляционная база данных перестает справляться с работой за разумные сроки: ожидание в 30 секунд уже совершенно неприемлемо для онлайн-системы. Время отклика Neo4j возрастает не намного, точнее на доли секунды, и время выполнения запросов, безусловно, остается приемлемым.

На глубине четырех уровней реляционная база данных показывает настолько огромное время выполнения, что делает невозможным применение такого запроса в онлайн-системе. Neo4j также показывает ухудшение, но задержка все еще остается приемлемой для интерактивной онлайн-системы. И наконец, на глубине пяти уровней выполнение запроса в реляционной базе данных занимает так много времени, чтобы просто невозможно дождаться завершения запроса. А Neo4j, напротив, возвращает результат за время, близкое к двум секундам. Как свидетельствует полученный на глубине пяти уровней

результат, практически вся сеть состоит из наших друзей. Поэтому в реальных применениях, скорее всего, потребуется отфильтровать результат, тем самым снизив время выполнения.

И агрегированные хранилища, и реляционные базы данных показывают плохую производительность, когда выходят за пределы узкого круга операций, для которых они предназначены. Запросы для получения сведений о маршрутах в графе, например друзья друзей, выполняются очень медленно. Мы не утверждаем, что графовые базы данных во всем превосходят агрегированные хранилища или реляционные базы данных. Просто они хорошо справляются с определенными задачами, но не отвечают необходимым требованиям при работе со связанными данными. Все прочее, кроме обхода непосредственных друзей или, возможно, выборки друзей друзей, будет выполняться медленно из-за количества поисков по индексам. Графовые базы данных, напротив, используют смежность без индексов, чтобы обход связанных данных выполнялся очень быстро.

Пример с социальной сетью помог проиллюстрировать, как разные технологии работают со взаимосвязанными данными, но является ли этот пример реальной задачей? Действительно ли потребуется выборка столь удаленных «друзей»? Возможно, и нет. Но замените социальную сеть любой другой областью применения, и вы увидите, что подобные преимущества производительности, моделирования и технического обслуживания будут востребованными. Будь то музыка или центр управления данными, биоинформация или футбольная статистика, сетевые датчики или серии торгов, графы обеспечивают эффективное моделирование этих данных. Давайте рассмотрим другое современное применение графов: система рекомендации товаров на основе истории покупок пользователя и его друзей, соседей и прочих похожих на него людей. В следующем примере мы соберем вместе несколько независимых аспектов жизни пользователя, чтобы подготовить точные и приносящие прибыль рекомендации.

Начнем с моделирования истории покупок пользователя как связанных данных. В графе очень просто соединить пользователя с его заказами и связать вместе заказы для получения истории покупок, как показано на рис. 2.6.

В графе на рис. 2.6 главное внимание уделено действиям клиентов. На нем видны все заказы пользователя и содержимое каждого заказа. Для структуры данных этой прикладной области добавлена поддержка нескольких популярных шаблонов доступа. Например, пользователи часто хотят увидеть свою историю заказов, поэтому мы

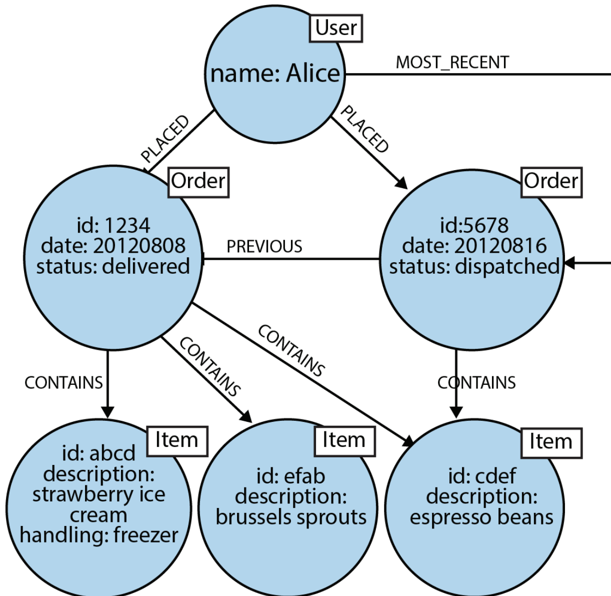


Рис. 2.6 ❖ Моделирование истории заказов пользователя с помощью графа

добавили в граф связи, позволяющие найти самые последние заказы пользователя, следуя по исходящей связи. Мы можем перебрать весь список, перемещаясь назад во времени. Если потребуется перемещаться во времени в прямом направлении, можно следовать по связи «предыдущий» в обратном направлении или добавить обратные связи «следующий».

Теперь начнем создавать рекомендации. Если мы замечаем, что многие пользователи, купившие клубничное мороженое, также покупают бобы эспрессо, мы можем рекомендовать эти бобы тем пользователям, которые обычно покупают только мороженое. Но это слишком примитивный подход к рекомендациям, его можно *значительно* улучшить. Для увеличения мощности графа присоединим его к графам из других областей. Так как графы – это многомерные структуры, легко можно создать более сложные запросы к данным, чтобы получить доступ к хорошо отлаженному сегменту рынка. Например, можно запросить у графовой базы данных «все виды мороженого, которые предпочитают люди, отдающие предпочтение эспрессо, но не любящие брюссельскую капусту, проживающие в конкретной местности».

В нашем случае определить популярность конкретного товара можно по частоте его покупки клиентами. Но как ответить на вопрос «живет в окрестности»? Геоинформационные координаты очень хорошо моделируются графами. Одним из самых популярных инструментов для предоставления геоинформационных координат является R-дерево. R-дерево (https://ru.wikipedia.org/wiki/R-дерево) – это подобный графу указатель, который описывает ограниченные географические области. Используя такую структуру, можно описать перекрывающиеся иерархии местоположений. Например, можно отразить тот факт, что Лондон расположен в Великобритании, а почтовый индекс SW11 1BD соответствует Баттерси – району Лондона, города в юго-восточной части Англии, находящейся, в свою очередь, в Великобритании. Следовательно, воспользовавшись почтовыми индексами Великобритании, можно ограничить местность, где живут люди с интересующими нас вкусами[1].

Такие запросы, включающие сопоставление с шаблоном, очень сложно выразить на SQL, и их создание требует кропотливого и вдумчивого отношения, но независимо от стараний выполняются они, как правило, очень медленно. Графовые базы данных, напротив, оптимизированы именно для таких типов перемещений и запросов с сопоставлением с шаблоном, обеспечивая во многих случаях выполнение за миллисекунды. Кроме того, большинство графовых баз данных предоставляет язык запросов, предназначенный для выражения графовой структуры, и запросы к графам. В следующей главе мы рассмотрим язык Cypher, который является языком, специально предназначенным для описания графов с помощью схем.

Наш пример графа можно использовать для выработки рекомендаций пользователям, но он способен принести пользу и продавцам. Например, учитывая определенную закономерность покупок (набор товаров, обычная их стоимость и т. д.), можно определить, является ли конкретная сделка подозрительной. В графе легко обнаружить выход покупки за пределы обычной для пользователя нормы, а затем привлечь к ней внимание (с использованием приемов определения сходства, хорошо известных из литературы о сборе данных с помощью графов), тем самым уменьшив риск мошенничества[2].

---

[1] Геопространственная библиотека Neo4j предоставляет *n*-мерные полигоны индексов. Ее можно найти по адресу https://github.com/neo4j-contrib/spatial.

[2] Более подробную информацию об определении сходства можно найти в статье Клейна Д. Дж. (Klein D. J.) «Centrality measure in graphs» // Journal of Mathematical Chemistry. May 2010. 47(4): 1209-1223.

Ясно, что, с практической точки зрения, графовая база данных является лучшей технологией для работы с комплексными, переменно-структурированными, тесно связанными наборами данных, т. е. с такими наборами данных, которые не поддаются обработке любыми другими средствами, кроме графов.

Итоги

В этой главе мы узнали, что взаимосвязи в реляционных базах данных и NOSQL-хранилищах требуют обработки на уровне приложений и, напротив, в графовых базах данных им придается первостепенное значение. В следующей главе мы подробно рассмотрим тему моделирования графов.

Глава 3

Моделирование данных графами

Предыдущие главы были посвящены описанию преимуществ графовых баз данных, и сравнению их с NOSQL-хранилищами и традиционными реляционными базами данных. Но после того как был сделан выбор в пользу графовых баз данных, возникает вопрос: как же моделировать посредством графов?

Эта глава посвящена моделированию графами. Для начала повторим графовую модель с метками и свойствами, самую популярную из графовых моделей, затем кратко остановимся на языке графовых запросов Cypher, используемом в большинстве примеров в книге. Существует несколько языков графовых запросов, но наиболее широко из них распространен язык Cypher, что и делает его де-факто стандартом. Он наиболее прост и понятен, особенно тем, кто уже знаком с языком запросов SQL. Затем, опираясь на полученные знания, приступим к рассмотрению нескольких примеров моделирования графами. На примере управления центром обработки данных будет произведено сравнение реляционного и графового методов моделирования. В другом примере, связанном с творчеством Шекспира, будет использовано соединение графов и запросов данных нескольких прикладных областей. Заканчивается глава списком распространенных ошибок при моделировании графами и описанием наиболее удачных приемов такого моделирования.

Модели и задачи

Прежде чем углубиться в моделирование графами, скажем несколько слов в целом о моделях. Моделирование – это процесс абстрагирования некой области деятельности для решения поставленной задачи. При моделировании происходит отображение некоторых сущностей неуправляемой области на пространство, где они могут быть струк-

турированы и где ими можно оперировать. При этом теряется естественность реального мира, он перестает быть тем, чем является «на самом деле», происходят отбор, абстракция и упрощение тех его черт, которые наиболее важны для достижения определенной цели.

Представление в виде графа не является исключением. Единственным его отличием от прочих методов моделирования данных является близкое сходство логической и физической моделей. Реляционные методы управления данными требуют значительного отхода от естественного представления прикладной области: сначала прикладная область трансформируется в логическую модель, а затем преобразуется в физическую. Эти преобразования приводят к семантическому диссонансу между концепциями реального мира и их реализациями в такой базе данных. В графовых базах данных этот разрыв значительно сокращается.

Мы уже общаемся посредством графов

Моделирование графами выглядит достаточно естественным – для абстрактного представления предметов и понятий используются кружки и квадратики, а затем их соединяют стрелками и линиями, отображающими связи между ними. Современные графовые базы данных в большей мере, чем прочие технологии баз данных, являются тем, что можно изобразить на лекционной доске. Типичное изображение проблемы на доске выглядит как граф. Так что эскизы проектных и аналитических моделей близки к моделям данных, реализуемых в базах данных.

С точки зрения наглядности, графовые базы данных сводят к минимуму потери соответствия между анализом и реализацией, которые в течение многих лет преследуют реляционное моделирование. Самое интересное в графовых моделях, что они не только связаны так, как мы это себе представляем, но и обмениваются теми же вопросами, на которые мы хотим получить ответы.

Как мы увидим в этой главе, графовые модели и графовые запросы действительно являются двумя сторонами одной медали.

Графовая модель со свойствами и метками

Графовая модель со свойствами и метками была описана в главе 1. Напомним ее основные особенности:

- *графовая модель со свойствами и метками* состоит из *узлов*, *взаимосвязей*, *свойств* и *меток*;
- узлы обладают свойствами. Их можно рассматривать как документы, которые хранят свойства в виде произвольных пар ключ-значение. В Neo4j ключи являются строками, а значе-

ния – Java-строками или другими примитивными типами данных, а также массивами этих типов;
- узлы могут быть помечены одной или несколькими метками. Метки группируют узлы и указывают роли, которые они играют в наборе данных;
- взаимосвязи соединяют узлы и образуют структуру графа. Взаимосвязи всегда имеют направление, единственное имя, *начальный узел* и *конечный узел* – не бывает незавершенных взаимосвязей. Наличие направления и имени у взаимосвязей добавляют семантической четкости структурированию узлов;
- как и узлы, взаимосвязи также могут обладать свойствами. Возможность добавления свойств взаимосвязям полезна, в частности, для предоставления графовым алгоритмам дополнительных метаданных, внося дополнительную семантику во взаимосвязи (в том числе качество и вес), а также для ограничений запросов.

Для создания сложных и семантически богатых моделей достаточно этих простых примитивов. До сих пор все модели изображались в виде схем. Схемы прекрасно подходят для представления графов, не связанных с технологическим контекстом, но когда дело доходит до использования их в базе данных, необходим некоторый механизм для создания, манипулирования и извлечения данных. Нам необходим язык запросов.

Графовые запросы: введение в Cypher

Cypher – это выразительный (но компактный) язык запросов графовых баз данных. В настоящее время Cypher используется лишь в Neo4j, но представление графов в виде привычных схем делает его идеальным средством программного описания графов. По этой причине мы используем язык Cypher во всех примерах книги для демонстрации графовых запросов и графовых структур. Cypher является, пожалуй, самым простым графовым языком запросов и послужит отличной основой для изучения графов. Освоение Cypher существенно облегчит вам последующее изучение других графовых языков запросов.

Следующие несколько разделов будут посвящены краткому обзору языка запросов Cypher. Это не справочник по Cypher, а всего лишь дружелюбное введение, более сложные сценарии графовых запросов мы рассмотрим позже[1].

---

[1] Электронную документацию Cypher можно найти по адресам http://goo.gl/W7Jh6x и http://goo.gl/ftv8Gx.

Другие языки запросов

В других графовых базах данных имеются свои средства извлечения данных. Многие из них, в том числе и Neo4j, поддерживают язык запросов SPARQL для RDF (http://www.w3.org/TR/rdf-sparql-query/) и императивный язык запросов Gremlin (https://github.com/tinkerpop/gremlin/wiki/), базирующийся на маршрутах. Но нас интересует представление свойств графов с помощью языка запросов, поэтому мы практически везде будем использовать Cypher.

Философия языка Cypher

Язык Cypher читабелен и понятен разработчикам, специалистам по базам данных и заказчикам. Простота его использования основывается на сходстве способа описания графов с интуитивным их представлением в виде схем.

Cypher предоставляет пользователю (или приложению, действующему от имени пользователя) возможность задавать шаблон для поиска данных. Проще говоря, можно попросить базу данных «найти что-то похожее на это». А для описания того, «как это должно выглядеть», мы будем использовать ASCII-графику. Рассмотрим простую модель, изображенную рис. 3.1.

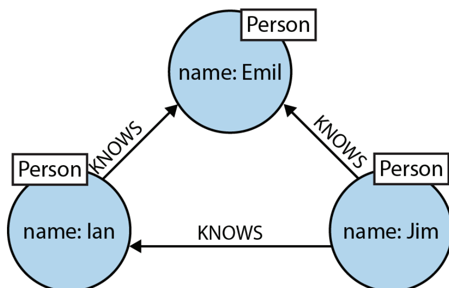


Рис. 3.1 ❖ Простая модель графа, изображенная схемой

Приведенный шаблон описывает трех друзей. А вот его эквивалент на Cypher, представленный ASCII-графикой:

```
(emil)<-[:KNOWS]-(jim)-[:KNOWS]->(ian)-[:KNOWS]->(emil)
```

Этот шаблон описывает *маршрут*, соединяющий узел с именем `jim` с двумя другими узлами `ian` и `emil`, которые также связаны соединением, идущим от узла `ian` к узлу `emil`. `ian`, `jim` и `emil` являются *идентификаторами*. Идентификаторы позволяют ссылаться на узлы

в описании модели и обойти тот факт, что язык запросов имеет только одно измерение (его текст записывается слева направо), в то время как схема графа использует два измерения. Несмотря на то что иногда возникает необходимость в повторении идентификаторов, смысл записи остается ясным. Запись шаблонов на Cypher выглядит очень естественной, близкой к изображению графов на лекционной доске.

Предыдущий Cypher-шаблон описывает структуру простого графа, но не определяет ссылок на конкретные данные в базе данных. Для связывания шаблона с конкретными узлами и взаимосвязями в существующем наборе данных необходимо указать конкретные значения свойств и метки узлов, которые позволят найти соответствующие элементы в наборе данных. Например:

```
(emil:Person {name:'Emil'})
  <-[:KNOWS]-(jim:Person {name:'Jim'})
  -[:KNOWS]->(ian:Person {name:'Ian'})
  -[:KNOWS]->(emil)
```

Здесь мы связали узлы с их идентификаторами, используя свойство `name` и метку `Person`. Идентификатор `emil`, например, связан с узлом в наборе данных с меткой `Person` и со свойством `name`, имеющим значение `Emil`. Прикрепление части шаблона к реальным данным в этом случае является обычной практикой для языка Cypher, в чем мы еще не раз убедимся в следующих разделах.

Спецификация на примере

Самым интересным в схемах графов является присутствие конкретных экземпляров узлов и взаимосвязей, а не классов или архетипов. Даже очень большие графы можно изображать с помощью меньших подграфов, состоящих из реальных узлов и взаимосвязей. Другими словами, для описания графов обычно используется *спецификация на примере*.

Шаблоны графов, представленные ASCII-графикой, являются основой Cypher. Запросы на Cypher прикрепляют одну или несколько частей шаблона к определенным местам графа с помощью предикатов, а затем перемещают незафиксированные части, пытаясь найти соответствие.

Точки фиксации в реальном графе, с которым связаны некоторые части шаблона, определяются в Cypher метками и предикатами свойств, включенными в запрос. В большинстве случаев Cypher использует метаинформацию о существующих индексах, ограничениях и предикатах для автоматического их определения. Но иногда ему требуются дополнительные подсказки.

Как и большинство языков запросов, Cypher состоит из фраз. Простейшие запросы состоят из фразы `MATCH` и следующей за ней фра-

зы RETURN (ниже в этой главе будет приведено описание и других фраз, используемых в запросах на Cypher). Следующий пример запроса на Cypher использует эти фразы для поиска прямых друзей пользователя Jim:

```
MATCH (a:Person {name:'Jim'})-[:KNOWS]->(b)-[:KNOWS]->(c), (a)-[:KNOWS]->(c)
RETURN b, c
```

Рассмотрим подробнее каждую из фраз.

MATCH

Фраза MATCH является основой большинства Cypher-запросов. Это часть *спецификации на примере*. Используя ASCII-символы для представления узлов и взаимосвязей, мы *прорисовываем* данные, которые нам нужны. Мы прорисовываем в скобках узлы и вычерчиваем взаимосвязи с помощью пар тире с символами больше или меньше (--> и <--). Символы < и > указывают направление взаимосвязей. Между тире располагается имя взаимосвязи, заключенное в квадратные скобки и предваряемое двоеточием. Метки узлов также предваряются двоеточием. Пары ключ-значение для свойств узла (и взаимосвязи) указываются в фигурных скобках (подобно объектам в JavaScript).

В нашем примере запроса мы ищем узел с меткой Person и значением Jim в свойстве name. Результату этого поиска присваивается идентификатор a. Этот идентификатор позволяет в остальной части запроса ссылаться на узел, представляющий пользователя Jim.

Этот начальный узел является частью простого шаблона: (a)-[:KNOWS]->(b)-[:KNOWS]->(c), (a)-[:KNOWS]->(c), который описывает маршрут, включающий в себя три узла, один из которых мы связали с идентификатором a, два других – с b и c. Эти узлы соединены между собой несколькими взаимосвязями KNOWS, как показано на рис. 3.1.

Этот шаблон теоретически может встретиться в графе несколько раз; если пользователей достаточно много, с этим шаблоном может быть сопоставлено множество взаимосвязей. Для локализации запроса необходимо закрепить некоторые его части в одном или нескольких местах графа. Указав, что поиск ведется от узла с меткой Person и значением Jim в свойстве name, мы прикрепляем шаблон к конкретному узлу в графе – узлу, представляющему пользователя Jim. Затем Cypher сопоставляет остальную часть шаблона с областью графа, непосредственно окружающей точку привязки. Если они существуют, найденным узлам присваиваются другие идентификаторы. Хотя идентификатор a всегда будет привязан к узлу Jim, идентификаторы

b и c в процессе выполнения запроса будут связаны с последовательностью узлов.

Описание точки прикрепления шаблона можно также оформить в виде предиката во фразе WHERE.

```
MATCH (a:Person)-[:KNOWS]->(b)-[:KNOWS]->(c), (a)-[:KNOWS]->(c)
WHERE a.name = 'Jim'
RETURN b, c
```

Здесь искомое свойство мы переместили из фразы MATCH во фразу WHERE. Результат этого запроса будет тем же.

RETURN

Эта фраза определяет, какие узлы, взаимосвязи и свойства в совпавших данных должны быть возвращены клиенту. В нашем примере запроса необходимо вернуть узлы, связанные с идентификаторами b и c. При обходе полученного результата будут получены все совпавшие узлы, связанные с идентификаторами.

Другие фразы языка Cypher

В запросах на Cypher можно использовать другие фразы:

WHERE
: Определяет критерии совпадения результатов для фильтрации шаблона.

CREATE и CREATE UNIQUE
: Создает узлы и взаимосвязи.

MERGE
: Гарантирует, что заданный шаблон будет существовать в графе либо за счет использования уже найденных в графе узлов и взаимосвязей, соответствующих заданным предикатам, либо за счет создания новых узлов и взаимосвязей.

DELETE
: Удаляет узлы, взаимосвязи и свойства.

SET
: Устанавливает значения свойств.

FOREACH
: Вносит изменения в каждый из элементов списка.

UNION
: Объединяет результаты двух или более запросов.

`WITH`

Объединяет части запроса в цепочку и передает результаты от одной части запроса к другой. Работает подобно именованным каналам в Unix.

`START`

Явно указывает одну или несколько отправных точек, узлов или взаимосвязей в графе. (Фраза `START` признана устаревшей, и вместо нее рекомендуется явно указывать отправные точки во фразе `MATCH`.)

Если эти фразы покажутся вам знакомыми, особенно если вы работали с SQL, отлично! Интуитивное понимание фраз языка Cypher поможет вам быстрее освоить его. В то же время следует подчеркнуть их отличие, потому что здесь мы имеем дело с графами, а *не* реляционными наборами данных.

Ниже в этой главе будут приведены примеры применения этих фраз. Мы более подробно будем описывать особенности их работы по мере встречи с ними.

Теперь, когда мы знаем, как определить граф и как извлечь данные из графа с помощью Cypher, можно перейти к рассмотрению примеров моделирования.

Сравнение реляционного и графового моделирования

Введение в моделирование графами начнем со сравнения приемов моделирования прикладной области методами, основанными на реляционной и графовой технологиях. Большинству разработчиков и специалистов по базам данных знакомы реляционные СУБД (системы управления базами данных) и связанные с ними методы моделирования данных. Поэтому имеет смысл сравнить методы графового моделирования с ними и выявить несколько общих особенностей и массу отличий. В частности, мы узнаем, как прост переход от концептуальной модели графа к физической и насколько мало графовая модель вносит искажений в представление, по сравнению с реляционной моделью.

Для облегчения сравнения рассмотрим простую схему управления центрами обработки данных. В этой прикладной области несколько центров обработки данных поддерживает множество приложений от имени многих клиентов с помощью различных составляющих инфра-

структуры, от виртуальных машин до распределителей физической нагрузки. Пример этой прикладной области приведен на рис. 3.2.

На рис. 3.2 мы видим упрощенное представление нескольких приложений и инфраструктуры центров обработки данных, необходимых для их поддержки. Приложения, представленные узлами `App 1`, `App 2` и `App 3`, зависят от серверов кластера баз данных, помеченных как `Database Server 1, 2, 3`. Пользователи логически зависят от наличия приложений и их данных, кроме того, между пользователями и приложениями имеется дополнительная физическая инфраструктура. Эта инфраструктура включает в себя виртуальные машины (`Virtual Machine 10, 11, 20, 30, 31`), реальные серверы (`Server 1, 2, 3`), стойки для серверов (`Rack 1, 2`) и распределители загрузки (`Load Balancer 1, 2`), которые являются фасадом приложений. Компоненты соединены массой сетевых элементов: кабелями, коммутаторами, коммутационными панелями, контроллерами сетевых интерфейсов (Network Interface Controllers, NIC), источниками питания, кондиционерами и прочими устройствами, которые могут выйти из строя в самый неподходящий момент. Для полноты картины имеется и единственный пользователь приложения 3, помеченный как `User 3`.

Перед нами как операторами системы стоят две основные задачи:

- текущее предоставление функциональности для удовлетворения (или превышения) условий договора об уровне обслуживания, включая возможность выполнять прогнозные анализы для определения потенциальных точек отказа и ретроспективные анализы для быстрого определения причин жалоб клиентов о недоступности услуги;
- учет потребляемых ресурсов, в том числе стоимость оборудования, виртуализации, сетевых услуг, а также затрат на разработку программного обеспечения (так как они являются просто логическим расширением рассматриваемой здесь системы).

При проектировании системы управления центром обработки данных необходимо гарантировать, что базовая модель данных позволит хранить данные и выполнять к ним запросы так, чтобы эффективно решать эти основные задачи. Кроме того, нужно иметь возможность обновлять базовую модель при изменении портфеля приложений, физического расширения центра обработки данных и миграции экземпляров виртуальных машин. Учитывая эти потребности и условия, сравним решения, основанные на реляционной и графовой моделях.

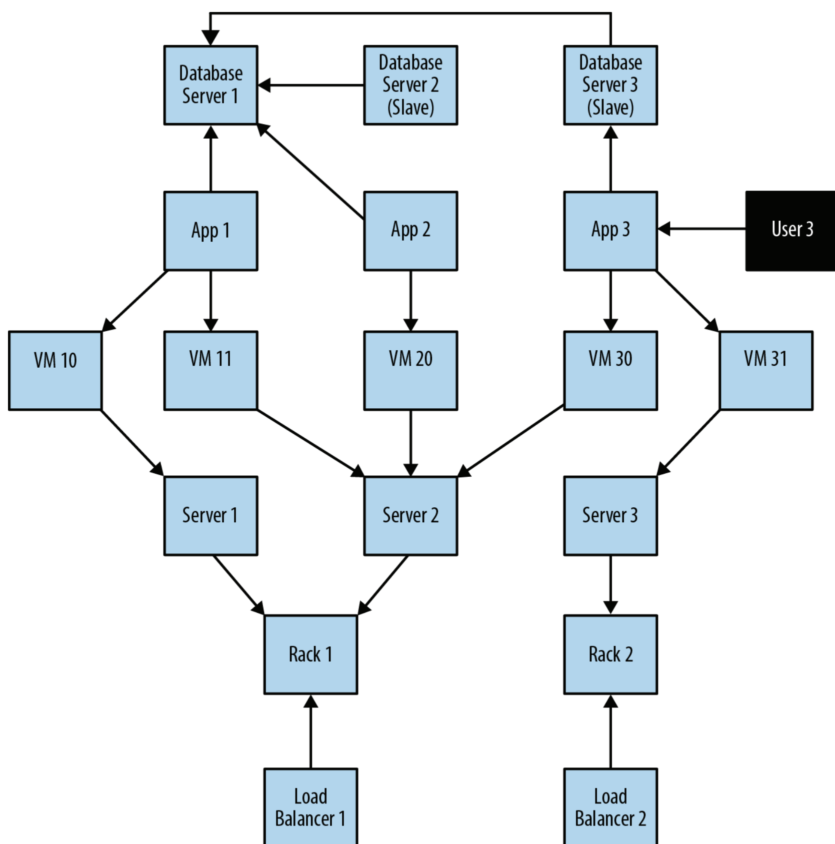


Рис. 3.2 ❖ Упрощенная схема развертывания приложений в центре обработки данных

Реляционная модель системы управления

Начальный этап моделирования в реляционном мире похож на начальный этап моделирования другими методами, т. е. он заключается в осмыслении концепций и достижении соглашений, касающихся объектов выбранной области, о том, как они взаимодействуют, и правил, регулирующих изменение их состояний. Большая часть работ на этом этапе обычно выполняется неофициально, часто используются эскизы на лекционных досках и их обсуждение экспертами в прикладной области, специалистами по архитектуре данных. Для выра-

жения общего понимания задачи и достигнутых соглашений создается схема, например такая, как на рис. 3.2, имеющая форму графа.

На следующем этапе соглашения фиксируются в более строгой форме, такой как схема сущностей-взаимосвязей (entity-relationship, сокращенно E-R), также в форме графа. При этом происходит преобразование концептуальной модели в логическую, использующую более строгую нотацию, что дает второй шанс для уточнения словаря прикладной области, чтобы его можно было использовать совместно со специалистами по реляционным базам данных. (Такой подход требуется не всегда: ревностные приверженцы реляционных моделей часто сразу переходят к разработке таблиц и нормализации без предварительного создания промежуточной E-R-схемы). В нашем примере мы отобразим прикладную область на E-R-схеме, как показано на рис. 3.3.

Будучи графами, E-R-схемы непосредственно демонстрируют недостатки реляционной модели при работе со сложной прикладной областью. Хотя в них и предусмотрены именованные взаимосвязи (это то, что поддерживают графовые базы данных, а некоторые реляционные хранилища – нет), E-R-схемы позволяют только единичные, неориентированные, именованные взаимосвязи между сущностями. В этом отношении реляционная модель плохо подходит для отображения реальных прикладных областей, где взаимосвязи между субъектами многочисленны, семантически отличаются друг от друга и разнообразны.

После получения логической модели выполняются ее отображение в таблицы и взаимосвязи с последующей нормализацией для устранения избыточности. Во многих ситуациях этот шаг чрезвычайно прост и состоит из преобразования E-R-схемы в табличную форму и загрузки полученных таблиц с помощью SQL-команд в базу данных. Но даже такой простейший случай иллюстрирует особенности реляционной модели. Например, по рис. 3.4 видно, что модель излишне усложнена из-за ограничений внешних ключей (помеченных `[FK]`), поддерживающих связи один ко многим, и соединений таблиц (например, `AppDatabase`), поддерживающих связи многие ко многим, и это уже тогда, когда еще не было добавлено ни одной строки реальных пользовательских данных. Эти ограничения заложены на уровне модели метаданных, которые нужны для определения конкретных взаимосвязей между таблицами при выполнении запросов. Однако присутствие таких структурных данных весьма ощутимо, потому что они загромождают данные прикладной области и смешивают их с данными, обслуживающими базу данных, а не пользователей.

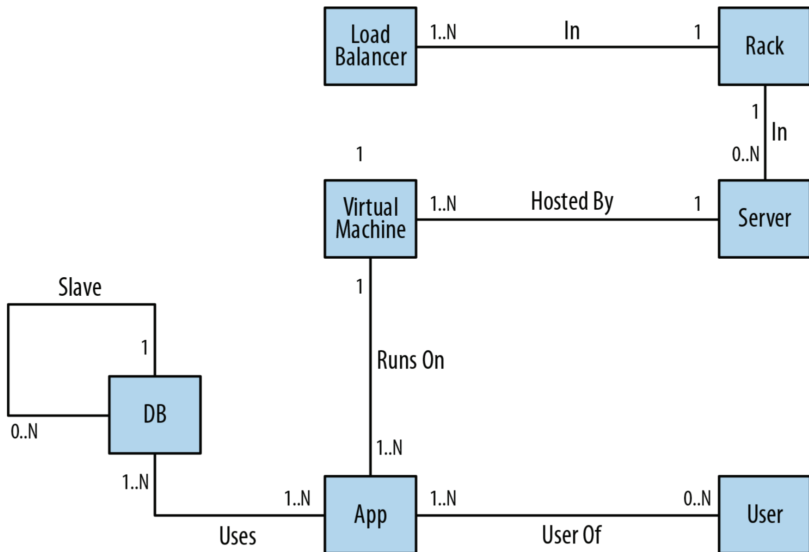


Рис. 3.3 ❖ E-R-схема для прикладной области центра обработки данных

Теперь у нас есть нормализованная модель, которая относительно точно отображает прикладную область. Эта модель, хотя и существенно усложнена внешними ключами и соединениями таблиц, не содержит дубликатов данных. Но проектирование еще не завершено. Одна из проблем реляционной парадигмы – в том, что нормализованные модели, как правило, недостаточно быстры для реального применения. Во многих промышленных системах нормализованная схема, которая в теории отвечает всем требованиям, предъявляемым к точности отображения прикладной области, на практике нуждается в дополнительной адаптации и специализации для соответствия конкретным моделям доступа. Другими словами, чтобы достичь нужного обеспечения реляционными хранилищами обычных потребностей приложений, необходимо отказаться от соответствия с прикладной областью и признать, что следует изменить пользовательскую модель данных *для удовлетворения нужд движка базы данных, а не пользователя*. Такая технология называется *денормализацией*.

Денормализация, кроме всего прочего, подразумевает дублирование данных (в некоторых случаях весьма существенное) для увеличения производительности запросов. В качестве примера рассмотрим

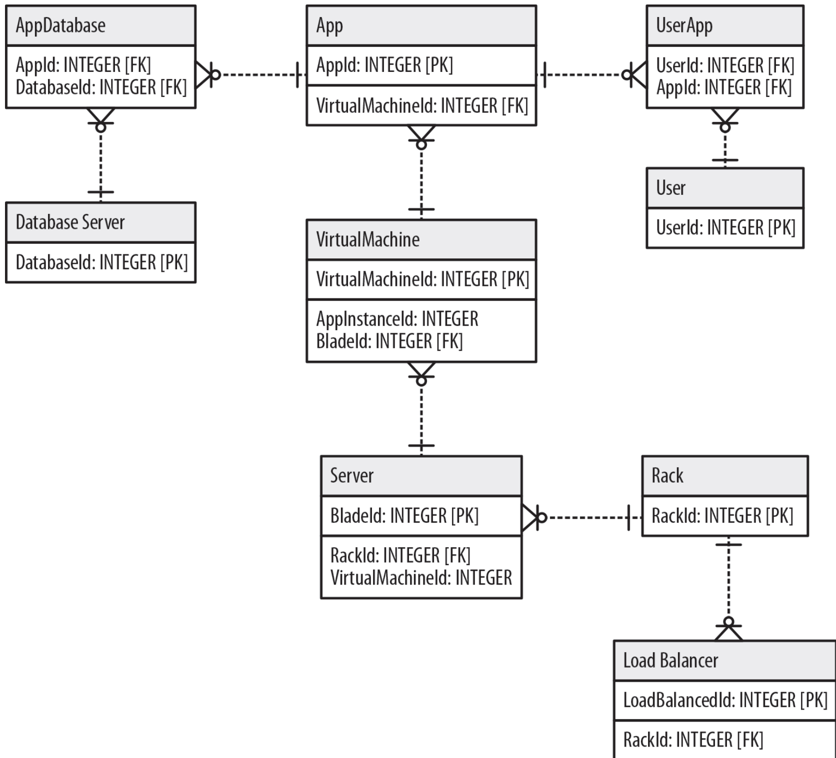


Рис. 3.4 ❖ Таблицы и взаимосвязи для прикладной области центра обработки данных

сведения о пользователях и их контактные данные. Типичный пользователь часто имеет несколько адресов электронной почты, которые в полноценной нормализованной модели следовало бы поместить в отдельную таблицу `EMAIL`. Но, чтобы уменьшить снижение производительности, вызванное необходимостью соединения двух таблиц, довольно часто эти данные встраиваются в таблицу `USER`, при этом в нее добавляется один или несколько столбцов для хранения наиболее важных адресов электронной почты.

Хотя денормализация не таит никакой опасности (предполагается, что разработчики знакомы с денормализацией моделей, знают, как она влияет на прикладной код, и имеют надежную транзакционную поддержку базы данных), она не является тривиальной задачей. Для достижения наилучших ее результатов обращаются к настоящим экс-

пертам по реляционным СУБД для искажения нормализованной модели и преобразования ее в денормализованную, в наибольшей мере соответствующую характеристикам СУБД и особенностям хранения на физическом уровне. При этом необходимо смириться с возникновением существенной избыточности данных.

Можно, конечно, считать, что наличие сложностей при проектировании-нормализации-денормализации вполне приемлемо, потому что все это делается лишь единожды. Такой ход мыслей предполагает, что стоимость затрат на разработку амортизируется в течение всего времени существования системы (включающего в себя разработку и эксплуатацию), так что дополнительные затраты на разработку реляционной модели сравнительно невелики, по сравнению с общей стоимостью проекта. Хорошо, если это действительно так, но во многих случаях такое предположение не соответствует действительности, так как в систему вносятся изменения не только в течение разработки, но и в течение всего времени ее эксплуатации.

Такая точка зрения на амортизацию модели данных, когда дорогостоящее внесение изменений при разработке перекрывается долгосрочными преимуществами устойчивой эксплуатации модели, предполагает, что система будет работать в неизменных эксплуатационных условиях и что среда эксплуатации будет оставаться стабильной. Это, конечно, возможно, но большинство систем функционирует в эксплуатационных средах, которые редко бывают стабильными. Изменение требований бизнеса или смена нормативных условий приводит к необходимости вносить соответствующие корректировки в системы и структуры данных, на которых они построены.

Модели данных всегда подвергаются существенной переработке на этапах проекта, связанных с проектированием и разработкой, и почти всегда эти изменения предназначены для приспособления модели к потребностям приложений, которые будут использовать ее при эксплуатации. Влияние проектирования на модель значительно до такой степени, что становится практически невозможным изменить приложение и модель во время эксплуатации так, чтобы она смогла начать поддерживать те требования, поддержка которых не была изначально предусмотрена.

Технический механизм, служащий для внесения структурных изменений в базу данных, называется *миграцией*, он широко используется во фреймворках для разработки приложений, таких как Rails (http://guides.rubyonrails.org/migrations.html). Миграции обеспечивают системный поэтапный подход к рефакторингу баз данных (http://

databaserefactoring.com/), состоящий в применении ряда операций к базе данных, преобразующих ее для удовлетворения меняющихся потребностей использующих ее приложений. Но, в отличие от рефакторинга кода, который обычно занимает считанные секунды или минуты, рефакторинг баз данных может занять несколько недель или месяцев, блокируя работу на все время изменения схемы. Рефакторинг баз данных выполняется медленно, несет в себе риски и является дорогим процессом.

Необходимость денормализации модели затрудняет ее быстрое обновление, связанное с изменением бизнес-требований. Мы убедились на примере центра обработки данных, что изменения, внесенные в модель на лекционной доске, создают пропасть между концептуальным и физическим размещением данных; такой концептуально-реляционный диссонанс затрудняет активное сотрудничество сторон, заинтересованных в дальнейшей эволюции системы. Взаимодействие заинтересованных сторон остается за порогом реляционного здания. Трудности при реализации изменений бизнес-требований в базовой, укоренившейся реляционной структуре вызывают отставание ее развития от развития бизнеса. Без участия высококлассных специалистов и жесткого планирования миграция денормализованных баз данных таит в себе ряд опасностей. Если миграция не приведет к получению поддерживаемой хранилищем структуры, может пострадать производительность. Если намеренно продублированные данные после миграции будут изолированы, возникает серьезная опасность глобального нарушения целостности данных.

Графовое моделирование системы управления

Мы убедились, что реляционные моделирование и сопутствующие ему мероприятия по реализации толкают нас на путь отделения базовой модели хранения данных приложения от концептуальной модели заказчиков. Реляционные базы данных, имеющие жесткие схемы и сложные определения моделей, не станут хорошим инструментом при необходимости быстрого внесения изменений. Нам нужна модель, которая будет тесно связана с прикладной областью, но не в ущерб производительности, и которая будет поддерживать развитие с сохранением целостности данных, так как ее потребуется оперативно корректировать и расширять. Такой моделью является графовая модель. Чем же отличается процесс реализации графового моделирования данных?

Ранние стадии анализа похожи на те, которые выполнялись при реляционном подходе: те же действия по созданию эскизов на лек-

ционных досках для общего описания и согласования. Но затем методики начинают отличаться. Вместо трансформирования графовой модели прикладной области в табличное представление выполняется ее обогащение с целью создания точного представления элементов прикладной области, имеющих отношение к задачам приложения. То есть каждая сущность в прикладной области должна быть охвачена соответствующими ролями в виде меток, ее атрибуты отражены свойствами, а связи с другими сущностями – выражены взаимосвязями.

Не забывайте, что модель прикладной области является не прозрачным, контекстно-независимым окном в реальность, а скорее целенаправленной абстракцией тех аспектов прикладной области, которые имеют отношение к целям ее применения. Что-то всегда служит причиной создания модели. При обогащении начального графа прикладной области дополнительными свойствами и взаимосвязями создается эффективная графовая модель, удовлетворяющая потребности в данных приложения, т. е. предусматривающая возможность получения ответов на различные запросы, данные приложением.

Моделирование прикладной области по форме совпадает с графовым моделированием. Приводя в соответствие модель прикладной области, мы неявно совершенствуем и графовую модель, потому что при использовании графовых баз данных *эскиз, изображенный на лекционной доске, и является обычно тем, что хранится в базе данных.*

Каждый узел графа должен иметь метки и свойства, соответствующие его роли, чтобы он мог выполнять свои обязанности, определенные прикладной областью. Также для каждого узла нужно обеспечить правильный семантический контекст, т. е. направленные именованные взаимосвязи (часто обладающие атрибутами), соединяющие его с другими узлами для охвата структурных аспектов прикладной области. Графовая модель управления центром обработки данных приведена на рис. 3.5.

Логически это все, что нам нужно сделать. Не нужно ни нормализовать таблицы, ни денормализовать их. Полученное точное представление модели прикладной области без труда можно применить к базе данных, в чем вы убедитесь в ближайшее время.

Обратите внимание, что большинство узлов здесь имеет две метки: метку определения типа (например, база данных `Database`, приложение `App` или сервер `Server`) и метку общего назначения `Asset` (ресурс). Это позволяет извлекать в некоторых запросах только отдельные типы ресурсов, а в других запросах – все ресурсы независимо от их типа.

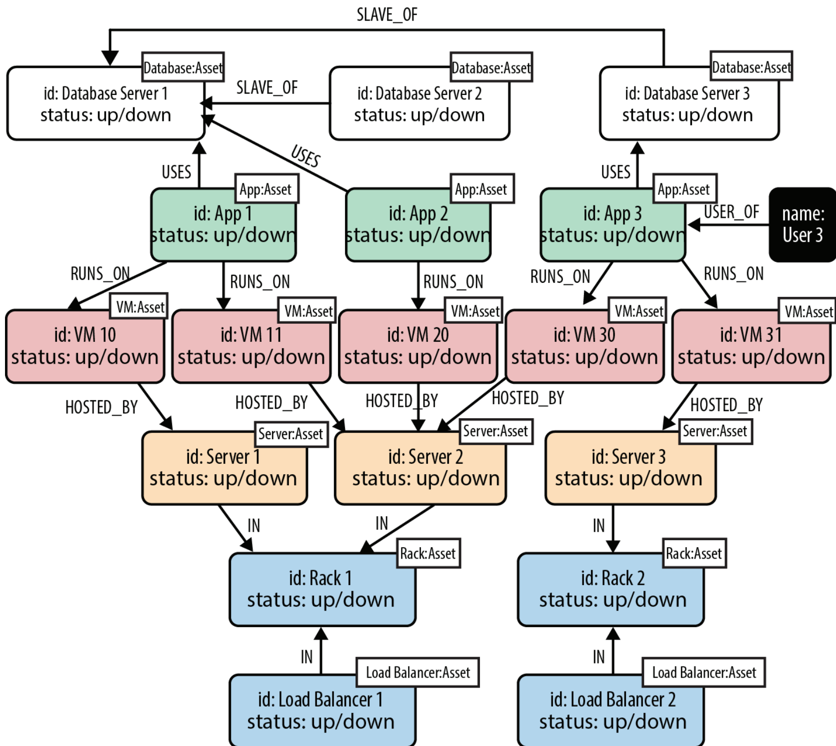


Рис. 3.5 ❖ Пример графа для реализации сценария управления центром обработки данных

Тестирование модели

После детализации модели прикладной области следующим шагом нужно проверить, обеспечивает ли она получение ответов на требуемые запросы. Хотя графы отлично поддерживают постоянно меняющиеся структуры (и, следовательно, позволяют исправление любых ошибок в проектных решениях), но существует несколько проектных решений, которые, будучи зафиксированы в приложении, станут препятствовать дальнейшим изменениям. Проанализировав модель прикладной области и полученную графовую модель на ранней стадии, можно избежать таких ловушек. Последующие изменения в структуре графа должны определяться исключительно изменениями в бизнесе, а не необходимостью исправления неудачных проектных решений.

Для этого существуют две часто применяемые на практике технологии. Первая и самая простая заключается в проверке читаемости графа. Выбирается начальный узел, а затем выполняется обход его взаимосвязей с другими узлами, при этом считываются метки каждого из узлов и имена каждой пройденной взаимосвязи. В процессе обхода должны получаться осмысленные предложения. Для примера центра обработки данных можно прочесть такие предложения, как «Экземпляры приложений App 1, 2 и 3 используют базы данных на серверах Database Servers 1, 2 и 3» и «На сервере Server 3 работает виртуальная машина VM 31, где выполняется экземпляр приложения App 3». Если чтение графа приводит ко вполне осмысленным результатам, можно с уверенностью сказать, что прикладная область отображена правильно.

Далее следует рассмотреть запросы, которые будут выполняться на графе, точнее оценить *доступность запросов*. Чтобы проверить, поддерживает ли граф ожидаемые виды запросов, необходимо сформулировать эти запросы. Для этого потребуется понимание целей конечных пользователей, т. е. прецеденты применения графа. В случае с центром обработки данных, например, одним из прецедентов использования являются сообщения от конечных пользователей, что приложение или служба не отвечает. Чтобы помочь этим пользователям, нужно определить причину сбоя, а затем устранить ее. Чтобы выявить причину, нужно определить, что находится на пути между пользователем и приложением, а также зависимости, необходимые приложению для работы. Чтобы окончательно удостовериться, что граф отвечает потребностям прикладной области, следует проверить, можно ли создать Cypher-запрос, выявляющий причину прецедента.

Продолжая рассмотрение прецедента из примера, предположим, что имеется возможность обновлять граф с помощью инструментов непрерывного мониторинга сети, обеспечивая тем самым информацию о состоянии сети в реальном времени. В большой физической сети можно было бы использовать обработку сложных событий (Complex Event Processing, CEP. https://ru.wikipedia.org/wiki/Обработка_сложных_событий) для обработки потоков сетевых низкоуровневых событий, обновляя граф, только когда данные CEP превысят порог значимости для прикладной области. Когда пользователь сообщает о проблеме, следует ограничиться рассмотрением физических проблем сетевых элементов, находящихся между пользователем и приложением и между приложением и его зависимостями. В нашем

графе найти неисправное оборудование можно с помощью следующего запроса:

```
MATCH (user:User)-[*1..5]-(asset:Asset)
WHERE user.name = 'User 3' AND asset.status = 'down'
RETURN DISTINCT asset
```

Здесь фраза MATCH описывает *маршрут переменной длины* от одной до пяти взаимосвязей. Для взаимосвязей не заданы имя и направление (отсутствуют двоеточие и имя взаимосвязей в квадратных скобках и подсказка в виде стрелки для указания направления). Ей соответствуют приведенные ниже маршруты:

```
(user)-[:USER_OF]->(app)
(user)-[:USER_OF]->(app)-[:USES]->(database)
(user)-[:USER_OF]->(app)-[:USES]->(database)-[:SLAVE_OF]->(another-database) (user)-[:USER_
OF]->(app)-[:RUNS_ON]->(vm)
(user)-[:USER_OF]->(app)-[:RUNS_ON]->(vm)-[:HOSTED_BY]->(server)
(user)-[:USER_OF]->(app)-[:RUNS_ON]->(vm)-[:HOSTED_BY]->(server)-[:IN]->(rack)
(user)-[:USER_OF]->(app)-[:RUNS_ON]->(vm)-[:HOSTED_BY]->(server)-[:IN]->(rack)
  <-[:IN]-(load-balancer)
```

То есть здесь фразе MATCH соответствуют все ресурсы, находящиеся от пользователя на расстоянии от 1 до 5 взаимосвязей, без учета направлений взаимосвязей. Для получения нужного результата добавлено условие, что узлы ресурсов должны содержать свойство состояния status со значением down. Если узел не имеет свойства status, он не включается в результат. Фраза RETURN DISTINCT asset гарантирует, что все ресурсы в возвращаемом результате будут уникальными, независимо от того, сколько раз они соответствовали фразе MATCH.

Учитывая, что такой запрос легко поддерживается графом, можно быть уверенным, что проект пригоден для выбранных целей.

Кросс-модели нескольких прикладных областей

Понимание прикладной области часто зависит от понимания скрытых переплетающихся эффектов в сложной цепи оценок. Для создания такого понимания необходимо объединить прикладные области без искажений или потерь особенностей, присущих каждой прикладной области. Особенности графов и здесь обеспечивают решение. С помощью свойств графов можно смоделировать цепь оценок в виде

графа графов, в котором определенные взаимосвязи связывают, сохраняя их различия, прикладные подобласти.

На рис. 3.6 приведено графовое представление цепи оценок востребованности шекспировских произведений. Здесь собрана инфор-

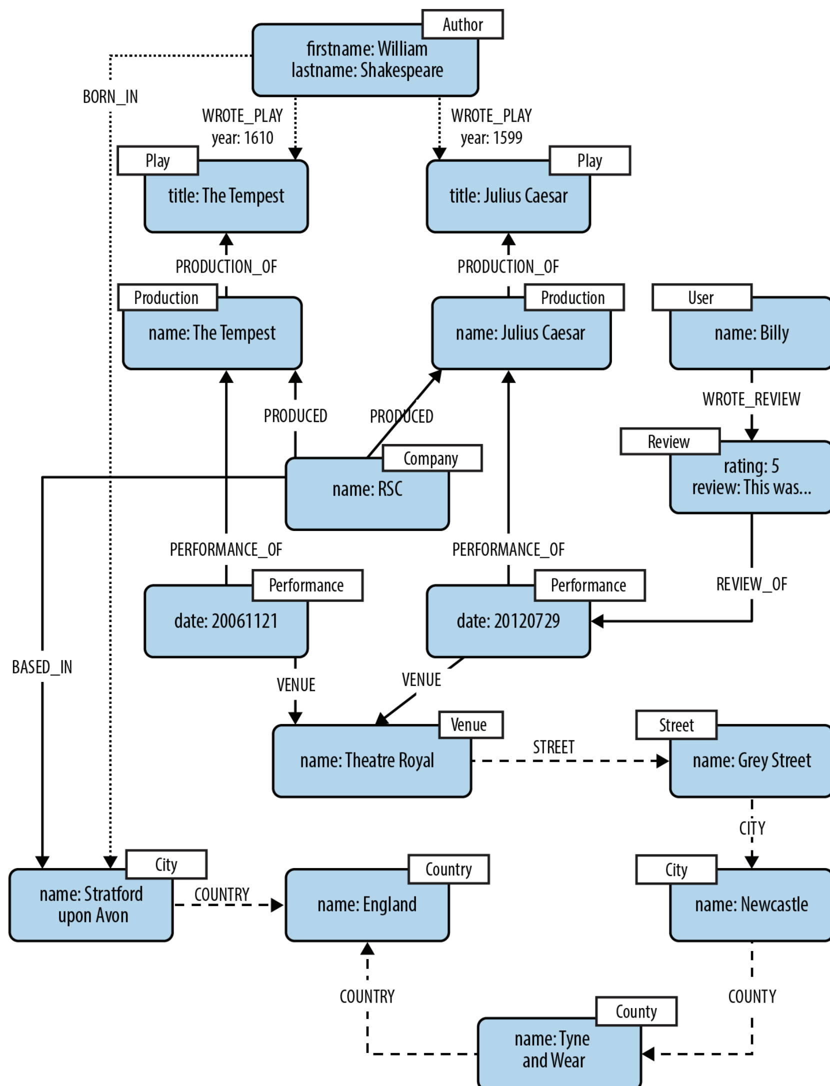


Рис. 3.6 ❖ Три прикладные области в одном графе

мация о Шекспире и некоторых его пьесах вместе со сведениями об одной из компаний, которая в настоящее время организует постановку пьес, плюс месторасположение театров и некоторые геоинформационные данные. Сюда также добавлены отзывы. В целом граф описывает и соединяет три различные прикладные области. На схеме эти три области различаются с помощью по-разному отформатированных взаимосвязей: линии из точек – для области литературы, сплошные – для театральной области и пунктирные – для геоинформационной области.

В литературной области имеется узел, представляющий самого Шекспира с меткой `Author` (автор) и свойствами `firstname:'William'` (имя: Уильям) и `lastname:'Shakespeare'` (фамилия: Шекспир). Этот узел соединен с парой узлов, каждый из которых помечен как `Play` (пьеса), представляющих пьесы «Юлий Цезарь» (*Julius Caesar*, `title:'Julius Caesar'`) и «Буря» (*The Tempest*, `title:'The Tempest'`), с помощью взаимосвязей `WROTE_PLAY`.

Выполняя обход этого подграфа слева направо, в направлении стрелок взаимосвязей, можно узнать, что Уильям Шекспир написал пьесы «Юлий Цезарь» и «Буря». На случай, если понадобится узнать даты их написания, взаимосвязи `WROTE_PLAY` снабжены свойством `date`, указывающим, что пьеса «Юлий Цезарь» написана в 1599 году, а «Буря» – в 1610 году. Несложно понять, что в граф можно включить любые другие произведения Шекспира, пьесы и стихи, просто добавив несколько узлов, представляющих произведения, и соединив их с узлом, представляющим Шекспира, взаимосвязями `WROTE_PLAY` и `WROTE_POEM`.

Водя пальцем по стрелкам взаимосвязей `WROTE_PLAY`, мы выполняем ту же работу, что и графовая база данных, хотя скорость человека существенно отличается от скорости компьютера. Как мы убедимся позже, эта простая операция *обхода* (*traversal*) является строительным блоком для сколь угодно сложных графовых запросов.

Далее в театральной области добавлена информация о Королевской Шекспировской компании (Royal Shakespeare Company, более известной по аббревиатуре RSC) в виде узла с меткой `Company`, ключевым свойством `name` и значением `RSC`. Театральная область связана с литературной, что неудивительно. В нашем графе нашел отражение тот факт, что RSC поставила (`PRODUCED`) пьесы «Юлий Цезарь» (`Julius Caesar`) и «Буря» (`The Tempest`). В свою очередь, эти театральные постановки связаны с пьесами в литературной области посредством взаимосвязей `PRODUCTION_OF` (организатор).

Также граф включает в себя сведения о конкретных спектаклях. Например, спектакль, поставленный компанией RSC по пьесе «Юлий Цезарь» (`Julius Caesar`), прошел 29 июля 2012 года в рамках сезона летних гастролей, организованных RSC. Для получения места проведения спектакля нужно пройти по взаимосвязи `VENUE` (место) и узнать, что спектакль прошел в Королевском театре (`Theatre Royal`), который представлен узлом с меткой `Venue`.

Граф позволяет также отражать обзоры конкретных спектаклей. В нашем примере в граф включен только один отзыв пользователя Билли (`Billy`) о спектакле 29 июля 2012 года. Его отображение обеспечивает взаимодействие узлов спектакля, рейтинга и пользователя. Здесь узел с меткой `User`, представляющий пользователя Билли (свойство `name:'Billy'`), с помощью исходящей взаимосвязи `WROTE_REVIEW` связан с узлом отзыва пользователя. Узел отзыва `Review` содержит числовое свойство рейтинга `rating` и свойство `review` с отзывом в виде текста в свободном формате. Отзыв связан с определенным спектаклем посредством исходящей взаимосвязи `REVIEW_OF` (автор отзыва). При масштабировании графа, т. е. включении в него множества пользователей, отзывов и спектаклей, достаточно просто добавить больше узлов с соответствующими метками и больше одинаково поименованных взаимосвязей.

Третья область, касающаяся геоинформационных данных, – это простое иерархическое дерево местоположений. Геоинформационная область связана с двумя другими областями в нескольких точках. Узел города Стратфорд-на-Эйвоне с меткой `City` (и свойством `name:'Stratford upon Avon'`) связан с литературной областью и отражает место рождения Шекспира (Шекспир был рожден (`BORN_IN`, в Стратфорде). Узел связан и с театральной областью, поскольку в нем расположена компания RSC (RSC базируется (`BASED_IN`) в Стратфорде). Чтобы больше узнать о географическом положении города Стратфорд-на-Эйвоне, можно проследить его исходящую взаимосвязь `COUNTRY`, чтобы обнаружить, что он находится в стране (`Country`) с названием Англия (`England`).

Обратите внимание, как граф избегает дублированных данных в прикладных областях. Узел `Stratford upon Avon`, например, используется во всех трех областях.

Граф позволяет отображать и более сложные географические данные. Из меток узлов, с которыми связан Королевский театр (Theatre Royal), к примеру, можно узнать, что он находится на улице `Grey`

Street в городе (City) Ньюкасл (Newcastle), что в графстве (County) Тайн-энд-Уир (Tyne and Wear), которое в конечном итоге находится в стране (Country) Англии (England), так же как Стратфорд-на-Эйвоне (Stratford upon Avon).

Взаимосвязи и метки

Имена взаимосвязей и метки узлов используются для структурирования графа и определения семантического контекста каждого узла.
Имена и направление взаимосвязей помогают отразить семантический контекст с помощью соединения двух узлов. Следуя по исходящей взаимосвязи WROTE_REVIEW, например, можно узнать, что узел на конце взаимосвязи является отзывом.
Взаимосвязи одновременно помогают разбить граф на отдельные области и связать их вместе. Как видно из примера, касающегося творчества Шекспира, графовая модель со свойствами позволяет легко объединить различные прикладные области, каждая из которых содержит свои собственными сущности, метки, свойства и взаимосвязи, не сливая их все вместе, а создавая общее представление с помощью связей между ними.
Метки отражают роли, которые узел играет в прикладной области. Поскольку узел может быть связан со многими прочими узлами, причем некоторые из них могут принадлежать к разным прикладным областям, узел, в принципе, может играть несколько ролей.
Метки играют важную роль в графовой модели со свойствами. Они не только указывают роли узлов в прикладной области, но и позволяют связать метаданные с узлами, к которым они прикреплены. Мы можем, например, перечислить все узлы с метками User или потребовать, чтобы все узлы с меткой Customer имели уникальное значение свойств адреса электронной почты.

Создание графа творчества Шекспира

Чтобы создать граф творчества Шекспира, приведенный на рис. 3.6, построим общую структуру с помощью фразы CREATE. Эта фраза выполняется внутри единственной транзакции, т. е. после ее успешного выполнения можно быть уверенным, что граф полностью создан в базе данных. Если транзакция не увенчается успехом, никакая часть графа не будет сохранена в базе данных. Как и следовало ожидать, Cypher имеет удобный визуальный способ создания графов:

```
CREATE (shakespeare:Author {firstname:'William', lastname:'Shakespeare'}),
       (juliusCaesar:Play {title:'Julius Caesar'}),
       (shakespeare)-[:WROTE_PLAY {year:1599}]->(juliusCaesar),
       (theTempest:Play {title:'The Tempest'}),
       (shakespeare)-[:WROTE_PLAY {year:1610}]->(theTempest),
       (rsc:Company {name:'RSC'}),
       (production1:Production {name:'Julius Caesar'}),
       (rsc)-[:PRODUCED]->(production1),
```

```
(production1)-[:PRODUCTION_OF]->(juliusCaesar),
(performance1:Performance {date:20120729}),
(performance1)-[:PERFORMANCE_OF]->(production1),
(production2:Production {name:'The Tempest'}),
(rsc)-[:PRODUCED]->(production2),
(production2)-[:PRODUCTION_OF]->(theTempest),
(performance2:Performance {date:20061121}),
(performance2)-[:PERFORMANCE_OF]->(production2),
(performance3:Performance {date:20120730}),
(performance3)-[:PERFORMANCE_OF]->(production1),
(billy:User {name:'Billy'}),
(review:Review {rating:5, review:'This was awesome!'}),
(billy)-[:WROTE_REVIEW]->(review),
(review)-[:RATED]->(performance1),
(theatreRoyal:Venue {name:'Theatre Royal'}),
(performance1)-[:VENUE]->(theatreRoyal),
(performance2)-[:VENUE]->(theatreRoyal),
(performance3)-[:VENUE]->(theatreRoyal),
(greyStreet:Street {name:'Grey Street'}),
(theatreRoyal)-[:STREET]->(greyStreet),
(newcastle:City {name:'Newcastle'}),
(greyStreet)-[:CITY]->(newcastle),
(tyneAndWear:County {name:'Tyne and Wear'}),
(newcastle)-[:COUNTY]->(tyneAndWear),
(england:Country {name:'England'}),
(tyneAndWear)-[:COUNTRY]->(england),
(stratford:City {name:'Stratford upon Avon'}),
(stratford)-[:COUNTRY]->(england),
(rsc)-[:BASED_IN]->(stratford),
(shakespeare)-[:BORN_IN]->stratford
```

Предыдущий код выполняет два действия: создает узлы с метками (и их свойства), а затем соединяет их взаимосвязями (и создает свойства взаимосвязей, где это необходимо). Например, фраза `CREATE (shakespeare:Author {first name:'William', lastname:'Shakespeare'})` создаст узел `Author`, представляющий Уильяма Шекспира. Вновь созданному узлу назначается идентификатор `shakespeare`. Идентификатор `shakespeare` позже используется для прикрепления взаимосвязей к основному узлу. Например, фраза`(shakespeare)-[:WROTE_PLAY {year:1599}]->(juliusCaesar)` создает взаимосвязь `WROTE` (автор) *от* узла, представляющего Шекспира, к узлу, представляющему пьесу «Юлий Цезарь». Эта взаимосвязь имеет свойство `year` со значением `1599`.

Идентификаторы доступны только внутри текущего запроса. Чтобы имена узлов оставались доступными вне запроса, нужно создать индекс для комбинации метки и ключевого свойства. Мы рассмотрим индексы в разделе «Индексы и ограничения» ниже.

В отличие от реляционной модели, приведенные выше команды не вносят избыточных усложнений в граф. Информационные метаданные модели, т. е. объединение узлов посредством меток и взаимосвязей, хранятся отдельно от бизнес-данных, которые отображаются исключительно с помощью свойств. Больше нет необходимости беспокоиться о внешних ключах и ограничении захламления реальных данных, так как в графовой модели прикладная область отображается явно, с помощью узлов и соединяющих их богатых семантически взаимосвязей.

Внести изменения в уже существующий граф можно двумя способами. Чтобы расширить граф, можно использовать все тот же оператор `CREATE`. Но, кроме того, можно использовать оператор `MERGE`, применение которого имеет смысл, когда *требуется*, чтобы в графе присутствовал определенный подграф узлов и взаимосвязей, часть которого, возможно, уже существует, а другая часть может отсутствовать на момент выполнения команды. Обычно оператор `CREATE` применяется для добавления с возможностью дублирования, а оператор `MERGE` – когда дублирование должно быть полностью исключено.

Введение в запросы

Теперь у нас есть граф и можно приступать к созданию запросов. В языке Cypher запросы всегда начинаются с одной или нескольких заранее известных отправных точек, которые называются *обязательными* узлами (bound nodes) графа. Cypher использует любые предикаты меток и свойств, поддерживаемые фразами `MATCH` и `WHERE`, совместно с метаданными, поддерживаемыми индексами и ограничениями, для поиска отправных точек графовых шаблонов.

Например, если требуется получить сведения о спектаклях в Королевском театре, запрос начинается с задания узла, представляющего в графе Королевский театр, который определяется с помощью метки `Venue` и соответствующего значения свойства `name`. Если требуется извлечь отзывы определенного пользователя, в качестве отправной точки запроса используется узел этого пользователя, определяемый меткой `User` и соответствующим значением свойства `name`.

Предположим, что нужно извлечь сведения обо всех событиях, связанных с Шекспиром, которые имели место в Королевском теат-

ре в Ньюкасле. Тогда отправными точками для запроса будут: имя автора (Author) Шекспир (Shakespeare), место проведения (Venue) с названием Королевский театр (Theatre Royal) и город (City) с названием Ньюкасл (Newcastle).

```
MATCH (theater:Venue {name:'Theatre Royal'}),
      (newcastle:City {name:'Newcastle'}),
      (bard:Author {lastname:'Shakespeare'})
```

Фраза MATCH выявит все узлы мест проведения (Venue) со значением ключевого свойства name, равным Theatre Royal, и свяжет их с идентификатором theater. (Что делать, если в графе существует несколько узлов Theatre Royal? Мы разберемся с этим чуть ниже.) На следующем шаге определяется узел, представляющий город (City) Ньюкасл, и связывается с идентификатором newcastle. И наконец, так же как в более раннем запросе по Шекспиру, при поиске узла, представляющего самого Шекспира, ищется узел с меткой Author и значением свойства lastname, равным Shakespeare. Результат поиска связывается с идентификатором bard.

С этого момента при использовании в запросе идентификаторов theater, newcastle и bard шаблоны будут присоединяться к реальным узлам графа, связанным с этими тремя идентификаторами. В сущности, эта информация связывает запрос с определенной частью графа, создавая отправные точки, из которых будет производиться выбор прилегающих узлов и взаимосвязей, согласованных с шаблонами непосредственно.

Индексы и ограничения

Индексы помогают оптимизировать процесс поиска конкретных узлов.

Мы можем с удовлетворением заявить, что большую часть времени выполнения графового запроса занимает процесс *поиска* покрытия узлов и взаимосвязей, который решает именно задачу выборки необходимой информации. При обходе взаимосвязей, соответствующих заданному графовому шаблону, происходит выборка элементов для включения в результат запроса. Тем не менее существуют определенные ситуации, когда требуется выбрать конкретные узлы напрямую, а не с помощью обнаружения их при помощи обхода. Например, для идентификации узлов, служащих отправными точками для обхода, нужно найти один или несколько конкретных узлов, основываясь на указанной комбинации меток и значений свойств.

Для увеличения эффективности поиска узлов язык Cypher позволяет создавать индексы комбинаций меток и свойств. Для уникальных значений свойств можно указать ограничения, которые гарантируют их уникальность. Для графа творчества Шекспира, в котором нужен непосредственный поиск мест проведения, можно создать индекс, содержащий все узлы, с метками Venue и базирующийся на свойстве name:

```
CREATE INDEX ON :Venue(name)
```

Чтобы гарантировать уникальность названий стран, можно добавить ограничение уникальности:

```
CREATE CONSTRAINT ON (c:Country) ASSERT c.name IS UNIQUE
```

В существующей базе данных индексы добавляются в фоновом режиме и становятся доступными по мере создания.
Поиск не требует обязательного наличия индексов, но его эффективность может быть улучшена с их помощью. Фраза `MATCH (theater:Venue {name:'Theatre Royal'})` будет работать в любом случае – и при наличии, и при отсутствии индекса. Но в большом наборе данных с несколькими тысячами мест проведения наличие индекса приведет к улучшению производительности. В Neo4j при отсутствии индекса выбор узла Королевского театра в качестве исходной точки запроса вызовет сканирование и фильтрацию всех узлов с меткой `Venue`.

Определение шаблонов для поиска

Фраза `MATCH` в Cypher служит средоточием чудес. В ней используется тот же синтаксис определения шаблона для поиска соответствия в базе данных, что и во фразе `CREATE`, для определения добавляемых сведений. Мы уже рассмотрели ранее простую фразу `MATCH`, теперь приведем более сложный шаблон для поиска всех спектаклей Шекспира, представленных на сцене Королевского театра в городе Ньюкасл:

```
MATCH (theater:Venue {name:'Theatre Royal'}),
      (newcastle:City {name:'Newcastle'}),
      (bard:Author {lastname:'Shakespeare'}),
      (newcastle)<-[:STREET|CITY*1..2]-(theater)
       <-[:VENUE]-()-[:PERFORMANCE_OF]->()
       -[:PRODUCTION_OF]->(play)<-[:WROTE_PLAY]-(bard)
RETURN DISTINCT play.title AS play
```

Этот шаблон во фразе `MATCH` использует несколько новых для нас синтаксических элементов. Совместно с закрепленными узлами, которые обсуждались выше, он использует шаблон, содержащий эти узлы, маршруты разной длины и анонимные узлы. Рассмотрим их все по порядку:

- Идентификаторы `newcastle`, `theater` и `bard` присоединяются к реальным узлам графа на основании указанных меток и значений свойств.
- Если в базе данных присутствует несколько королевских театров (например, Королевские театры существуют в британских городах Плимут, Бат, Винчестер и Норвич), идентификатор `theater` будет связан со всеми этими узлами. Чтобы

ограничить шаблон Королевским театром в Ньюкасле, используется синтаксис `<-[:STREET|CITY*1..2]-`, который означает, что узел `theater` не может отстоять далее двух исходящих взаимосвязей `STREET` и/или `CITY` от узла, представляющего в графе город Ньюкасл-апон-Тайн (Newcastle-upon-Tyne). Позволяя переменную протяженность маршрута, шаблон учитывает детализацию адресной иерархии (включающей, например, улицу, район и город).

- Синтаксис `(theater)<-[:VENUE]-()` использует *анонимный* узел, отсюда и пустые скобки. В данном случае анонимному узлу будут соответствовать спектакли, а так как сведения об отдельных спектаклях не используются ни в самом запросе, ни в его результатах, узлу не присваивается имя и к нему не привязывается идентификатор.
- Вновь анонимный узел применяется для соединения спектакля с организатором (`()-[:PERFORMANCE_OF]->()`). Если нужно возвратить сведения о спектаклях и организаторах, следует заменить вхождения анонимных узлов идентификаторами: `(performance)-[:PERFORMANCE_OF]->(production)`.
- Остальная часть фразы `MATCH` является простым шаблоном узел-взаимосвязь-узел: `(play)<-[:WROTE_PLAY]-(bard)`. Этот шаблон гарантирует возврат только пьес, написанных Шекспиром. Так как узел `()` присоединен к анонимному узлу организатора и через маршрут к спектаклю, можно сделать вывод, что он была поставлен в Ньюкасле в Королевском театре. Присваивая узлу имя `play`, мы делаем возможным его дальнейшее использование в запросе.

Этот запрос вернет все пьесы Шекспира, по которым были поставлены спектакли в Королевском театре в Ньюкасле:

```
+-----------------+
| play            |
+-----------------+
| "Julius Caesar" |
| "The Tempest"   |
+-----------------+
2 rows
```

Запрос отлично справляется с извлечением всей истории постановок Шекспира в Королевском театре, но если потребуется извлечь только определенные пьесы, постановки или выступления, то необходимо каким-то образом ограничить результирующий набор.

Ограничение совпадений

Для ограничения совпадений используется фраза WHERE. Фраза WHERE позволяет исключить из результатов совпавшие подграфы, обеспечивая одно или несколько из следующих условий:

- в совпавших подграфах должны присутствовать (или отсутствовать) указанные маршруты;
- узлы должны иметь указанные метки или взаимосвязи с заданными именами;
- узлы должны иметь (или не иметь) указанные свойства и взаимосвязи вне зависимости от их значений;
- некоторые свойства совпавших узлов и взаимосвязей должны иметь заданные значения;
- должны быть удовлетворены прочие предикаты (например, спектакли должны происходить в указанную дату или до указанной даты).

В отличие от фразы MATCH, описывающей структурные взаимосвязи и назначающей идентификаторы частям шаблона, фраза WHERE только ограничивает совпадение с текущим шаблоном. Допустим, например, что нужно ограничить диапазон пьес лишь *поздним периодом* творчества Шекспира, который принято отсчитывать от 1608 года. Это можно сделать фильтрацией по свойству year взаимосвязей WROTE_PLAY. Для включения такой фильтрации необходимо внести изменения во фразу MATCH, присвоив взаимосвязи WROTE_PLAY идентификатор w (идентификаторы взаимосвязей помещаются перед двоеточием – префиксом взаимосвязей). Затем добавляется фраза WHERE, которая и фильтрует взаимосвязи по значению свойства year:

```
MATCH (theater:Venue {name:'Theatre Royal'}),
      (newcastle:City {name:'Newcastle'}),
      (bard:Author {lastname:'Shakespeare'}),
      (newcastle)<-[:STREET|CITY*1..2]-(theater)
        <-[:VENUE]-()-[:PERFORMANCE_OF]->()
        -[:PRODUCTION_OF]->(play)<-[w:WROTE_PLAY]-(bard)
WHERE w.year > 1608
RETURN DISTINCT play.title AS play
```

Данная фраза WHERE означает, что для каждого найденного совпадения база данных проверит, имеет ли взаимосвязь WROTE_PLAY между узлом, представляющем в графе Шекспира, и пьесой свойство year (год) со значением больше 1608. Такие пьесы пройдут проверку и будут включены в результат. Пьесы, не прошедшие проверки, будут ис-

ключены из результата. Добавление этой фразы гарантирует возврат только пьес из позднего периода творчества Шекспира:

```
+---------------+
| play          |
+---------------+
| "The Tempest" |
+---------------+
1 row
```

Обработка результатов

Фраза RETURN позволяет выполнить некоторую обработку отобранных данных перед возвратом их пользователю (или приложению).

В предыдущих запросах просто возвращались названия отобранных пьес:

```
RETURN DISTINCT play.title AS play
```

Ключевое слово DISTINCT гарантирует уникальность возвращаемых значений. Так как пьеса может быть поставлена в одном том же театре несколько раз, названия пьес могут повторяться. Добавление ключевого слова DISTINCT позволяет отсеять дубликаты.

Мы можем улучшить результат несколькими способами, в том числе группировкой, упорядочиванием, фильтрацией и ограничением количества возвращаемых данных. Например, чтобы получить лишь *количество* пьес, соответствующих заданным критериям, применяется функция count:

```
RETURN count(play)
```

Если необходимо упорядочить пьесы по количеству спектаклей, нужно сначала во фразе MATCH связать взаимосвязи PERFORMANCE_OF с идентификатором p, подсчитать их с помощью функции count и упорядочить:

```
MATCH (theater:Venue {name:'Theatre Royal'}),
      (newcastle:City {name:'Newcastle'}),
      (bard:Author {lastname:'Shakespeare'}),
      (newcastle)<-[:STREET|CITY*1..2]-(theater)
        <-[:VENUE]-()-[p:PERFORMANCE_OF]->()
        -[:PRODUCTION_OF]->(play)<-[:WROTE_PLAY]-(bard)
RETURN play.title AS play, count(p) AS performance_count
ORDER BY performance_count DESC
```

Здесь фраза RETURN подсчитывает количество взаимосвязей PERFORMANCE_OF , используя идентификатор p (который связан во фра-

зе MATCH со взаимосвязями PERFORMANCE_OF), и присваивает результату псевдоним performance_count. Затем она упорядочивает результаты, основываясь на значениях performance_count, помещая наиболее часто исполняемые пьесы вперед:

```
+-----------------------------------+
| play            | performance_count |
+-----------------------------------+
| "Julius Caesar" | 2                 |
| "The Tempest"   | 1                 |
+-----------------------------------+
2 rows
```

Цепочки в запросах

Прежде чем завершить краткий тур по языку Cypher, остановимся на еще одной его особенности – фразе WITH. Бывают ситуации, когда неудобно (или невозможно) вместить все, что требуется, в одну фразу MATCH. Фраза WITH позволяет выстроить цепочку сопоставлений, передавая результаты предыдущих частей запроса в последующие. В примере ниже выполняется поиск пьес, написанных Шекспиром, и их упорядочение по годам, причем написанные последними пьесы будут помещены вперед. С помощью фразы WITH результаты передаются во фразу RETURN, где функция collect генерирует список названий пьес, разделенных запятыми:

```
MATCH (bard:Author {lastname:'Shakespeare'})-[w:WROTE_PLAY]->(play)
WITH play
ORDER BY w.year DESC
RETURN collect(play.title) AS plays
```

Этот запрос для нашего примера графа вернет следующий результат:

```
+---------------------------------+
| plays                           |
+---------------------------------+
| ["The Tempest","Julius Caesar"] |
+---------------------------------+
1 row
```

Фразы WITH могут применяться для отделения фраз, служащих только для чтения, от операций SET, осуществляющих запись. В целом фразы WITH служат для применения к сложным запросам принципа «разделяй и властвуй», позволяя разбивать сложный запрос на несколько простых шаблонов.

Распространенные просчеты при моделировании

Графовое моделирование является очень наглядным способом представления сложных прикладных областей, но его использование само по себе не дает гарантий достижения поставленных целей. Действительно, даже те, кто работает с графами ежедневно, порой допускают ошибки. В этом разделе будут рассмотрены модели, в которых что-то пошло не так. При этом будет описано, как на ранней стадии выявить просчеты в моделях и как их исправить.

Проблемы анализа источников электронных писем

В качестве примера рассмотрим анализ переписки по электронной почте. Анализ модели коммуникаций является классической графовой задачей, включающей графовые запросы для выявления экспертов в определенных областях, оказывающих важное влияние на распространяемую информацию, и каналов связи для распространения информации. Но здесь целью будет не поиск источников позитивного влияния (экспертов), а идентификация мошенников с помощью выявления подозрительных проявлений в электронной переписке, нарушающих принятые правила или даже законы.

Первый блин комом?

При анализе выбранной прикладной области определяются применяемые мошенниками способы запутывания следов: скрытые копии (*blind-copy*, сокращенно BCC), псевдонимы при переписке для имитации легальных взаимоотношений между реальными партнерами по бизнесу. На основе такого анализа подготавливается графовая модель, которая, казалось бы, охватывает все соответствующие сущности и их действия.

Для иллюстрации начальной модели создадим с помощью фразы `CREATE` несколько узлов, представляющих пользователей и псевдонимы. Также сгенерируем взаимосвязь, указывающую, что Алиса (`Alice`) является одним из псевдонимов Боба (`Bob`). (Предполагается, что графовая база данных проиндексирует эти узлы так, что позже их можно будет просматривать и использовать в качестве отправных точек в запросах.) Следующий Cypher-запрос создаст первый граф:

```
CREATE (alice:User {username:'Alice'}),
       (bob:User {username:'Bob'}),
       (charlie:User {username:'Charlie'}),
```

```
    (davina:User {username:'Davina'}),
    (edward:User {username:'Edward'}),
    (alice)-[:ALIAS_OF]->(bob)
```

Из полученной графовой модели легко определить, что *Алиса является псевдонимом Боба*, как это показано на рис. 3.7.

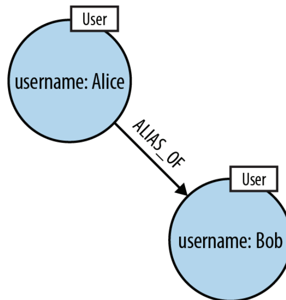


Рис. 3.7 ❖ Пользователи и псевдонимы

Затем добавим к пользователям их обмен по электронной почте:

```
MATCH (bob:User {username:'Bob'}),
      (charlie:User {username:'Charlie'}),
      (davina:User {username:'Davina'}),
      (edward:User {username:'Edward'})
CREATE (bob)-[:EMAILED]->(charlie),
       (bob)-[:CC]->(davina),
       (bob)-[:BCC]->(edward)
```

На первый взгляд, граф дает достаточно точное представление о прикладной области. Каждая фраза удобно читается слева направо, тем самым соответствуя одному из неофициальных тестов корректности. Например, из графа можно узнать, что «Боб (`Bob`) послал электронное письмо Чарли (`Charlie`)». Ограничения этой модели проявятся, только когда нужно будет точно определить, *что* рассылает потенциальный мошенник Боб (и его альтернативное эго Алиса). Можно узнать, что Боб включает некоторых людей в заголовки CC или BCC, но нельзя ознакомиться с самым важным – содержанием электронных писем.

Эта первая попытка моделирования привела к созданию графа в виде звезды с Бобом в центре. Действия Боба, такие как отправка электронных писем, их копий и скрытых копий, представлены

в виде взаимосвязей, которые простираются от узла, представляющего Боба, к узлам, представляющим получателей. Но, как можно увидеть на рис. 3.8, недостает наиболее важного элемента данных, самих *электронных писем*.

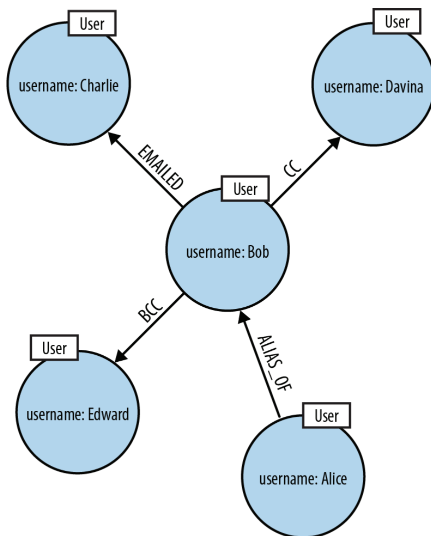


Рис. 3.8 ❖ Отсутствие узлов электронных писем ведет к потере информации

Факт потери информации с такой организацией графа становится очевидным при выполнении следующего запроса:

```
MATCH (bob:User {username:'Bob'})-[e:EMAILED]->
      (charlie:User {username:'Charlie'})
RETURN e
```

Этот запрос возвращает взаимосвязи `EMAILED` (отправил электронное письмо) между Бобом и Чарли (их будет столько же, сколько электронных писем Боб отправил Чарли). Результат сообщает только о факте обмена электронными письмами, но не о самих письмах:

```
+-----------------+
| e               |
+-----------------+
| :EMAILED[1] {} |
+-----------------+
1 row
```

Можно было бы подумать, что, добавив взаимосвязям EMAILED свойства, представляющие атрибуты электронных писем, можно исправить ситуацию, но это ничего не даст. Даже со свойствами, прикрепленными к взаимосвязям EMAILED, невозможно отследить соотношение между взаимосвязями EMAILED, CC и BCC, т. е. нельзя выяснить, для каких писем были созданы копии или скрытые копии и кому они были отправлены.

Дело в том, что при моделировании был допущен промах, обусловленный скорее вольным обращением с английским языком, чем недостатками теории графов. Обычная практика использования языка сосредоточила наше внимание на глаголе «emailed» (отправил электронное письмо), а не на самом электронном письме, и в результате модель утеряла правильное понимание прикладной области.

В английском языке обычным делом является сокращение фразы «Bob sent an email to Charlie» («Боб написал электронное письмо Чарли») до «Bob emailed Charlie» («Боб написал Чарли»). В большинстве случаев потеря существительного (email – электронное письмо) не имеет никакого значения, потому что смысл фразы остается тем же. Но в данной ситуации такое сокращение приводит к проблеме. Направленность та же, но сведения о количестве, содержании и получателях писем, отправленных Бобом, во взаимосвязях EMAILED будут отсутствовать и не войдут в модель явно, в виде собственных узлов.

Со второго раза все получится

Чтобы исправить модель, приводящую к потере данных, нужно добавить узлы, представляющие реальные электронные письма, и расширить набор видов взаимосвязей, чтобы полностью охватить поля адресов, поддерживаемых электронной почтой. Структуры, приводящие к потерям информации, подобные следующей:

```
CREATE (bob)-[:EMAILED]->(charlie)
```

следует заменить более подробными:

```
CREATE (email_1:Email {id:'1', content:'Hi Charlie, ... Kind regards, Bob'}),
       (bob)-[:SENT]->(email_1),
       (email_1)-[:TO]->(charlie),
       (email_1)-[:CC]->(davina),
       (email_1)-[:CC]->(alice),
       (email_1)-[:BCC]->(edward)
```

При этом будет создан другой граф в виде звезды, с электронным письмом в центре, как показано на рис. 3.9.

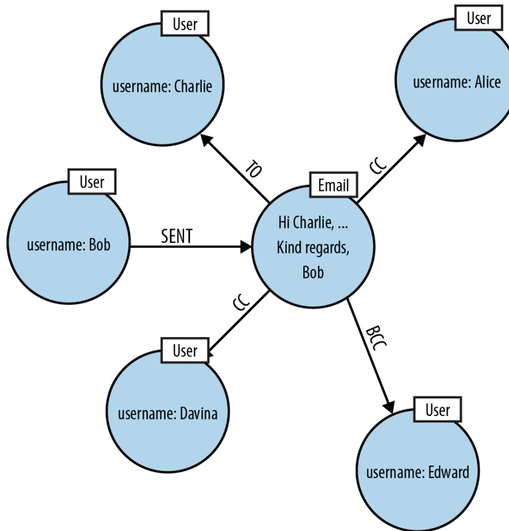


Рис. 3.9 ❖ Граф в виде звезды с электронным письмом в центре

Конечно, в реальной системе таких электронных писем будет гораздо больше, и каждое из них будет иметь свою запутанную сеть связей, которую требуется проанализировать. Со временем сервером электронной почты будет выполнено множество операторов `CREATE`, подобных приведенным ниже (для краткости в них опущены опорные узлы):

```
CREATE (email_1:Email {id:'1', content:'email contents'}),
       (bob)-[:SENT]->(email_1),
       (email_1)-[:TO]->(charlie),
       (email_1)-[:CC]->(davina),
       (email_1)-[:CC]->(alice),
       (email_1)-[:BCC]->(edward);
CREATE (email_2:Email {id:'2', content:'email contents'}),
       (bob)-[:SENT]->(email_2),
       (email_2)-[:TO]->(davina),
       (email_2)-[:BCC]->(edward);
CREATE (email_3:Email {id:'3', content:'email contents'}),
       (davina)-[:SENT]->(email_3),
       (email_3)-[:TO]->(bob),
       (email_3)-[:CC]->(edward);
CREATE (email_4:Email {id:'4', content:'email contents'}),
```

```
       (charlie)-[:SENT]->(email_4),
       (email_4)-[:TO]->(bob),
       (email_4)-[:TO]->(davina),
       (email_4)-[:TO]->(edward);
CREATE (email_5:Email {id:'5', content:'email contents'}),
       (davina)-[:SENT]->(email_5),
       (email_5)-[:TO]->(alice),
       (email_5)-[:BCC]->(bob),
       (email_5)-[:BCC]->(edward);
```

В результате будет создан более сложный и интересный граф, изображенный на рис. 3.10.

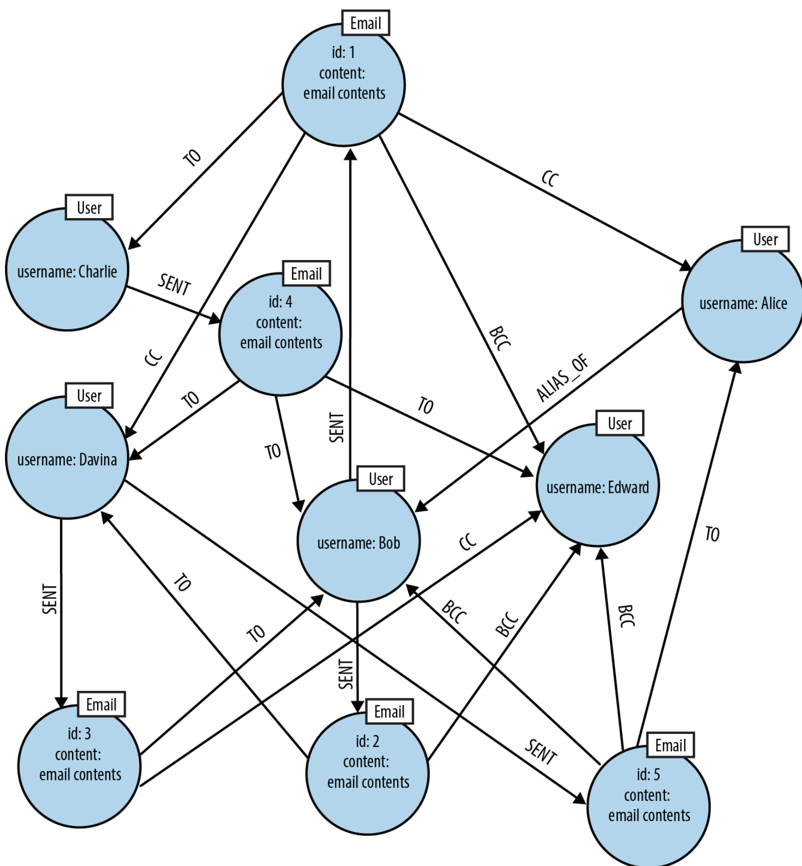


Рис. 3.10 ❖ Граф взаимодействий по электронной почте

Теперь можно выполнить запрос для выявления подозрительных действий:

```
MATCH (bob:User {username:'Bob'})-[:SENT]->(email)-[:CC]->(alias),
      (alias)-[:ALIAS_OF]->(bob)
RETURN email.id
```

Результатом его будут все электронные письма, отправленные Бобом, где в адресе копии CC присутствует один из его псевдонимов. Любое электронное письмо, соответствующее этому шаблону, свидетельствует о подозрительном поведении. А так как язык Cypher и графовая база данных основаны на графах, запросы даже на больших наборах данных работают очень быстро. Этот запрос вернет следующий результат:

```
+------------------------------------------+
| email                                    |
+------------------------------------------+
| Node[6]{id:"1",content:"email contents"} |
+------------------------------------------+
1 row
```

Эволюция прикладной области

Как и любая другая база данных, графовая база данных предназначена для моделирования эволюционирующей системы. Так что же делать при эволюции системы? Как узнать, что что-то нарушилось и где именно произошли эти нарушения? В графовой базе данных невозможно полностью отказаться от миграций, они необходимы в любом хранилище данных. Но в графовой базе данных миграции выполняются гораздо проще.

Чтобы добавить в граф новые факты или композиции, часто достаточно добавить новые узлы и взаимосвязи, не затрагивая самой модели. Добавление *новых* видов взаимосвязей не повлияет на уже существующие запросы и не представляет никакой опасности. Внесение изменений в *существующие* виды взаимосвязей и корректировка свойств (самих свойств, а не только их значений) существующих узлов также *могут* быть безопасными, но требуют пересмотра существующих запросов, чтобы удостовериться, что они все еще выполняют свои задачи после структурных изменений. Но эти операции ничем не отличаются от действий, выполняемых при нормальном функционировании базы данных, поэтому миграции в графовых базах данных являются простым и обычным делом.

На данный момент у нас имеется граф, описывающий отправителей и получателей электронных писем, а также их содержимое. Но и, конечно, одним из преимуществ электронной почты является возможность пересылать или отвечать на полученную электронную почту. Это увеличивает эффективность взаимодействий и ускоряет обмен информацией, однако в некоторых случаях приводит к утечке секретных сведений. При поиске подозрительных шаблонов переписки имеет смысл обратить внимание на пересылаемые сообщения и ответы на них.

На первый взгляд покажется, что обновление графа, связанное с поддержкой новых требований, не потребует миграции базы данных. Для этого нужно лишь добавить виды взаимосвязей FORWARDED (переслал) и REPLIED_TO (ответил), как показано на рис. 3.11. Это никак не повлияет на уже существующие запросы, потому что они не способны распознать новые взаимосвязи.

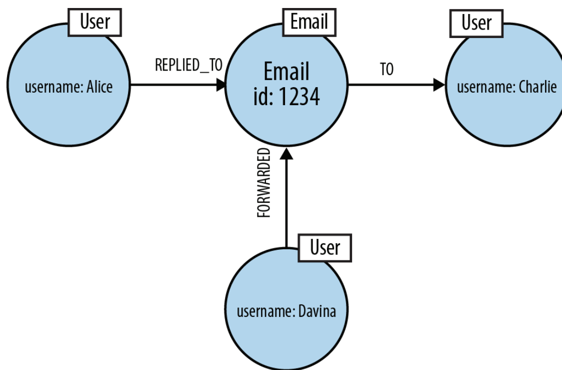


Рис. 3.11 ❖ Наивный подход, ведущий к потере данных и не учитывающий, что пересылка и ответ на электронные письма здесь являются самыми важными сущностями

Но такой подход не лишен недостатков, которые очень быстро дадут о себе знать. Добавление взаимосвязей FORWARDED и REPLIED приведет к потерям информации, как в первом варианте графа со взаимосвязями EMAILED. Для иллюстрации рассмотрим следующий оператор CREATE:

```
...
MATCH (email:Email {id:'1234'})
CREATE (alice)-[:REPLIED_TO]->(email)
CREATE  (davina)-[:FORWARDED]->(email)-[:TO]->(charlie)
```

Первый оператор CREATE фиксирует тот факт, что Алиса (Alice) ответила на указанное электронное письмо. Оператор вполне логичен при чтении слева направо, но все портит невозможность узнать, ответила ли Алиса всем получателям электронного письма email или только автору. Мы знаем лишь, что был отправлен некий ответ. Второй оператор также хорошо читается слева направо: Давина (Davina) отправила электронное письмо Чарли (Charlie). Но мы уже используем взаимосвязь TO, чтобы указать, что данное электронное письмо имеет в заголовке список первичных адресатов. Повторное применение взаимосвязи TO делает невозможным выяснение, кто был адресатом и кто получил начальную версию электронного письма.

Чтобы решить эту проблему, нужно еще раз проанализировать прикладную область. Ответ на электронное письмо сам по себе является новым электронным письмом (Email), но еще и ответом (Reply). Другими словами, у ответа есть две роли, которые в графе можно представить посредством присоединения двух меток – Email и Reply – к узлу ответа. Кому отправлен ответ – самому отправителю, всем получателям или их подмножеству, легко можно указать с помощью уже знакомых взаимосвязей TO, CC и BCC, в то время как само оригинальное электронное письмо можно описать с помощью взаимосвязи REPLY_TO. Ниже приводится пересмотренный набор операций записи результатов нескольких событий электронной почты (здесь так же опущена привязка узлов):

```
CREATE (email_6:Email {id:'6', content:'email'}),
       (bob)-[:SENT]->(email_6),
       (email_6)-[:TO]->(charlie),
       (email_6)-[:TO]->(davina);
CREATE (reply_1:Email:Reply {id:'7', content:'response'}),
       (reply_1)-[:REPLY_TO]->(email_6),
       (davina)-[:SENT]->(reply_1),
       (reply_1)-[:TO]->(bob),
       (reply_1)-[:TO]->(charlie);
CREATE (reply_2:Email:Reply {id:'8', content:'response'}),
       (reply_2)-[:REPLY_TO]->(email_6),
       (bob)-[:SENT]->(reply_2),
       (reply_2)-[:TO]->(davina),
       (reply_2)-[:TO]->(charlie),
       (reply_2)-[:CC]->(alice);
CREATE (reply_3:Email:Reply {id:'9', content:'response'}),
       (reply_3)-[:REPLY_TO]->(reply_1),
       (charlie)-[:SENT]->(reply_3),
```

```
      (reply_3)-[:TO]->(bob),
      (reply_3)-[:TO]->(davina);
CREATE (reply_4:Email:Reply {id:'10', content:'response'}),
      (reply_4)-[:REPLY_TO]->(reply_3),
      (bob)-[:SENT]->(reply_4),
      (reply_4)-[:TO]->(charlie),
      (reply_4)-[:TO]->(davina);
```

Их выполнение приведет к созданию графа, изображенного на рис. 3.12, который включает множество ответов и ответов на ответы.

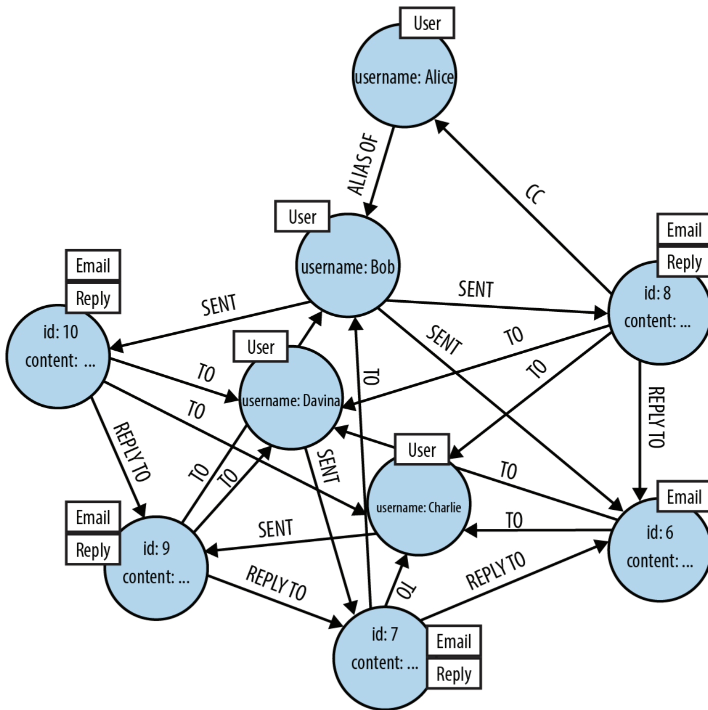


Рис. 3.12 ❖ Явное моделирование ответов с высокой точностью

Здесь легко можно увидеть, кто ответил на начальное сообщение Боба. Сначала отыскивается интересующее письмо, затем все совпадения со входящими взаимосвязями `REPLY_TO` (ответов может быть несколько) и, наконец, совпадения с входящими взаимосвязями `SENT`, это и будет отправитель или отправители. Реализовать это на язы-

ке Cypher достаточно просто. На самом деле Cypher позволяет легко найти ответы на ответы на ответы и т. д., вплоть до произвольной глубины (здесь глубина ограничена четырьмя уровнями):

```
MATCH p=(email:Email {id:'6'})<-[:REPLY_TO*1..4]-(:Reply)<-[:SENT]-(replier)
RETURN replier.username AS replier, length(p) - 1 AS depth
ORDER BY depth
```

Здесь каждый из найденных путей связывается с идентификатором `p`. Затем во фразе `RETURN` вычисляется длина цепочки ответов (за вычетом 1 для учета взаимосвязи `SENT`), и возвращается имя отправителя и глубина его ответов. Этот запрос вернет следующий результат:

```
+-------------------+
| replier   | depth |
+-------------------+
| "Davina"  | 1     |
| "Bob"     | 1     |
| "Charlie" | 2     |
| "Bob"     | 3     |
+-------------------+
4 rows
```

Как видите, Давина (`Davina`) и Боб (`Bob`) ответили непосредственно на исходное сообщение Боба, Чарли (`Charlie`) ответил на один из ответов, и Боб ответил на один из ответов на ответ.

Аналогично выглядит шаблон для пересылаемых электронных писем, которые можно рассматривать как новые электронные письма, случайно содержащие часть текста исходного электронного письма. Как и при ответах, новые электронные письма моделируются явно. По ссылке на оригинальное пересылаемое электронное письмо можно всегда точно отследить его происхождение. То же относится к получателям отправленных ими самими электронных писем. Например, если Алиса (альтернативное эго Боба) отправила электронное письмо Бобу, чтобы попытаться идентифицировать неких конкретных особ, а затем Боб (продолжая задуманное) пересылает его Чарли, который пересылает его Давине, нужно выполнить анализ трех писем. Если предположить, что пользователи (и их псевдонимы) уже имеются в базе данных, тогда включить необходимую для аудита информацию можно с помощью следующих команд на языке Cypher:

```
CREATE (email_11:Email {id:'11', content:'email'}),
       (alice)-[:SENT]->(email_11)-[:TO]->(bob);
CREATE (email_12:Email:Forward {id:'12', content:'email'}),
```

```
      (email_12)-[:FORWARD_OF]->(email_11),
      (bob)-[:SENT]->(email_12)-[:TO]->(charlie);
CREATE (email_13:Email:Forward {id:'13', content:'email'}),
      (email_13)-[:FORWARD_OF]->(email_12),
      (charlie)-[:SENT]->(email_13)-[:TO]->(davina);
```

После выполнения этих операций база данных будет содержать подграф, изображенный на рис. 3.13.

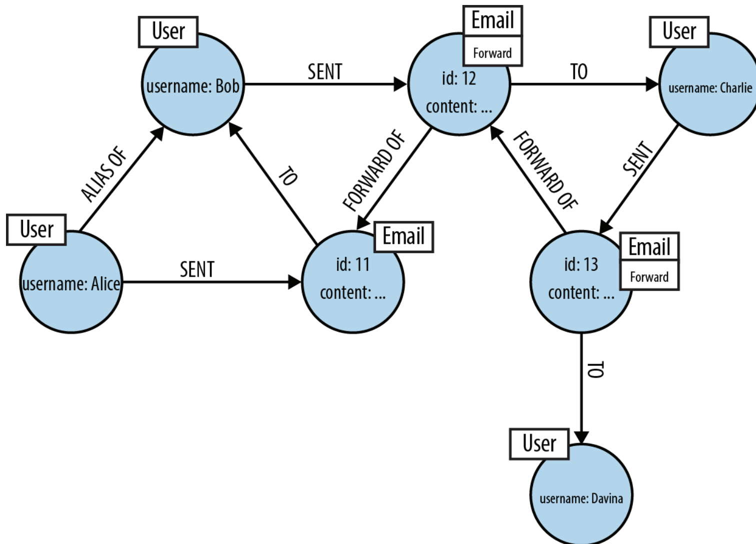


Рис. 3.13 ❖ Явное моделирование отсылки электронного письма

Используя этот граф, можно получить разные маршруты в цепочке пересылки электронных писем.

```
MATCH (email:Email {id:'11'})<-[f:FORWARD_OF*]-(:Forward)
RETURN count(f)
```

Этот запрос выполняет поиск всех входящих взаимосвязей `FORWARD_OF` в дереве пересылаемых писем, независимо от глубины, начиная с заданного письма. Этим взаимосвязям присваивается идентификатор `f`. Для расчета количества пересылок электронных писем подсчитывается количество взаимоотношений `FORWARD_OF` с помощью функции Cypher `count`. В этом примере оригинальное электронное письмо было отправлено дважды:

```
+----------+
| count(f) |
+----------+
| 2        |
+----------+
1 row
```

Идентификация узлов и взаимосвязей

Процесс моделирования лучше рассматривать как попытку создания структуры графа, обеспечивающей ответы на вопросы в прикладной области. То есть проектирование основывается на возможности выполнения запросов:

1. Опишите цели клиента или конечного пользователя, заказывающего модель.
2. Преобразуйте эти цели в вопросы для прикладной области.
3. Определите сущности и взаимосвязи, фигурирующие в вопросах.
4. Переведите сущности и взаимосвязи в Cypher-выражения.
5. Выразите вопросы в виде графовых шаблонов, используя маршрутные выражения, схожие с выражениями, использованными для моделирования прикладной области.

Анализируя предложения, описывающие прикладную область, можно легко определить основные элементы графа:

- нарицательные существительные становятся метками, например «пользователь» и «электронное письмо» превращаются в метки `User` и `Email`;
- глаголы преобразуются в имена взаимосвязей, например «отослал» и «написал» становятся именами `SENT` и `WROTE`;
- имена собственные, например имена людей или названия компаний, соотносятся с отдельными экземплярами, представленными в модели в виде узлов, с помощью одного или нескольких свойств, отображающих соответствующие атрибуты.

Как избежать антишаблонов

Как правило, сущностям не ставятся в соответствие взаимосвязи. Назначение взаимосвязей – показать, *как* сущности связаны между собой, и отразить качество их отношений.

Сущности прикладной области не всегда удается сразу распознать в текстовом представлении, следовательно, нужно проанализировать,

чем в действительности являются существительные. Особенности обычного языка, позволяющие легко превращать существительные в глаголы (термин «*verbing*» в английском языке), часто скрывают существительные и соответствующие им сущности прикладной области. Технический и деловой жаргон особенно изобилует такими неологизмами, например уже знакомое нам «email» вместо «send an email» или «google» («гуглить») вместо «search Google» («искать в Google»).

Важно также осознавать, что графы являются естественно расширяемой структурой. Для них вполне естественным является добавление сущностей прикладной области и их взаимодействий в виде новых узлов и взаимосвязей, даже если это приводит к значительному увеличению хранимых базой данных сведений. В общем случае не имеет смысла объединять элементы данных при записи, чтобы повысить эффективность запросов. Моделирование выполняется в соответствии с вопросами, на которые должны предоставлять ответы данные, и направлено на точное представление прикладной области. Получив модель данных, можно не беспокоиться о способности графовой базы данных извлечь нужные сведения.

В графовых базах данных запросы выполняются быстро, даже при хранении очень больших объемов данных. Изучая графовые базы данных, основное внимание следует уделять структуре графов, а не их денормализации.

Итоги

Графовые базы данных позволяют представлять прикладные области в виде графов, хранить их в виде графов и выполнять графовые запросы. Графы используются для четкого представления прикладной области, а графовые базы данных позволяют хранить это представление, не нарушая тесной взаимосвязи между прикладной областью и данными. Кроме того, графовое моделирование полностью устраняет необходимость нормализовать и денормализовать данные с помощью сложного кода управления данными.

Однако многие из нас еще не знакомы с моделированием графами. Создаваемые графы должны хорошо читаться с помощью запросов, а исключение соединений сущностей и действий влечет потерю полезных сведений о прикладной области. Несмотря на отсутствие

неких абсолютных правил моделирования, приведенное в этой главе руководство поможет вам в создании графов данных, обеспечивающих потребности разрабатываемых систем и не отстающих от их развития.

Вооружившись пониманием графового моделирования, можно перейти к описанию реализации графовой базы данных. В следующей главе мы рассмотрим проектирование и разработку решения, основанного на графовой базе данных.

Глава 4

Разработка приложений графовых баз данных

Эта глава посвящена описанию практической работы с графовой базой данных. В предыдущих главах мы ознакомились с графами данных, в этой главе мы применим полученные знания для разработки приложений, основанных на графовых базах данных. Здесь будут рассмотрены отдельные вопросы, возникающие при моделировании данных, и некоторые варианты архитектуры приложений.

Исходя из нашего опыта, разработка приложения графовых баз данных прекрасно стыкуется с популярными сейчас эволюционными, поэтапными и итерационными методами разработки программного обеспечения. Ключевой особенностью этих методов является непрерывное тестирование в течение всего цикла разработки программного обеспечения. Здесь будет продемонстрирована разработка моделей данных и приложений в тесно связанном с тестированием режиме.

В конце главы будут рассмотрены некоторые проблемы, которые следует учитывать при планировании.

Моделирование данных

В главе 3 мы подробно рассмотрели особенности моделирования и использования графов данных. Здесь будут приведены лишь наиболее важные принципы моделирования и рассмотрены реализации графовой модели данных, соответствующие итерационным и поэтапным методам разработки программного обеспечения.

Описание модели с учетом потребностей приложения

Вопросы, ответы на которые потребуется получить, основываясь на исходных данных, помогут определиться с сущностями и взаимосвязями. Полученные из непосредственного общения в форме рассказа пожелания пользователей представляют собой взгляд извне, с точки зрения пользователя, на требования к программе, а также очерчивают вопросы, которые возникают при удовлетворении этих требований[1]. Ниже приводится пример пользовательской истории о пожеланиях для веб-приложения обзора книжной продукции:

> **КАК** читатель, которому понравилась книга, **Я ХОЧУ** знать, какие еще книги нравятся другим читателям, которым понравилась та же книга, что и мне, **ЧТОБЫ** я мог выбрать другие книги для чтения.

Эта история выражает потребности пользователя, которые определяют форму и содержание модели данных. С точки зрения моделирования данных, фраза КАК устанавливает контекст, включающий две сущности, читатель и книга, плюс взаимосвязь НРАВИТСЯ, соединяющую их. Фраза Я ХОЧУ ставит вопрос: какие книги нравятся читателям, которым понравилась книга, понравившаяся мне? Этот вопрос предполагает увеличение количества взаимосвязей НРАВИТСЯ и сущностей, отражающих других читателей и другие книги.

Сущности и взаимосвязи, полученные при анализе рассказа пользователя, можно с легкостью перенести в простую модель данных, изображенную на рис. 4.1.

Так как модель данных непосредственно выражает вопрос из истории пользователя, она определяет и запрос, отображающий структуру нужного вопроса: *Алисе* (`Alice`) *нравится книга «Дюна»* (`Dune`), *какие книги нравятся другим людям, которым нравится книга «Дюна»*:

```
MATCH (:Reader {name:'Alice'})-[:LIKES]->(:Book {title:'Dune'})
      <-[:LIKES]-(:Reader)-[:LIKES]->(books:Book)
RETURN books.title
```

[1] Больше узнать о пожеланиях пользователей можно из книги Майка Кона (Mike Cohn) «User Stories Applied» (Addison-Wesley, 2004): Майк Кон. Пользовательские истории. Гибкая разработка программного обеспечения. М.: Вильямс, 2012. ISBN: 978-5-8459-1795-9.

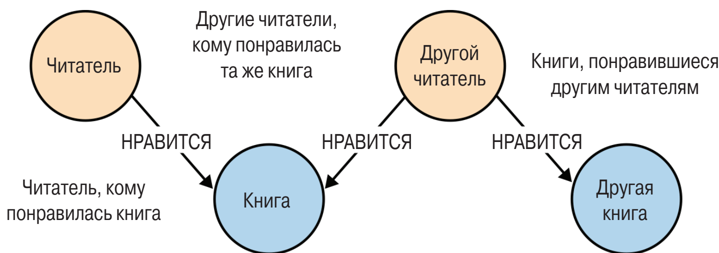


Рис. 4.1 ❖ Модель данных обзора книг, основанная на рассказе пользователя

Узлы представляют сущности, взаимосвязи формируют структуру

Следующие общие принципы помогают определить, когда использовать узлы, а когда взаимосвязи, даже при том, что их можно применить не всегда:

- используйте узлы для представления сущностей, т. е. интересующих вас *вещей* прикладной области, которые можно снабдить метками и сгруппировать;
- используйте взаимосвязи для представления *взаимоотношений* между сущностями и определения семантического контекста каждой сущности, тем самым формируя структуру прикладной области;
- используйте направления взаимосвязей для уточнения семантики взаимосвязей. Большинство взаимоотношений асимметричны, поэтому взаимосвязи в графе со свойствами всегда имеют направление. Для извлечения двунаправленных взаимосвязей необходимо создавать запросы, игнорирующие направление, а не искать пару взаимосвязей;
- используйте свойства узлов для представления атрибутов сущностей, а также любых метаданных, например моментов времени, номеров версий и т. д.;
- используйте свойства взаимосвязей для выражения силы, веса или качества взаимоотношений, а также любых метаданных взаимоотношений, например моментов времени, номеров версий и т. д.

Их соблюдение обеспечивает надлежащий охват и отражение сущностей прикладной области. В главе 3 был описан пример ошибки моделирования, когда вместо узлов были созданы связи с небрежно подо-

бранными названиями. Если решено использовать взаимосвязи для моделирования сущностей, например электронных писем или отзывов, необходимо удостовериться, что сущность не может быть связана с более чем двумя другими сущностями. Вспомним, что взаимосвязи всегда должны быть соединены с двумя узлами: начальным и конечным, причем узлов должно быть не больше и не меньше. Если позже обнаруживается, что сущность следует соединить более чем с двумя другими сущностями, нужно провести реорганизацию структуры, выделив взаимосвязь в отдельный узел. Это приведет к коренным изменениям в модели данных и, скорее всего, потребует исправить запросы и код приложения, которые записывают и читают данные.

Подробные имена или свойства взаимосвязей

При проектировании взаимосвязей следует помнить о компромиссе между использованием подробных имен взаимосвязей и их квалификационных свойств. То есть либо использовать имена `DELIVERY_ADDRESS` (адрес доставки) и `HOME_ADDRESS` (домашний адрес), либо имена `ADDRESS {type:'delivery'}` и `ADDRESS {type:'home'}` с квалификационными признаками.

Взаимосвязи являются важной составляющей графа. Разделение взаимосвязей по именам является лучшим способом избежать обхода больших участков графа. Использование одного или нескольких значений свойств, чтобы проверить – стоит ли перемещаться по взаимосвязи, влечет за собой дополнительные операции ввода-вывода, так как свойства хранятся в отдельном файле, но только при первом обращении (затем они кэшируются).

Подробные имена взаимосвязей используются при наличии конечного множества имен взаимосвязей. Если взаимосвязи обладают весом, как того требует алгоритм поиска кратчайших маршрутов, множество их имен вряд ли будет конечным, и следует воспользоваться свойствами взаимосвязей.

Впрочем, бывают случаи, когда имеется конечный набор имен взаимосвязей, но при обходах требуется выбирать как конкретные виды взаимосвязей, так и все взаимосвязи, независимо от их вида. Хорошим примером являются адреса. Исходя из принципа конечного множества имен, можно создать взаимосвязи `HOME_ADDRESS`, `WORK_ADDRESS` и `DELIVERY_ADDRESS`. Это позволит учитывать только конкретный вид адресных взаимосвязей (например, `DELIVERY_ADDRESS`), игнорируя все остальные. Но как поступить, если нужно найти все адреса пользователя? Существует несколько вариантов. Во-первых,

можно перечислить в коде запроса *все* виды взаимосвязей, например `MATCH (user)-[:HOME_ADDRESS|WORK_ADDRESS|DELIVERY_ADDRESS]->(address)`. Но такая запись получится слишком громоздкой, если видов взаимосвязей окажется достаточно много. Кроме того, можно добавить в модель более обобщенные взаимосвязи `ADDRESS`, вдобавок ко взаимосвязям с подробными именами. Каждый узел, представляющий в модели адрес, будет подключен к пользователю двумя взаимосвязями: взаимосвязью с подробным именем (например, `DELIVERY_ADDRESS`) *и* более обобщенной взаимосвязью `ADDRESS {type:'delivery'}`.

Как уже отмечалось в разделе «Описание модели с точки зрения потребностей приложения», ключевым моментом при проектировании взаимосвязей модели являются вопросы, на которые требуется получить ответы, основанные на данных.

Моделирование фактов в виде узлов

При взаимодействии двух или более сущностей прикладной области в течение определенного периода времени возникает факт. Факт представляется отдельным узлом, соединенным со всеми сущностями, причастными к этому факту. Моделирование действия, основанное на его результате, т. е. на *вещи*, возникшей в результате действия, приводит к созданию промежуточного узла, представляющего результат взаимодействия между двумя или более сущностями. Таким узлам можно придать свойства, отображающие их начальный и конечный моменты времени.

Следующие примеры демонстрируют моделирование фактов и действий с помощью промежуточных узлов.

Работа

Факт приема Яна (`Ian`) на работу в компанию Neo Technology инженером (`engineer`) представлен в виде графа на рис. 4.2.

На языке Cypher это можно записать так:

```
CREATE (:Person {name:'Ian'})-[:EMPLOYMENT]->
        (employment:Job {start_date:'2011-01-05'})
        -[:EMPLOYER]->(:Company {name:'Neo'}),
      (employment)-[:ROLE]->(:Role {name:'engineer'})
```

Исполнение ролей

Факт, что Уильям Хартнелл (`William Hartnell`) сыграл роль доктора (`The Doctor`) в серии «Сенсориты» (`Sensorites`), представлен в виде графа на рис. 4.3.

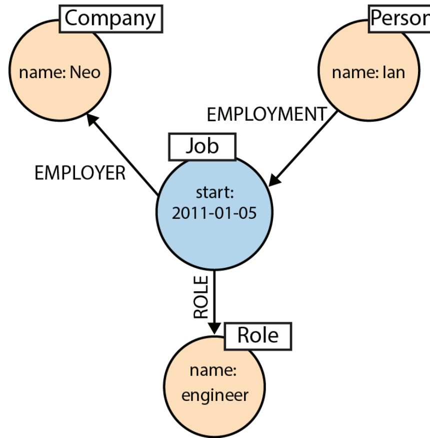


Рис. 4.2 ❖ Ян принят на работу инженером в компанию Neo Technology

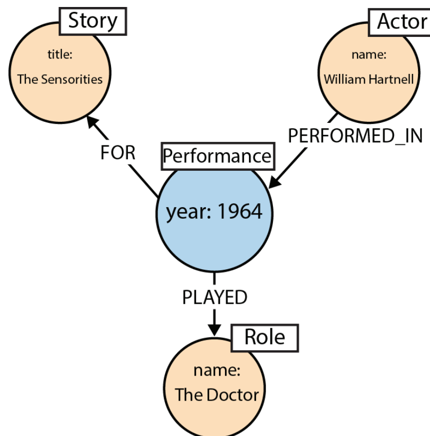


Рис. 4.3 ❖ Уильям Хартнелл сыграл доктора в серии «Сенсориты»

На языке Cypher:

```
CREATE (:Actor {name:'William Hartnell'})-[:PERFORMED_IN]->
         (performance:Performance {year:1964})-[:PLAYED]->
         (:Role {name:'The Doctor'}),
       (performance)-[:FOR]->(:Story {title:'The Sensorites'})
```

Электронная переписка

Рисунок 4.4 демонстрирует отправку Яном (`Ian`) электронного письма Джиму (`Jim`) и его копии Алистеру (`Alistair`).

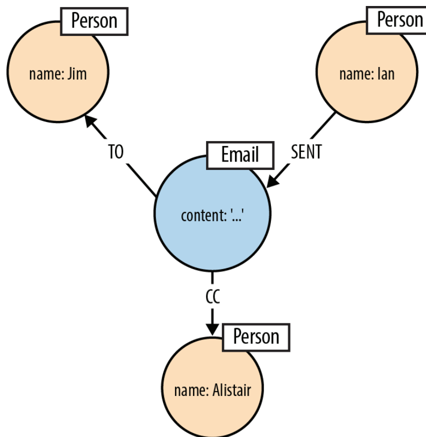


Рис. 4.4 ❖ Ян послал электронное письмо Джиму и его копию Алистеру

На языке Cypher это можно выразить следующим образом:

```
CREATE (:Person {name:'Ian'})-[:SENT]->(e:Email {content:'...'})
         -[:TO]->(:Person {name:'Jim'}),
       (e)-[:CC]->(:Person {name:'Alistair'})
```

Рецензии

На рис. 4.5 изображен граф, представляющий действие Алистера (`Alistair`), написавшего рецензию на фильм.

На языке Cypher:

```
CREATE (:Person {name:'Alistair'})-[:WROTE]->
         (review:Review {text:'...'})-[:OF]->(:Film {title:'...'}),
       (review)-[:PUBLISHED_IN]->(:Publication {title:'...'})
```

Представление комплексных типовых значений в виде узлов

Значением считается все, что не имеет идентичности и чья эквивалентность основывается исключительно на их величинах. Примерами значений могут служить *деньги* или *идентификаторы товарных*

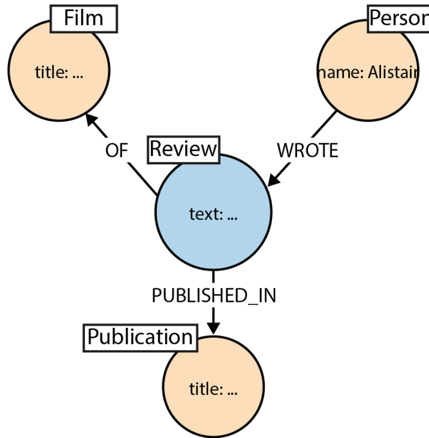


Рис. 4.5 ❖ Алистер написал рецензию на фильм, которая была опубликована в журнале

позиций (Stock Keeping Unit, аббревиатура SKU). Комплексными значениями считаются такие значения, для хранения которых требуется более одного поля или свойства. Примером комплексного значения может служить *адрес*. Значения с несколькими свойствами удобно представлять в виде отдельных узлов:

```
MATCH (:Order {orderid:13567})-[:DELIVERY_ADDRESS]->(address:Address)
RETURN address.first_line, address.zipcode
```

Время

Время в графе может быть представлено несколькими способами. Здесь будут описаны два из них: хронологическое древо и связный список. Иногда оба этих метода применяются в одном проекте.

Хронологическое древо

Если требуется извлечь все события, произошедшие в течение определенного периода, можно построить хронологическое древо, изображенное на рис. 4.6.

Каждый год (`Year`) имеет собственный набор узлов месяцев (`Month`). Каждый месяц имеет собственный набор узлов дней (`Day`). Нам нужно только добавить узлы в хронологическое древо, туда, где они необходимы. Предположим, что корневой узел хронологического древа был проиндексирован или может быть найден при обходе графа. Тогда

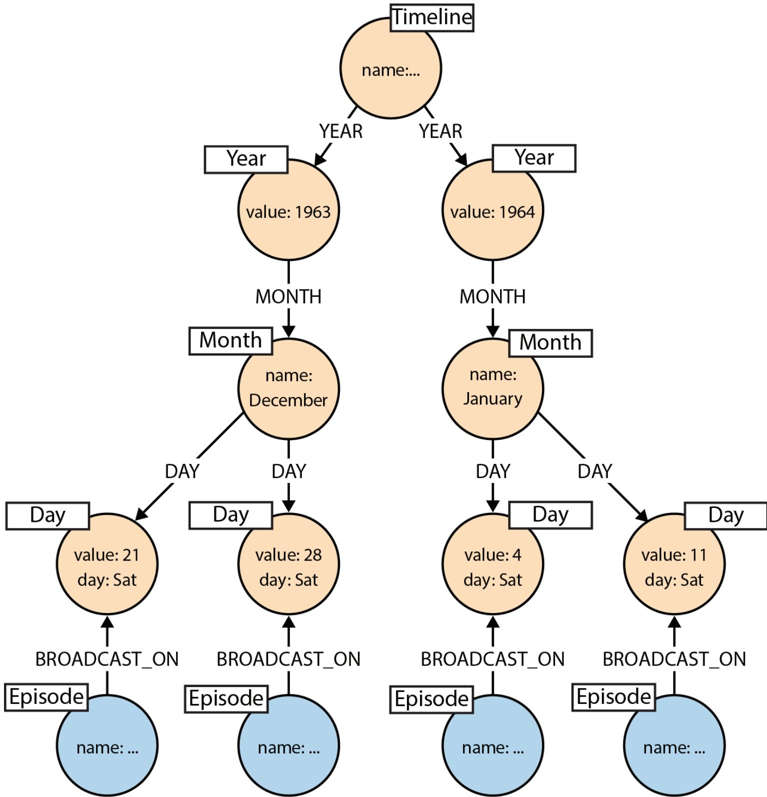


Рис. 4.6 ❖ Хронологическое древо, отображающие даты трансляции четырех передач телевизионной программы

можно утверждать, что выполнение следующего Cypher-оператора гарантирует, что все необходимые узлы и взаимосвязи для конкретного события – год (YEAR), месяц (MONTH), день (DAY) и узел, представляющий само событие (Episode), – либо уже присутствуют в графе, либо будут добавлены в граф (фраза MERGE позволяет добавлять только недостающие элементы):

```
MATCH (timeline:Timeline {name:{timelineName}})
MERGE (episode:Episode {name:{newEpisode}})
MERGE (timeline)-[:YEAR]->(year:Year {value:{year}})
MERGE (year)-[:MONTH]->(month:Month {name:{monthName}})
```

```
MERGE (month)-[:DAY]->(day:Day {value:{day}, name:{dayName}})
MERGE (day)<-[:BROADCAST_ON]-(episode)
```

Запрос календаря всех событий, происшедших между датой начала (включительно) и датой окончания (исключительно), на языке Cypher можно записать так:

```
MATCH (timeline:Timeline {name:{timelineName}})
MATCH (timeline)-[:YEAR]->(year:Year)-[:MONTH]->(month:Month)-[:DAY]->
      (day:Day)<-[:BROADCAST_ON]-(n)
WHERE ((year.value > {startYear} AND year.value < {endYear})
      OR ({startYear} = {endYear} AND {startMonth} = {endMonth}
        AND year.value = {startYear} AND month.value = {startMonth}
        AND day.value >= {startDay} AND day.value < {endDay})
      OR ({startYear} = {endYear} AND {startMonth} < {endMonth}
        AND year.value = {startYear}
        AND ((month.value = {startMonth} AND day.value >= {startDay})
          OR (month.value > {startMonth} AND month.value < {endMonth})
          OR (month.value = {endMonth} AND day.value < {endDay})))
      OR ({startYear} < {endYear}
        AND year.value = {startYear}
        AND ((month.value > {startMonth})
          OR (month.value = {startMonth} AND day.value >= {startDay})))
      OR ({startYear} < {endYear}
        AND year.value = {endYear}
        AND ((month.value < {endMonth})
          OR (month.value = {endMonth} AND day.value < {endDay}))))
RETURN n
```

Немного громоздкая фраза `WHERE` в этом запросе предназначена всего лишь для фильтрации полученных соответствий на основании дат начала и окончания, указанных в запросе.

Связанные списки

Многие события связаны по времени происхождения с другими событиями, которые либо предшествуют им, либо следуют за ними. Следовательно, можно воспользоваться взаимосвязями `NEXT` (следующее) и/или `PREVIOUS` (предыдущее) (в зависимости от предпочтений) для создания связанных списков, которые отражают их естественную хронологию, как показано на рис. 4.7[1]. Связанные списки позволяют очень быстро выполнять обход событий в хронологическом порядке.

[1] Двунаправленность списка является излишеством, потому что взаимосвязи всегда можно обойти в любом направлении.

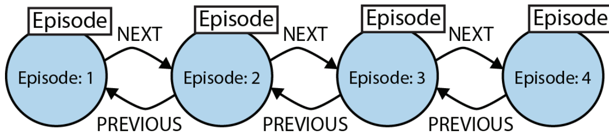


Рис. 4.7 ❖ Двунаправленный список, представляющий хронологию ряда событий

Управление версиями

Система управления версиями графа позволяет восстановить состояние графа в конкретный момент времени. Большинство графовых баз данных не имеет встроенной поддержки управления версиями как таковой. Однако всегда можно создать схему управления версиями *внутри* графовой модели. В такой схеме узлы и взаимосвязи снабжаются временными отметками и архивируются при модификации[1]. Недостатком таких схем управления версиями является их просачивание в тексты абсолютно всех запросов, что делает сложными даже простейшие запросы.

Итеративная и поэтапная разработка

Модель данных разрабатывается от одной функции к другой, от одной истории пользователя к другой. Это гарантирует тождественность взаимосвязей приложения со взаимосвязями, используемыми в запросах к графу. Модель данных, созданная в соответствии с принципами итеративной и поэтапной разработки приложений, будет весьма отличаться от модели данных, созданной изначально, но она будет правильной моделью, целиком соответствующей потребностям приложения и учитывающей вопросы, возникшие в связи с удовлетворением этих потребностей.

Графовые базы данных обеспечивают плавную эволюцию модели данных. Миграция и денормализация в них не становятся проблемами. Новые факты и новые связи представляются новыми узлами и взаимосвязями, но оптимизация шаблонов, где нужна высокая производительность, как правило, требует введения прямых взаимосвязей между двумя узлами, которые можно связать только через посредников. В отличие от стратегий оптимизации в реляционном мире, которые обычно основаны на денормализации и тем самым снижают

[1] Пример можно найти по адресу http://iansrobinson.com/2014/05/13/time-based-versioned-graphs/.

качество модели, здесь не стоит вопрос: либо подробная, высоконормализованная структура, либо высокая производительность. Граф позволяет сохранить оригинальную структуру и обогатить его новыми элементами, обслуживающими вновь возникшие потребности.

Рассмотрим, как взаимосвязи, предназначенные для удовлетворения различных нужд, могут сосуществовать, не искажая модели и не делая ее походящей только для одной какой-либо потребности. Проиллюстрируем это на примере адресов. Допустим, разрабатывается приложение для розничной торговли. Для удовлетворения пожеланий пользователей добавлена возможность отправки заказчику посылок по его адресу доставки, который можно получить с помощью следующего запроса:

```
MATCH (user:User {id:{userId}})
MATCH  (user)-[:DELIVERY_ADDRESS]->(address:Address)
RETURN address
```

Позже, при добавлении функции оформления счетов (биллинговых услуг), были введены взаимосвязи BILLING_ADDRESS. Затем клиентам была предоставлена возможность управлять своими адресами. Для реализации этой возможности потребовалось извлекать все адреса доставки, отправки счетов и любых других. Для облегчения этого введем обобщенные взаимосвязи ADDRESS:

```
MATCH (user:User {id:{userId}})
MATCH (user)-[:ADDRESS]->(address:Address)
RETURN address
```

На текущий момент модель данных выглядит, как показано на рис. 4.8. Взаимосвязь DELIVERY_ADDRESS (адрес доставки) предназначена для нужд доставки, взаимосвязь BILLING_ADDRESS (адрес доставки счетов) – для нужд выписки счетов покупателям, а взаимосвязь ADDRESS – для управленческих нужд клиентов приложения.

Наличие возможности добавления новых взаимосвязей не означает, что они обязательно должны добавляться. Нужно постоянно оценивать возможности для выполнения рефакторинга модели. Часто бывает достаточно простого переименования существующих взаимосвязей, чтобы обеспечить две разные потребности. Если такие возможности существуют, следует использовать именно их. Если разработка сопровождается постоянным тестированием, как описывается далее в этой главе, значит, имеется полный набор регрессионных тестов. Их наличие придает уверенности при внесении существенных изменений в модель.

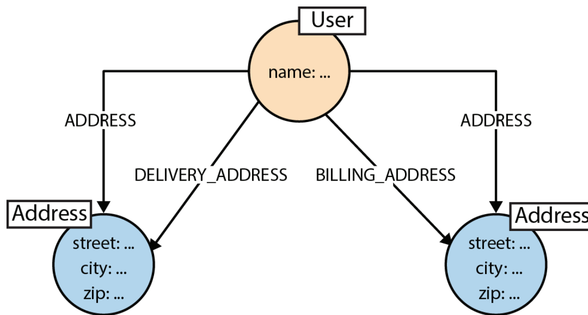


Рис. 4.8 ❖ Различные взаимосвязи для различных нужд приложения

Архитектура приложений

При разработке проектов, основанных на графовых базах данных, существует несколько вариантов построения архитектуры приложения. Эти варианты незначительно отличаются друг от друга в зависимости от выбранной базы данных. В этом разделе мы опишем некоторые из них и отдельно остановимся на вариантах, доступных при использовании Neo4j.

Встроенная поддержка или сервер

Большинство баз данных сегодня работает в качестве серверов, обращение к которым выполняется через клиентскую библиотеку. База данных Neo4j несколько необычна тем, что может выполняться как во встроенном, так и в серверном режимах. Если вернуться почти на десять лет назад, она изначально была встроенной графовой базой данных.

Встроенные базы данных далеко не то же самое, что базы данных для работы в оперативной памяти. Встроенный экземпляр Neo4j также сохраняет все данные на диске. Позже, в разделе «Тестирование», мы рассмотрим базу данных `ImpermanentGraphDatabase`, которая является версией базы данных Neo4j для работы в оперативной памяти и предназначена исключительно для тестирования.

Встроенная поддержка Neo4j

Во встроенном режиме Neo4j выполняется в одном процессе с приложением. Встроенный режим идеально подходит для устройств, настольных приложений и для внедрения в приложение собственных серверов. Вот некоторые из преимуществ встроенного режима:

- малое время отклика – приложение обращается к базе данных напрямую, а не через сеть;
- широкий выбор программных интерфейсов – имеется доступ к полному спектру интерфейсов для создания и выполнения запросов, доступны программный интерфейс Core API, фреймворк Traversal и язык запросов Cypher;
- управление транзакциями – использование программного интерфейса Core API позволяет управлять жизненным циклом транзакций, выполнять сколь угодно сложную последовательность команд базы данных в контексте одной транзакции. Программные интерфейсы Java API также предоставляют возможность вмешательства в жизненный цикл транзакций, позволяя подключать пользовательские обработчики событий в транзакциях (http://docs.neo4j.org/chunked/stable/transactions-events.html), выполняющие дополнительные действия.

Но при работе во встроенном режиме следует учитывать следующие моменты:

- возможность использования только виртуальной машины Java (Java virtual machine, сокращенно JVM) – база данных Neo4j основывается на JVM. То есть многие из ее интерфейсов доступны только из языков, поддерживаемых JVM;
- сборка мусора – при работе во встроенном режиме Neo4j является субъектом системы сборки мусора (Garbage Collection, GC) как хост-приложение. Длинные паузы между циклами сборки мусора могут приводить к увеличению времени выполнения запросов. Кроме того, при выполнении встроенного экземпляра как части отказоустойчивого кластера (High-Availability, HA) длительные паузы при сборке мусора могут вызвать выполнение кластерного протокола перевыборов ведущего узла;
- жизненный цикл базы данных – приложение отвечает за управление всем жизненным циклом базы данных, включая ее запуск и безопасное закрытие.

Базы данных Neo4j, работающие во встроенном режиме, можно объединять в кластеры для обеспечения высокой доступности и масштабировать точно так же, как базы данных в серверной версии. На самом деле можно запустить смешанный кластер встроенных и серверных экземпляров (кластеризация выполняется на уровне базы данных, а не на уровне сервера). Это типичный сценарий интеграции на уровне предприятия, где регулярные обновления от других систем

выполняются встроенными экземплярами, а затем копируются серверными экземплярами.

Серверный режим

В настоящее время наиболее распространен запуск базы данных Neo4j в серверном режиме. Каждый сервер основан на встроенном экземпляре базы данных Neo4j. Ниже перечислены некоторые из преимуществ работы в серверном режиме:

- доступность интерфейса REST API – сервер предоставляет расширенный программный интерфейс REST API, позволяющий клиентам отправлять запросы в формате JSON по протоколу HTTP. Ответы включают в себя документы в JSON-формате, содержащие гиперссылки, расширяющие функции набора данных. Интерфейс REST API поддерживает расширение конечными пользователями и выполнение запросов на Cypher;
- независимость от платформы – поскольку поддерживается доступ посредством документов в формате JSON, передаваемых по протоколу HTTP, сервер Neo4j доступен клиенту, работающему практически на любой платформе. Все, что требуется, – это клиентская библиотека HTTP[1];
- независимость от масштабирования – при работе базы данных Neo4j в режиме сервера кластер базы данных можно масштабировать независимо от кластера сервера приложений;
- изоляция от сборки мусора – в серверном режиме база данных Neo4j защищена от любых неблагоприятных влияний, связанных со сборкой мусора. Конечно, работа базы данных Neo4j приводит к образованию мусора, но ее влияние на сборку мусора тщательно контролируется, и она настроена так, чтобы смягчить любые серьезные побочные эффекты. Но база данных Neo4j обеспечивает использование серверных расширений, позволяя запускать на сервере произвольный Java-код (подробнее в разделе «Серверные расширения»), а выполнение серверных расширений может повлиять на сборку мусора сервера.

При использовании Neo4j в режиме сервера нужно иметь в виду следующее:

---

[1] Список удаленных клиентских библиотек Neo4j, разработанных сообществом, можно найти по адресу http://neo4j.com/developer/language-guides/.

- сетевые накладные расходы – существуют некоторые накладные расходы при каждом HTTP-запросе, хотя они довольно малы. После первого запроса TCP-соединение с клиентом остается открытым, пока не будет закрыто этим клиентом;
- состояние транзакций – транзакции сервера Neo4j основываются на Cypher-запросах. Это позволяет клиенту выполнить определенную серию Cypher-запросов внутри одной транзакции. Каждый запроса клиента расширяет транзакцию. Если клиент не подтвердит или не откатит транзакцию, состояние транзакции просуществует на сервере до истечения определенного времени (по умолчанию сервер исправляет осиротевшие транзакции через 60 секунд). Для реализации более сложных, многошаговых операций, требующих единой транзакции, следует использовать серверные расширения (подробнее в разделе «Серверные расширения»).

Обычно доступ к серверу Neo4j осуществляется посредством интерфейса REST API, упомянутого выше. REST API (http://docs.neo4j.org/chunked/stable/rest-api.html) включает возможность работы с документами в формате JSON через HTTP. С помощью REST API можно передавать Cypher-запросы, настраивать именованные индексы и выполнять встроенные графовые алгоритмы. Также имеется возможность описать в формате JSON несколько маршрутов и выполнить пакет операций. Обычно возможностей REST API вполне достаточно, но если потребуется что-то, недоступное через REST API, можно подумать о разработке серверного расширения.

Серверные расширения

Серверные расширения позволяют запускать Java-код на сервере. С помощью серверных расширений можно расширить интерфейс REST API или даже полностью сменить его.

Расширения имеют вид аннотированных классов JAX-RS. Класс JAX-RS (http://jax-rs-spec.java.net/) – это Java-интерфейс для создания RESTful-ресурсов. Используя аннотации JAX-RS, можно декорировать все классы расширения, чтобы подсказать серверу, какие HTTP-запросы должны обрабатываться этими классами. Дополнительные аннотации контролируют форматы запроса и ответа, HTTP-заголовки и форматирование шаблонов URI.

Ниже приводится реализация простого серверного расширения, позволяющая клиенту запросить расстояние между двумя членами социальной сети:

```
@Path("/distance")
public class SocialNetworkExtension
{
    private final GraphDatabaseService db;

    public SocialNetworkExtension( @Context GraphDatabaseService db )
    {
        this.db = db;
    }

    @GET
    @Produces("text/plain")
    @Path("/{name1}/{name2}")
    public String getDistance ( @PathParam("name1") String name1,
                                @PathParam("name2") String name2 )
    {
        String query = "MATCH (first:User {name:{name1}}),\n" +
                "(second:User {name:{name2}})\n" +
                "MATCH p=shortestPath(first-[*..4]-second)\n" +
                "RETURN length(p) AS depth";

        Map<String, Object> params = new HashMap<String, Object>();
        params.put( "name1", name1 );
        params.put( "name2", name2 );

        Result result = db.execute( query, params );

        return String.valueOf( result.columnAs( "depth" ).next() );
   }
}
```

Особый интерес здесь представляют различные аннотации:

- аннотация `@Path(«/distance»)` указывает, что это расширение будет отвечать на запросы, направленные к относительному URI, начинающемуся с */distance*;
- аннотация `@Path(«/{name1}/{name2}»)` перед функцией `getDistance()` уточняет шаблон URI, связанный с этим расширением. Указанный в аннотации фрагмент объединяется с */distance*, чтобы получить */distance/{name1}/{name2}*, где *{name1}* и *{name2}* – заполнители для любых символов, помещенных между косыми чертами. Далее, в разделе «Тестирование серверных расширений», мы зарегистрируем это расширение под относительным URI-адресом */socnet*. С этого момента эти несколько разных

частей адреса обеспечат направление HTTP-запросов по относительному URI-адресу, начинающемуся с */socnet/distance/{name1}/{name2}* (например, *http://localhost/socnet/distance/Ben/Mike*), и будут переданы экземпляру этого расширения;
- аннотация `@GET` указывает, что функция `getDistance()` должна вызываться, только если запрос является HTTP-запросом типа GET. Аннотация `@Produces` указывает, что тело ответа будет отформатировано как *text/plain*;
- две аннотации `@PathParam`, предваряющие параметры функции `getDistance()`, служат для присвоения содержимого заполнителей пути *{name1}* и *{name2}* параметрам метода `name1` и `name2`. С учетом URI-адреса *http://localhost/socnet/distance/Ben/Mike* функция `getDistance()` будет вызвана со значением *Ben* в параметре `name1` и значением *Mike* в параметре `name2`;
- аннотация `@Context`в конструкторе служит для передачи расширению ссылки на встроенную внутрь сервера графовую базу данных. Серверная инфраструктура позаботится о создании расширения и предоставлении ему экземпляра графовой базы данных, но само присутствие параметра `GraphDatabaseService` делает расширение удобным для тестирования. Далее в разделе «Тестирование серверных расширений» будет описано, как провести модульное тестирование расширений без их запуска на сервере.

Серверные расширения являются мощными элементами в архитектуре приложения. Их главные преимущества:
- сложные транзакции – расширения позволяют выполнять сколь угодно сложную последовательность операций в контексте единой транзакции;
- широкий выбор программных интерфейсов API – каждое расширение снабжено ссылкой на встроенную в сервер графовую базу данных. Это обеспечивает доступ при разработке расширений к полному спектру интерфейсов: Core API, Traversal Framework, графовым пакетным алгоритмам и Cypher;
- инкапсуляция – так как каждое расширение скрывается за RESTful-интерфейсом, всегда имеется возможность внести изменения и улучшения в его реализацию;
- форматы ответов – формирование ответов полностью контролируется разработчиком – не только формат представления, но и содержимое HTTP-заголовков. Это позволяет создавать ответные сообщения, основанные на терминологии прикладной области, а не графа (например, *пользователи*, *товары* и *заказы* вместо

узлов, взаимосвязей и свойств). Более того, контроль HTTP-заголовков, присоединенных к ответам, позволяет использовать HTTP-протокол для кэширования и условных запросов.

При использовании серверных расширений следует иметь в виду следующие моменты:

- возможность использования только JVM – так как разработка ведется на встроенном сервере Neo4j, необходимо использовать язык, доступный в JVM;
- сборка мусора – серверные расширения имеют возможность выполнять сколь угодно сложные (и опасные) действия. Необходимо отслеживать сборку мусора, чтобы гарантированно избегать появления неприятных побочных эффектов.

Кластеризация

Как будет подробно описано в разделе «Отказоустойчивость», для обеспечения высокой отказоустойчивости и горизонтального масштабирования кластеры Neo4j используют репликации вида главный-подчиненный (master-slave). В этом разделе будут рассмотрены некоторые из стратегий использования кластеризации Neo4j.

Репликация

Хотя все записи в кластер координируются главным сервером, Neo4j позволяет выполнять запись через подчиненные серверы, но только когда подчиненный сервер записывает данные для синхронизации с главным сервером перед их возвратом клиенту. Из-за дополнительных затрат на сетевые взаимодействия и на работу протокола координации запись через подчиненный сервер может быть на порядок медленнее, чем запись непосредственно через главный сервер. Единственными причинами для записи через подчиненный сервер являются увеличение надежности записи (запись в двух местах надежнее, чем в одном) и гарантия возможности чтения только что сделанной записи при использовании распределительного кэша (подробнее в разделах «Распределительный кэш» и «Чтение собственных записей» ниже). Поскольку новые версии Neo4j позволяют указывать, что запись на главном сервере должна повторяться на одном и более подчиненных серверах, чтобы увеличить надежность записи на главном сервере, для использования записи на подчиненном сервере не остается убедительных причин. В настоящее время рекомендуется направлять все записи на главный сервер, а затем выполнять их репликацию на подчиненные серверы, используя параметры

конфигурации `ha.tx_push_factor` и `ha.tx_push_strategy` (http://docs.neo4j.org/chunked/milestone/ha-configuration.html).

Запись буфера с помощью очередей

В ситуациях, когда требуется записать большие объемы данных, можно использовать очереди для буферизации записей и регулировать загрузку. При такой стратегии операции записи в кластер буферизируются и ставятся в очередь. Затем выполняются опрос очереди и пакетная запись в базу данных. Это позволяет не только регулировать трафик записи, но уменьшает конкуренцию и позволяет приостанавливать операции записи при обработке клиентских запросов в период технического обслуживания.

Глобальные кластеры

Для приложений, обслуживающих пользователей по всему миру, можно установить межрегиональный кластер в нескольких центрах обработки данных и облачных платформах, таких как Amazon Web Services (AWS). Межрегиональный кластер позволит производить чтение из географически ближайшей клиенту части кластера. Но в таких ситуациях задержка, вызванная физическим разделением областей, может привести к нарушению протокола координации. Поэтому желательно ограничить выбор главного сервера одним регионом. Чтобы добиться этого, создаются только подчиненные серверы базы данных, не участвующие в выборе главного сервера. Это делается с помощью установки параметра конфигурации экземпляра `ha.slave_coordinator_update_mode=none`.

Балансировка нагрузки

При использовании кластерной графовой базы данных следует предусмотреть балансировку нагрузки в кластере, чтобы оптимизировать пропускную способность и уменьшить задержки. База данных Neo4j не имеет встроенных инструментов балансировки, полагаясь на возможности инфраструктуры сети.

Отделение трафика чтения от трафика записи

Учитывая рекомендацию о направлении большей части трафика записи на главный сервер, следует предусмотреть четкое разделение запросов на чтение и запросов на запись. Необходимо так настроить балансировку нагрузки, чтобы направить трафик записи на главный сервер, а трафик чтения распределить по всему кластеру.

В веб-приложениях обычно бывает достаточно анализа HTTP-метода, чтобы отличить запросы на запись POST, PUT и DELETE, изменяющие ресурсы сервера со значительными побочными эффектами, от запросов на чтение GET, не имеющих побочных эффектов.

При использовании серверных расширений операции чтения и записи различаются по аннотациям `@GET` и `@POST`. Если приложение использует только серверные расширения, этого будет достаточно для разделения. Если же для выполнения Cypher-запросов к базе данных используется интерфейс REST API, ситуация усложняется. Интерфейс REST API использует только POST-запросы с обобщенной семантикой «обработай это» для Cypher-запросов на чтение и запись. В этом случае, чтобы отделить запросы на чтение и запись, можно применить пару балансировщиков нагрузки: балансировщик записи, всегда направляющий запросы на главный сервер, и балансировщик чтения, равномерно распределяющий запросы по всему кластеру. Решение о выборе одного из двух адресов в зависимости от вида операции должно приниматься для каждого запроса в самом приложении, как показано на рис. 4.9.

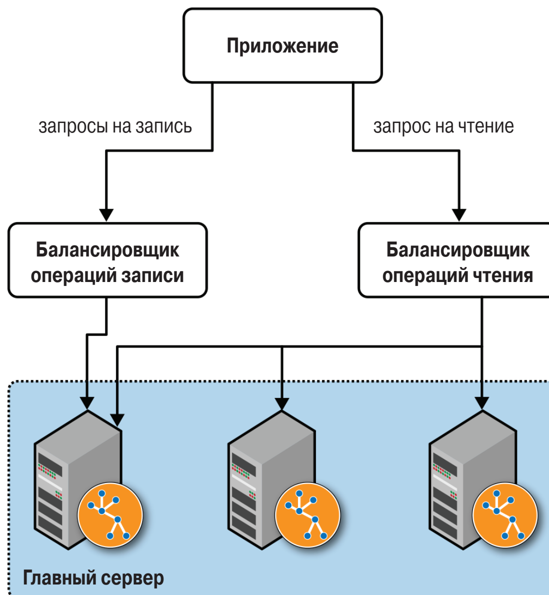


Рис. 4.9 ❖ Использование балансировщиков нагрузки для распределения запросов в кластере

При работе в режиме сервера Neo4j либо предоставляет URI-адрес, указывая, что этот экземпляр в настоящее время является главным сервером, а если это не так, то определяет, какой из экземпляров является главным сервером. Балансировщики нагрузки используют этот URI-адрес, чтобы определить, куда направить трафик.

Распределительный кэш

Запросы выполняются быстрее, если необходимые для их удовлетворения части графа находятся в основной памяти. Если граф имеет несколько миллиардов узлов, взаимосвязей и свойств, он не вмещается в основную память целиком. Другие технологии обработки данных обычно позволяют решить эту проблему путем разделения данных, но при разделении или распределении графов возникают большие проблемы (подробнее в разделе «Поиски чаши Грааля масштабируемости графов»). Как же тогда обеспечить высокую производительность запросов к огромному графу?

Одним из решений является использование технологии распределительного кэша (рис. 4.10), которая заключается в направлении каждого запроса экземпляру базы данных HA-кластера, в котором часть графа, необходимая для удовлетворения запроса, *наверное,* уже находится в основной памяти (вспомните, что каждый экземпляр в кластере содержит полную копию данных). Если большинство запросов приложения являются локальными применительно к графу, в том смысле, что они начинают работу с поиска одной или нескольких конкретных точек графа, а затем обходят окружающие их подграфы, тогда механизм, последовательно направляющий запросы, начинающиеся с одинакового набора начальных точек, в один и тот же экземпляр базы данных увеличит вероятность наличия подготовленного для запроса кэша.

Стратегия реализации последовательной маршрутизации будет варьироваться в зависимости от особенностей прикладной области. Иногда достаточно привязки к сеансу, в другом случае маршрутизация должна основываться на особенностях набора данных. Простейшей стратегией является изначальное прикрепление экземпляра к конкретному пользователю с дальнейшей посылкой запросов этого пользователя только этому экземпляру. Также работают подходы, ориентированные на особенности прикладной области. Например, в географической системе данных можно направлять запросы о конкретных местностях конкретным экземплярам базы данных, уже готовым для работы с этим местоположением. Обе стратегии направлены

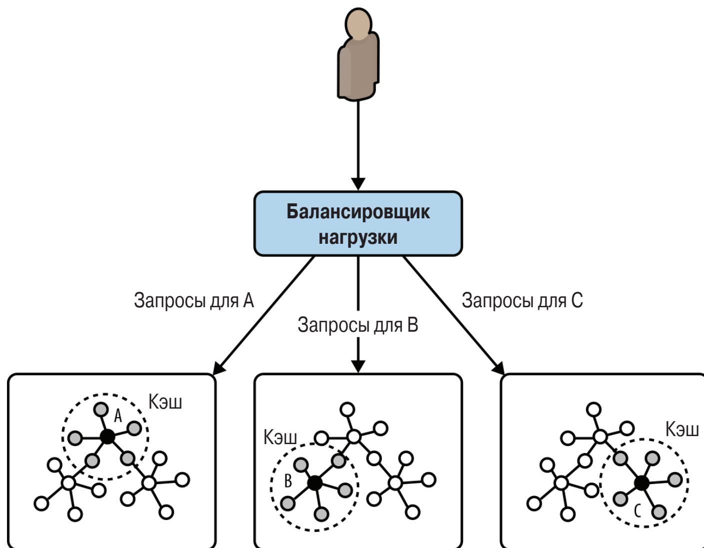


Рис. 4.10 ❖ Распределительный кэш

на увеличение вероятности предварительного кэширования необходимых узлов и взаимоотношений в оперативной памяти, откуда они могут быть быстро извлечены и обработаны.

Чтение собственных записей

Иногда приложению требуется выполнить чтение данных, ранее записанных им самим. Обычно такая необходимость возникает при применении изменений, внесенных конечным пользователем, и связана с последующим запросом для извлечения результатов, вызванных этими изменениями. Так как запись производится непосредственно на главном сервере, в конечном итоге кластер придет к согласованности. Но как гарантировать, что запись на главном сервере будет учтена в следующем запросе на чтение при наличии балансировки нагрузки? Одним из решений является использование той же техники последовательной маршрутизации, которая применяется при распределительном кэшировании, т. е. адресовать запись подчиненному серверу, который будет использоваться для обслуживания следующего чтения. Это предполагает последовательную маршрутизацию запросов на запись и чтение на основании некоторых критериев прикладной области.

Это один из тех немногих случаев, когда имеет смысл производить запись на подчиненном сервере. Однако вспомним, что запись через подчиненный сервер может выполняться на порядок медленнее, чем запись непосредственно на главном сервере. Эту технологию следует использовать экономно. Если значительная часть операций записи потребует чтения этих же записей, применение такого метода значительно повлияет на пропускную способность и задержки.

Тестирование

Тестирование является фундаментальной частью процесса разработки приложений, не только средством проверки корректности выполнения запросов или функций приложения, но и способом проектирования и документирования приложения и его модели данных. В этом разделе нелишним будет подчеркнуть, что тестирование является каждодневной заботой. Разработка решения для графовой базы данных, основанная на тестировании, позволит в дальнейшем быстро развивать систему и оперативно реагировать на новые требования к ней.

Разработка модели данных, основанная на тестировании

Рассматривая моделирование данных, мы сделали упор на соответствие графовой модели запросам, которые будут к ней применены. Разрабатывая модели данных через тестирование, мы документируем понимание прикладной области и проверяем правильность поведения запросов.

При моделировании данных через тестирование пишутся модульные тесты, основанные на небольших типовых примерах графов, полученных из прикладной области. Эти примеры содержат лишь столько данных, сколько будет достаточно для выражения особенностей прикладной области. Часто они содержат лишь с десяток узлов плюс соединяющие их взаимосвязи. Такие примеры используются, чтобы описать, что считается нормальным для прикладной области, а что – исключительным. По мере выявления аномалий и частных случаев в реальных данных мы пишем тесты, воспроизводящие обнаруженные особенности.

Примеры графов, создаваемые для каждого теста, включают в себя настройки, или контекст теста. В этом контексте выполняется запрос, подтверждающий ожидаемые результаты. Так как содержание исход-

ных данных для тестов контролируется, автор теста заранее знает, каких результатов следует ждать.

Тесты могут служить документацией. Знакомясь с тестами, разработчики осознают проблемы и нерешенные задачи в приложении и определяют пути их решения. Таким образом, желательно, чтобы тест был предназначен только для одного аспекта прикладной области. Гораздо проще разобраться в ряде коротких и простых тестов, каждый из которых связан с отдельной особенностью данных, чем в одном большом громоздком тесте, охватывающем всю сложную прикладную область. Во многих случаях конкретный запрос можно проверить с помощью нескольких тестов, одни из которых основаны на типичных для прикладной области данных, а другие – наоборот, на какой-то исключительной структуре или на очень редко встречающемся наборе значений[1].

Со временем накапливается набор тестов, который может быть использован в качестве мощного инструмента регрессионного тестирования. В процессе развития приложения, для удовлетворения новых потребностей, в него добавляются новые источники данных или вносятся изменения в модель, а регрессионное тестирование позволяет подтвердить, что существующие функции по-прежнему работают должным образом. Эволюция архитектуры и поэтапные интерактивные методы разработки программного обеспечения, которые ее поддерживают, требуют для своего применения надежной базы. Модульное тестирование при разработке модели данных, описанное здесь, позволяет разработчикам реагировать на новые требования прикладной области, практически не рискуя разрушить сделанное ранее, и придает им уверенности, что качество решения по-прежнему соответствует нужному уровню.

Пример: модель данных социальной сети и ее тестирование

Этот пример демонстрирует разработку простейшего Cypher-запроса для социальной сети. Получив имена пары членов сети, запрос определяет расстояние между ними.

---

[1] Тесты не только могут служить документацией, их также можно использовать для генерации документации. Вся документация с описанием языка Cypher из руководства Neo4j (http://docs.neo4j.org/chunked/stable/cypher-query-lang.html) сгенерирована автоматически с помощью модульных тестов, применяемых при разработке языка запросов Cypher.

Для начала создадим небольшой граф, представляющий прикладную область. С помощью языка Cypher создадим сеть, состоящую из 10 узлов и 8 взаимосвязей:

```
public GraphDatabaseService createDatabase()
{
    // Создать узлы
    String createGraph = "CREATE\n" + "(ben:User {name:'Ben'}),\n" +
    "(arnold:User {name:'Arnold'}),\n" +
    "(charlie:User {name:'Charlie'}),\n" +
    "(gordon:User {name:'Gordon'}),\n" +
    "(lucy:User {name:'Lucy'}),\n" +
    "(emily:User {name:'Emily'}),\n" +
    "(sarah:User {name:'Sarah'}),\n" +
    "(kate:User {name:'Kate'}),\n" +
    "(mike:User {name:'Mike'}),\n" +
    "(paula:User {name:'Paula'}),\n" +
    "(ben)-[:FRIEND]->(charlie),\n" +
    "(charlie)-[:FRIEND]->(lucy),\n" +
    "(lucy)-[:FRIEND]->(sarah),\n" +
    "(sarah)-[:FRIEND]->(mike),\n" +
    "(arnold)-[:FRIEND]->(gordon),\n" +
    "(gordon)-[:FRIEND]->(emily),\n" +
    "(emily)-[:FRIEND]->(kate),\n" +
    "(kate)-[:FRIEND]->(paula)";
    String createIndex = "CREATE INDEX ON :User(name)";

    GraphDatabaseService db =
        new TestGraphDatabaseFactory().newImpermanentDatabase();

    db.execute( createGraph );
    db.execute( createIndex );

    return db;
}
```

В функции `createDatabase()` особый интерес представляют два момента. Во-первых, использование версии `ImpermanentGraphDatabase` – облегченного варианта Neo4j для работы в оперативной памяти, специально предназначенного для модульного тестирования. Это позволяет избежать сохранения файлов на диске после завершения теста. Класс находится в архиве `neo4j-kernel tests.jar`, который подключается с помощью следующей ссылки на зависимости:

```
<dependency>
    <groupId>org.neo4j</groupId>
    <artifactId>neo4j-kernel</artifactId>
    <version>${project.version}</version>
    <type>test-jar</type>
    <scope>test</scope>
</dependency>
```

Класс `ImpermanentGraphDatabase` предназначен для использования исключительно в модульных тестах. Эта версия Neo4j действует только в оперативной памяти и не предназначена для обычной эксплуатации.

Во-вторых, особый интерес в функции `createDatabase()` представляет Cipher-команда индексации узлов с заданной меткой по заданному свойству. В данном случае она осуществляет индексацию узлов с меткой `:User` по значениям их свойств `name`.

После создания примера графа можно переходить к написанию первого теста. Следующий тест проверяет модель данных социальной сети и результаты запросов к ней:

```
public class SocialNetworkTest
{
    private static GraphDatabaseService db;
    private static SocialNetworkQueries queries;

    @BeforeClass
    public static void init()
    {
        db = createDatabase();
        queries = new SocialNetworkQueries( db );
    }
    @AfterClass
    public static void shutdown()
    {
        db.shutdown();
    }

    @Test
    public void shouldReturnShortestPathBetweenTwoFriends() throws Exception
    {
        // когда
        Result result = queries.distance( "Ben", "Mike" );

        // то
        assertTrue( result.hasNext() );
```

```
        assertEquals( 4, result.next().get( "distance" ) );
    }

    // прочие тесты
}
```

Этот тест включает метод инициализации с аннотацией `@BeforeClass`, который выполняется перед началом тестирования. В нем вызывается метод `createDatabase()` для создания экземпляра графа и экземпляра `SocialNetworkQueries`, в котором будут находиться разрабатываемые запросы.

Первый тест `shouldReturnShortestPathBetweenTwoFriends()` проверяет, способен ли разрабатываемый запрос найти расстояние между любыми двумя членами сети, в данном случае между Беном (`Ben`) и Майком (`Mike`). Из содержимого графа можно узнать, что Бен и Майк связаны, но весьма отдаленно, и расстояние между ними равно 4. Поэтому тест должен подтвердить, что запрос возвратит не пустой результат, а результат, содержащий значение `distance`, равное 4.

Написав тест, приступим к разработке первого запроса. Следующий класс `SocialNetworkQueries` содержит его реализацию:

```
public class SocialNetworkQueries
{
    private final GraphDatabaseService db;

    public SocialNetworkQueries( GraphDatabaseService db )
    {
        this.db = db;
    }

    public Result distance( String firstUser, String secondUser )
    {
        String query = "MATCH (first:User {name:{firstUser}}),\n" +
            "(second:User {name:{secondUser}})\n" +
            "MATCH p=shortestPath((first)-[*..4]-(second))\n" +
            "RETURN length(p) AS distance";

        Map<String, Object> params = new HashMap<String, Object>();
        params.put( "firstUser", firstUser );
        params.put( "secondUser",  secondUser );

        return db.execute( query, params );
    }

    // Другие запросы
}
```

В конструкторе класса `SocialNetworkQueries` экземпляр поддерживаемой базы данных сохраняется во внутренней переменной, что позволяет его повторное использование и после уничтожения экземпляра `queries`. Сам запрос реализован в методе `distance()`. Здесь создается Cypher-оператор, инициализируется экземпляр `Map` с параметрами запроса и выполняется Cypher-оператор.

Если вызов `shouldReturnShortestPathBetweenTwoFriends()` завершится успешно (а это так), можно переходить к дополнительным проверкам. Например, что произойдет, если два члена сети разделены более чем четырьмя взаимосвязями? Проверим этот случай, написав другой тест:

```
@Test
public void shouldReturnNoResultsWhenNoPathAtDistance4OrLess()
    throws Exception
{
    // когда
    Result result = queries.distance( "Ben", "Arnold" );

    // то
    assertFalse( result.hasNext() );
}
```

Второй тест не потребовал вносить изменений в основной Cypher-запрос. Но чаще новый тест требует внесения изменений в реализацию запроса. Когда это происходит, следует изменить запрос, чтобы удовлетворить потребности нового теста, а затем запустить все тесты в наборе. Неудача любого из тестов в наборе выявит нарушения в работе одной или нескольких уже существующих функций. Запрос следует изменить так, чтобы все тесты выполнялись успешно.

Тестирование серверных расширений

При разработке серверных расширений можно использовать те же методы тестирования, что и для встроенного сервера Neo4j. Протестируем описанное выше простое серверное расширение с помощью следующего теста:

```
@Test
public void extensionShouldReturnDistance() throws Exception
{
    // создание
    SocialNetworkExtension extension = new SocialNetworkExtension( db );

    // когда
```

```
    String distance = extension.getDistance( "Ben", "Mike" );

    // то
    assertEquals( "4", distance );
}
```

Так как конструктор расширения принимает в качестве параметра экземпляр `GraphDatabaseService`, ему можно передать тестовый экземпляр (экземпляр `ImpermanentGraphDatabase`), а затем вызывать его методы как методы любого другого объекта.

Но чтобы проверить, как расширение работает на сервере, потребуется несколько больше настроек:

```
public class SocialNetworkExtensionTest
{
private ServerControls server;

    @BeforeClass
    public static void init() throws IOException
    {
    // Создать узлы
    String createGraph = "CREATE\n" + "(ben:User {name:'Ben'}),\n" +
        "(arnold:User {name:'Arnold'}),\n" +
        "(charlie:User {name:'Charlie'}),\n" +
        "(gordon:User {name:'Gordon'}),\n" +
        "(lucy:User {name:'Lucy'}),\n" +
        "(emily:User {name:'Emily'}),\n" +
        "(sarah:User {name:'Sarah'}),\n" +
        "(kate:User {name:'Kate'}),\n" +
        "(mike:User {name:'Mike'}),\n" +
        "(paula:User {name:'Paula'}),\n" +
        "(ben)-[:FRIEND]->(charlie),\n" +
        "(charlie)-[:FRIEND]->(lucy),\n" +
        "(lucy)-[:FRIEND]->(sarah),\n" +
        "(sarah)-[:FRIEND]->(mike),\n" +
        "(arnold)-[:FRIEND]->(gordon),\n" +
        "(gordon)-[:FRIEND]->(emily),\n" +
        "(emily)-[:FRIEND]->(kate),\n" +
        "(kate)-[:FRIEND]->(paula)";

   server = TestServerBuilders
       .newInProcessBuilder()
       .withExtension(
           "/socnet",
           ColleagueFinderExtension.class )
```

```
            .withFixture( createGraph )
            .newServer();
    }

    @AfterClass
    public static void teardown()
    {
        server.close();
    }

    @Test
    public void serverShouldReturnDistance() throws Exception
    {
        HTTP.Response response = HTTP.GET( server.httpURI()
            .resolve( "/socnet/distance/Ben/Mike" ).toString() );

        assertEquals( 200, response.status() );
        assertEquals( "text/plain", response.header( "Content-Type" ));
        assertEquals( "4", response.rawContent( ) );
    }
}
```

Здесь для размещения расширения используется экземпляр `ServerControls`. С помощью встроенного класса `TestServerBuilders` в методе `init()`создается сервер и заполняется его база данных. Этот класс позволяет зарегистрировать расширение и связать его с относительным URI-адресом (в данном случае начинающимся с */socnet*). После завершения метода `init()` будет создан и запущен экземпляр сервера базы данных.

В самом тесте `serverShouldReturnDistance()` для доступа к серверу используется HTTP-клиент из библиотеки тестирования Neo4j. Клиент выполняет HTTP-запрос GET к ресурсу */socnet/distance/Ben/Mike*. (На стороне сервера этот запрос передается экземпляру расширения `SocialNetworkExtension`.) Когда клиент получит ответ, тест проверит получение правильного кода состояния HTTP, тип содержимого и собственно содержимое ответа.

Тестирование производительности

Описанное выше тестирование было связано с контекстом и верным отражением прикладной области, т. е. все это были тесты на правильность. Но эти тесты никак не были связаны с производительностью. То, что работает быстро на небольшом примере графа из 20 узлов, может работать очень медленно на гораздо большем графе. Поэтому,

кроме модульных тестов, нужны еще тесты производительности запросов. Учитывая это, с самого начала разработки приложения нужно заниматься тщательным тестированием производительности приложения.

Тесты производительности запросов

Тесты производительности запросов отличаются от полноценных тестов производительности приложений. Все, что нас интересует на этом этапе, – это выполнение конкретного запроса на графе примерно такого же размера, как размер графа, с которым предполагается столкнуться в производственной среде. В идеале эти тесты разрабатываются одновременно с модульными тестами. Нет ничего хуже, чем потратить массу времени на совершенствование запроса, а потом обнаружить, что он не подходит для размера данных, необходимого для промышленной эксплуатации.

При создании тестов производительности следует иметь в виду следующие рекомендации:

- создавайте набор тестов производительности, разработанных с помощью модульного тестирования. Запишите количественные показатели производительности, чтобы оценить эффект от настроек запроса, от изменения размера множества или при переходе от одной версии графовой базы данных к другой;
- запускайте эти тесты часто, чтобы своевременно выявлять любые ухудшения производительности. Можно продумать включение этих тестов в процесс пакетной сборки, прерывая сборку, если результаты тестов превысят заданные значения;
- выполняйте тесты в одном потоке. На данном этапе нет необходимости имитировать одновременную работу нескольких клиентов, поскольку если производительность недостаточно высока для одного клиента, она вряд ли улучшится при нескольких клиентах. Хотя, строго говоря, эти тесты не являются модульными, но для них можно использовать тот же фреймворк модульного тестирования, который был использован при разработке модульных тестов;
- запускайте каждый запрос несколько раз, выбирая при этом начальные узлы наугад, чтобы увидеть эффект запуска на пустом кэше, который постепенно заполняется при выполнении нескольких запросов.

Тесты производительности приложений

Тесты производительности приложений, в отличие от тестов производительности запросов, предназначены для тестирования производительности всего приложения в условиях, приближенных к типичным условиям промышленной эксплуатации.

Так же как при тестировании производительности запросов, этот вид тестирования рекомендуется проводить параллельно с разработкой, по мере создания приложения, а не выводить в отдельный этап работы над проектом[1]. Чтобы облегчить тестирование производительности приложений в начале жизненного цикла проекта, часто бывает необходимо создать «действующий макет» с фрагментами всей системы, с помощью которого можно выполнить тестирование производительности для клиентов. Созданный макет поможет не только в тестировании производительности, но и для настройки частичного контекста архитектуры графовой базы данных решения. Это позволит проверить архитектуру приложения, а также определить слои и абстракции, позволяющие проводить дискретное тестирование отдельных компонентов.

Тесты производительности служат двум целям, демонстрируют, как система будет работать во время эксплуатации, и исключают оперативные допуски, что упрощает диагностику проблем производительности, некорректного поведения и ошибок. Все, что удастся узнать о производительности в тестовой среде, станет бесценным, когда дело дойдет до развертывания и эксплуатации системы.

При выборе критериев теста производительности мы рекомендуем использовать количественные, а не средние показатели. Никогда нельзя предугадать нормальное распределение времени отклика: реальный мир так не работает. Для некоторых приложений требуется, чтобы *все* запросы выполнялись в течение заданного периода времени. Редко когда бывает важна скорость выполнения *самого первого* запроса, когда кэши еще пусты. Чаще нужно гарантировать выполнение большинства запросов в течение определенного периода времени, например выполнение 98% запросов в течение 200 мсек. Важно сохранить записи о последовательных тестовых прогонах, чтобы можно было сравнить показатели производительности в разные моменты времени и выявить замедление или аномальное поведение.

---

[1] Прекрасное описание тестирования производительности можно найти в эссе Алистера Джонса (Alistair Jones) и Патрика Куа (Patrick Kua) «Extreme Performance Testing» в *The ThoughtWorks Anthology, том 2* (Pragmatic Bookshelf, 2012).

Так же как модульные тесты и тесты производительности запросов, тесты производительности приложений наиболее ценны при использовании их в процессе автоматизированного развертывания в среде тестирования, тесты будут выполнены, и результаты автоматически проанализированы. Регистрационные файлы и результаты тестов должны сохраняться для последующего просмотра, анализа и сравнения. Регрессы и неудачи должны прерывать сборку, подталкивая разработчиков к своевременному решению проблем. Наибольшим преимуществом тестирования производительности в течение всего цикла разработки приложения, а не в его конце, является возможность связать неудачи и регрессы с последним периодом разработки. Это позволит быстрее и успешнее диагностировать, выявлять и устранять проблемы.

Для нагрузочного тестирования необходим агент, генерирующий нагрузку. Для веб-приложений существует несколько открытых инструментов стресс-тестирования и тестирования под нагрузкой, в том числе Grinder (http://grinder.sourceforge.net/), JMeter (http://jmeter.apache.org/) и Gatling (http://gatling-tool.org/)[1]. При тестировании веб-приложений с балансировкой нагрузки следует распределить тестовых клиентов по разным IP-адресам для балансировки запросов в кластере.

Тестирование с помощью репрезентативных данных

Для тестирования производительности запросов и приложений нужен набор данных, имитирующий данные, с которыми приложение столкнется во время эксплуатации. Следовательно, необходимо создать или где-то получить такой набор данных. Иногда набор данных можно получить от третьей стороны, иногда – адаптировать имеющийся набор данных, но в любом случае, если данные не представлены в виде графа, нужно написать код для импорта.

Однако часто приходится начинать с нуля. В этом случае нужно выделить некоторое время на создание генератора набора данных. Как и прочие составляющие процесса разработки программного обеспечения, создавать такой генератор лучше всего с помощью поэтапного и итеративного подхода. Вводя новый элемент в модель данных прикладной области, как это было описано для модульных тестов, следует добавить соответствующий элемент в генератор на-

---

[1] Макс Де Марзи (Max De Marzi) описывает пользование инструмента Gatling для тестирования на Neo4j.

бора данных для тестирования производительности. Таким образом, тесты производительности будут приближены к реальным требованиям промышленной эксплуатации, насколько это позволит текущее понимание прикладной области.

При создании репрезентативных наборов данных желательно определить все выявленные инварианты прикладной области: минимальное, максимальное и среднее количество взаимосвязей для узлов, распространенность различных видов взаимосвязей, диапазоны значений свойств и т. д. Конечно, не всегда можно все это определить заранее, и часто работа ведется по приблизительным оценкам, пока данные, накопленные во время эксплуатации, не позволят проверить сделанные предположения.

В идеале тестирование всегда нужно проводить на наборе данных, имеющем тот же размер, что и набор данных при производственной эксплуатации, но часто невозможно или нежелательно воспроизводить огромные объемы данных в среде тестирования. В таких случаях следует хотя бы удостовериться, что размер сгенерированного набора данных не дает возможности разместить его целиком в оперативной памяти. При этом можно будет наблюдать эффект вытеснения кэша и извлекаемая часть графа не будет находиться в настоящий момент в оперативной памяти.

Репрезентативные наборы данных пригодятся и при планировании мощности. Создается либо полноразмерный набор данных ожидаемого при промышленной эксплуатации объема, либо уменьшенная его копия для оценки возможности размещения на диске данных при промышленной эксплуатации. Эти показатели в дальнейшем пригодятся при принятии решения о количестве оперативной памяти, выделяемой под кэши и под кучу виртуальной Java-машины (JVM) (подробнее в разделе «Планирование производственных мощностей»).

В следующем примере с помощью генератора набора данных Neode (https://github.com/iansrobinson/neode) создается пример набора данных для социальной сети:

```
private void createSampleDataset( GraphDatabaseService db )
{
    DatasetManager dsm = new DatasetManager( db, SysOutLog.INSTANCE );

    // Определение узла User
    NodeSpecification userSpec =
        dsm.nodeSpecification( "User",
```

```
            indexableProperty( db, "User", "name" ) );

    // Определение взаимосвязи FRIEND
    RelationshipSpecification friend =
        dsm.relationshipSpecification( "FRIEND" );

    Dataset dataset =
        dsm.newDataset( "Social network example" );

    // Создание узлов пользователей
    NodeCollection users =
        userSpec.create( 1_000_000 ).update( dataset );


    // Связывание пользователей между собой
    users.createRelationshipsTo( getExisting( users )
        .numberOfTargetNodes( minMax( 50, 100 ) )
        .relationship( friend )
        .relationshipConstraints( RelationshipUniqueness.BOTH_DIRECTIONS ) )
    .updateNoReturn( dataset );

    dataset.end();
}
```

Генератор набора данных Neode использует определения узлов и взаимосвязей для их отображения в графе вместе с их свойствами и допустимыми значениями свойств. Neode имеет весьма гибкий интерфейс для создания и связывания узлов.

Планирование производственных мощностей

В какой-то момент в цикле разработки приложения нужно начать планирование его развертывания в промышленной среде. Во многих случаях организационная часть управления проектом не может обойтись без определенного понимания требований приложения во время эксплуатации. Планирование производственных мощностей важно как для бюджетных целей, так и для обеспечения достаточного времени для закупки основного оборудования и резервирования производственных ресурсов.

В этом разделе описаны методы подбора аппаратных средств и планирования производственных мощностей. Возможность оценки производственных нужд зависит от ряда факторов. Чем больше собрано

данных о размере графа, производительности запросов и ожидаемом числе пользователей и их поведении, тем больше возможностей оценки потребностей в оборудовании. Большую часть необходимой информации на раннем этапе разработки приложения можно получить с помощью методов, описанных в разделе «Тестирование». Кроме того, надо понимать, что компромиссы стоимости и производительности следует рассматривать в контексте потребностей бизнеса.

Критерии оптимизации

При проектировании производственной среды можно использовать несколько вариантов оптимизации. Какому из них отдать предпочтение, будет зависеть от потребностей бизнеса:

- стоимость – можно оптимизировать стоимость, выбрав минимальную аппаратную конфигурацию, необходимую для работы приложения;
- производительность – можно оптимизировать производительность, приобретя самые быстрые аппаратные средства (с учетом бюджетных ограничений);
- избыточность – можно оптимизировать резервирование и отказоустойчивость путем создания кластера базы данных достаточно большого размера, чтобы обеспечить выживание при выходе из строя определенного количества компьютеров (например, чтобы продолжить работу при выходе из строя двух компьютеров, нам понадобится кластер, состоящий из пяти компьютеров);
- нагрузка – при реализации решения графовой базы данных с репликацией можно оптимизировать нагрузку с помощью горизонтального масштабирования (нагрузка при чтении) и вертикального масштабирования (нагрузка при записи).

Производительность

Избыточность и нагрузку можно оценить количеством компьютеров, необходимым для достижения определенного уровня отказоустойчивости (например, пять компьютеров для поддержания работоспособности при выходе из строя двух из них) и масштабируемости (один компьютер для обслуживания некоторого количества одновременных запросов, исходя из расчетов, приведенных в разделе «Нагрузка» ниже). Но что можно сказать о производительности? Как можно оценить производительность?

Калькуляция затрат на увеличение производительности графовой базы данных

Для оценки финансовых затрат на оптимизацию производительности нужно разобраться в ряде эксплуатационных характеристик базы данных. Как будет подробно описано в разделе «Нативное хранилище графов» ниже, графовые базы данных используют дисковые накопители для длительного хранения и оперативную память для кэширования отдельных частей графа.

Жесткие диски дешевы, но медленно выполняют произвольное позиционирование (около 6 мсек для современного диска). Запросы, полностью выбирающие данные с жесткого диска, выполняются на порядок медленнее, чем запросы, получающие данные только из оперативной памяти. Время доступа к диску можно улучшить путем замены жестких дисков твердотельными (Solid-State Drives, SSD), это даст приблизительно 20-кратное увеличение производительности, или промышленными флеш-накопителями, применение которых уменьшит задержку еще больше.

Для приложений, в которых размер графовых данных значительно превышает объем доступной оперативной памяти (и, следовательно, объем кэша), SSD-накопители являются отличным выбором, потому что не содержат механических частей и свободны от связанных с их использованием временных затрат, характерных для жестких дисков.

Варианты оптимизации производительности

Существуют три пути оптимизации производительности:

- увеличение размера кучи виртуальной машины Java;
- увеличение соотношения объема кэшированных данных к общему объему хранимых данных;
- применение быстрых дисковых накопителей: SSD или промышленных флеш-накопителей.

Как демонстрирует график на рис. 4.11, лучший баланс производительности и стоимости находится в точке, где файлы хранилища могут быть полностью размещены в кэше, при сохранении скромного, но достаточного размера памяти, выделяемого под кучу. Размер кучи от 4 до 8 ГБ не редкость, но во многих случаях уменьшение размера кучи ведет к улучшению производительности (это связано с сокращением затрат на сборку мусора).

Расчет объема оперативной памяти, выделяемого под кучу и кэш, зависит от знания точного размера проектируемого графа. Оценить этот

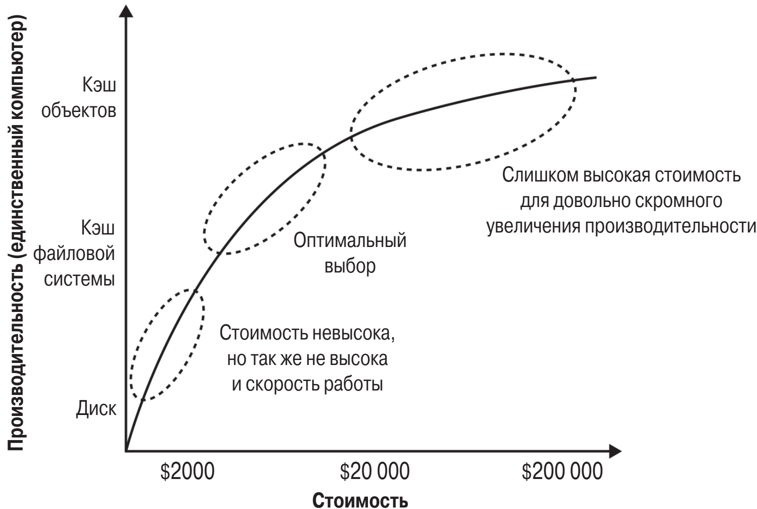


Рис. 4.11 ❖ Баланс между производительностью и стоимостью

размер на начальном этапе разработки поможет генерация репрезентативных данных. Если нет возможности целиком разместить граф в оперативной памяти, следует подумать об использовании распределительного кэша (подробнее в разделе «Распределительный кэш» выше).

Более подробную информацию о производительности и советы, связанные с ее настройкой, можно найти на *http://neo4j.com/docs/stable/configuration.html*.

При оптимизации производительности решения на основе графовой базы данных следует принимать во внимание следующие рекомендации:

- нужно как можно больше увеличить объем кэша файловой системы и по возможности постараться *практически целиком* отобразить файлы хранилища в кэш;
- при настройке кучи JVM следует увеличивать ее размер постепенно, контролируя сборку мусора, пока не будет обнаружена точка резкого увеличения затрат на сборку мусора;
- для повышения производительности в начале работы, когда обращения к дисковым накопителям не избежать, следует подумать об использовании быстрых SSD-дисков или промышленных флеш-накопителей.

Избыточность

При планировании избыточности требуется определить, при потере скольких экземпляров серверов в кластере приложение сохранит работоспособность. Для приложений, не имеющих критической важности, это количество может быть равно единице (или даже нулю). То есть после выхода из строя одного сервера выход из строя следующего сделает приложение недоступным. Но для критически важных приложений это число должно быть не меньше двух, т. е. даже после того, как отказали два сервера, приложение должно продолжать обслуживать запросы.

Для графовой базы данных, протокол управления кластером которой требует для нормальной работы доступности большинства серверов, избыточность в один сервер потребует от трех до четырех серверов, а избыточность в два сервера – не менее пяти. В этом отношении наличие четырех серверов ничем не лучше наличия трех, так как при выходе из строя двух из четырех серверов кластера оставшиеся два не смогут выбрать, кто из них главный.

Нагрузка

Оптимизация нагрузки является, наверное, самым сложным элементом планирования производственных мощностей. Ниже приведена эмпирическая формула:

Количество одновременных обращений = (1000 / *Среднее время обработки обращения* (*в миллисекундах*)) * *Количество ядер сервера* * *Количество серверов*.

Но иногда определить или спрогнозировать переменные, фигурирующие в этой формуле, достаточно сложно:

- *Среднее время обработки обращения* – время обработки обращения – это длительность интервала от момента получения сервером обращения до момента отсылки им ответа. В определении среднего времени обработки обращения помогут тесты производительности, если тесты выполняются на репрезентативных аппаратных средствах и используются репрезентативные данные (если это не так, лучше перестраховаться). Во многих случаях «репрезентативный набор данных» сам по себе основан лишь на грубых прикидках, по мере уточнения оценок следует изменять и фигуранты формулы.
- *Количество одновременных обращений* – средняя нагрузка и пиковая нагрузка – не одно и то же. Определение максимального поддерживаемого количества одновременных обращений явля-

ется достаточно трудной задачей. При смене или модернизации существующего приложения можно использовать накопленные статистические данные о его эксплуатации. Имеющиеся данные можно экстраполировать, чтобы оценить вероятные требования к новому приложению. Кроме этого, при оценке проектной загрузки системы нужно остерегаться и ее завышения.

Импорт и массовая загрузка данных

Во многих, если не большинстве, случаях развертывание баз данных любого типа начинается не с нуля. При развертывании новой базы данных начальные данные можно получить из устаревшего приложения, от сторонней системы или импортировать тестовые данные, например сгенерированные в примерах из этой главы. В определенный момент придется выполнить массовую загрузку данных из предоставляющих их систем в рабочее хранилище.

База данных Neo4j предоставляет инструменты для первоначальной массовой загрузки и для текущих операций массового импорта потока данных из различных источников.

Первоначальный импорт

Для первоначального импорта база данных Neo4j предлагает инструмент `neo4j-import`, который позволяет обрабатывать данные со средней скоростью 1 000 000 записей в секунду[1]. Такие впечатляющие показатели производительности обеспечиваются за счет отказа от обычно используемых в базах данных транзакционных механизмов. Вместо этого файлы хранилища заполняются методами, подобными методам обработки растровых данных, при использовании которых отдельные слои накладываются друг на друга, и лишь по завершении картина становится *согласованной*.

Входными данными инструменту `neo4j-import` служит набор CSV-файлов с данными об узлах и взаимосвязях. Для примера рассмотрим следующие три CSV-файла с информацией о фильмах.

Первый файл *movies.csv* содержит следующее:

```
:ID,title,year:int,:LABEL
1,"The Matrix",1999,Movie
2,"The Matrix Reloaded",2003,Movie;Sequel
3,"The Matrix Revolutions",2003,Movie;Sequel
```

[1] Имеется в виду новая реализация инструмента, доступная в Neo4j версии 2.2 и выше.

Первый файл содержит сведения о самих фильмах. В первой строке находится список метаданных, описывающих фильмы. Из них можно определить, что каждый фильм имеет идентификатор ID, название title и год выпуска year (целое число). Поле ID служит ключом. Другие поля являются сведениями о фильме с указанным ID. У фильма также может иметься одна или несколько меток, таких как Movie (кинофильм) и Sequel (продолжение).

Второй файл *actors.csv* содержит сведения о киноактерах: идентификатор ID, свойство name (имя) и метка Actor (актер):

```
:ID,name,:LABEL
keanu,"Keanu Reeves",Actor
laurence,"Laurence Fishburne",Actor
carrieanne,"Carrie-Anne Moss",Actor
```

Третий файл *roles.csv* определяет роли, сыгранные актерами в фильмах. Он используется для создания взаимосвязей в графе:

```
:START_ID,role,:END_ID,:TYPE
keanu,"Neo",1,ACTS_IN
keanu,"Neo",2,ACTS_IN
keanu,"Neo",3,ACTS_IN
laurence,"Morpheus",1,ACTS_IN
laurence,"Morpheus",2,ACTS_IN
laurence,"Morpheus",3,ACTS_IN
carrieanne,"Trinity",1,ACTS_IN
carrieanne,"Trinity",2,ACTS_IN
carrieanne,"Trinity",3,ACTS_IN
```

Каждая строка в этом файле содержит идентификаторы START_ID и END_ID, значение role и вид взаимосвязи TYPE. Поле START_ID содержит идентификатор актера ID из файла *actors.csv*. Поле END_ID содержит идентификатор фильма ID из файла *movies.csv*. Описание каждой взаимосвязи включает в себя START_ID и END_ID, свойство role и название типа связи TYPE.

Имея эти файлы, можно запустить инструмент импорта из командной строки:

```
neo4j-import --into target_directory \
--nodes movies.csv --nodes actors.csv --relationships roles.csv
```

При этом neo4j-import создаст файлы хранилища базы данных и поместит их в каталог target_directory.

Пакетный импорт

Другой востребованной возможностью является массовая загрузка внешних данных в *действующий* граф. В базе данных Neo4j это делается с помощью команды `LOAD CSV`. Входными данными для нее являются такие же CSV-данные, как те, что использует `neo4j-import`. Она поддерживает мгновенную загрузку около миллиона элементов, что делает ее идеальным средством для выполнения регулярных пакетных обновлений из внешних систем.

Для примера расширим существующий граф с информацией о фильмах, данными о местах их съемки. Файл *locations.csv* содержит поля `title` (название) и `location` (местоположение), причем последнее содержит список мест, разделенных точками с запятой, где проходили съемки фильма:

```
title,locations "The Matrix",Sydney
"The Matrix Reloaded",Sydney;Oakland
"The Matrix Revolutions",Sydney;Oakland;Alameda
```

Имея эти данные, можно загрузить их в рабочую базу данных Neo4j с помощью следующей Cypher-команды `LOAD CSV`:

```
LOAD CSV WITH HEADERS FROM 'file:///data/locations.csv' AS line
WITH split(line.locations,";") as locations, line.title as title
UNWIND locations AS location
MERGE (x:Location {name:location})
MERGE (m:Movie {title:title})
MERGE (m)-[:FILMED_IN]->(x)
```

Первая строка содержит команду загрузки CSV-данных из файла, который можно найти по указанному адресу (команда `LOAD CSV` работает также с URI-адресами). Фраза `WITH HEADERS` сообщает базе данных, что первая строка CSV-файла содержит названия заголовков. Фраза `AS line` присваивает входной файл переменной `line`. Остальная часть сценария будет выполнена для каждой строки CSV-данных исходного файла.

Вторая строка с ключевым словом `WITH` указывает на необходимость разбивки значения поля `locations` на коллекцию строк с помощью функции `split` языка Cypher. Затем полученный набор со значением поля `title` передается остальной части скрипта.

С ключевого слова `UNWIND` начинается самое интересное. Фраза `UNWIND` служит для расщепления коллекции. В данном случае она используется для расщепления коллекции `locations` на отдельные строки `location` (вспомним, что здесь мы имеем дело с местами съемок

одного фильма), каждая из которых будет обработана следующими ниже операторами MERGE.

Первый оператор MERGE преобразует местоположение в узел базы данных. Второй – преобразует название фильма в узел. Третий оператор MERGE образует взаимосвязь FILMED_IN (снято в) между узлами местоположения и фильма.

MERGE можно рассматривать как смесь MATCH и CREATE. Если шаблон, указанный в операторе MERGE, уже существует в графе, идентификаторы, заданные в операторе, будут связаны с уже имеющимися данными, подобно MATCH. Если шаблон отсутствует в графе, MERGE создаст его, подобно CREATE.

Чтобы фраза MERGE признала уже имеющиеся данные соответствующими, все элементы шаблона должны существовать в графе. Если будут найдены не все данные, соответствующие *всем* частям шаблона, MERGE создаст новый экземпляр *целого* шаблона. Именно поэтому в сценарии команды LOAD CSV использованы три оператора MERGE. Возьмем конкретный фильм и конкретное местоположение, вполне возможно, то или другое уже присутствует в графе. Возможно, также существует и то, и другое, но без соединяющей их взаимосвязи. Если бы вместо трех операторов был использован один большой оператор MERGE:

```
MERGE (:Movie {title:title})-[:FILMED_IN]-> (:Location
{name:location}))
```

соответствие было бы найдено, только если бы существовали узел фильма *и* узел местоположения *и* взаимосвязь между ними. Если хотя бы одна часть шаблона отсутствует, будут созданы все его части, что приведет к дублированию данных.

Примененный здесь подход расщепляет большой шаблон на отдельные фрагменты. Первым проверяется существование местоположения. Затем – существование фильма. И наконец, соединение двух узлов. Такой поэтапный подход часто применяется при использовании оператора MERGE.

Итак, мы познакомились с возможностью массово добавлять CSV-данные в действующий граф. Но еще не рассмотрели результаты такого импорта. Если выполнить подобный этому запрос на большом наборе данных, добавление займет очень много времени. Для повышения эффективности следует обратить внимание на два ключевых момента:

- индексация существующего графа;
- использование транзакции при записи в базу данных.

Те, кто работал с реляционными базами, сочтут необходимость индексации очевидной. Без индексов придется выполнить обход всех узлов фильмов в базе данных (в худшем случае – вообще всех узлов), чтобы проверить наличие фильма. С точки зрения затрат времени

такую операцию можно выразить как $O(n)$. При наличии индекса по фильмам затраты упадут до $O(\log n)$, что означает существенное увеличение производительности, особенно для больших наборов данных. То же самое можно сказать и о местах съемок.

Создание индексов, как было описано в предыдущей главе, реализуется очень просто. Чтобы создать индекс для фильмов, достаточно выполнить команду `CREATE INDEX ON :Movie(title)`. Сделать это можно в браузере или в командной оболочке. Если индекс используется только для импорта (например, если он не играет никакой роли в запросах), после импорта его можно удалить командой `DROP INDEX ON :Movie(title)`.

В некоторых случаях бывает удобно присваивать узлам временные идентификаторы в виде свойств, чтобы упростить ссылку на них при импорте, особенно при создании сети взаимосвязей. Эти идентификаторы не имеют никакого отношения к прикладной области. Они предназначены только для поиска конкретных соединяемых узлов в процессе многошагового импорта.

Использование временных идентификаторов вполне допустимо. Только не забудьте удалить их с помощью `REMOVE` сразу по завершении импорта.

Обновления в экземплярах Neo4j имеют транзакционную природу, откуда следует, что пакетная операция импорта `LOAD CSV` также является транзакционной. В простейшем случае команда `LOAD CSV` запускает единственную транзакцию и загружает все данные в базу данных. Однако при добавлении больших объемов данных это решение может оказаться неэффективным, так как базе данных придется иметь дело с транзакциями, содержащими слишком большое количество данных (иногда в несколько гигабайт).

Увеличить скорость импорта больших объемов данных можно, разбив одну большую транзакцию на серию более мелких, выполняемых базой данных последовательно. Добиться этого можно с помощью инструмента `PERIODIC COMMIT`, который разбивает импорт на множество транзакций, которые фиксируются после обработки определенного количества строк (по умолчанию 1000). Например, в случае добавления мест съемок можно уменьшить количество CSV-строк на транзакцию до 100, добавив в начало предыдущего Cypher-сценария строку: `USING PERIODIC COMMIT 100`. Полный сценарий будет выглядеть так:

```
USING PERIODIC COMMIT 100
LOAD CSV WITH HEADERS FROM 'file:///data/locations.csv' AS line
```

```
WITH split(line.locations,";") as locations, line.title as title
UNWIND locations AS location
MERGE (x:Location {name:location})
MERGE (m:Movie {title:title})
MERGE (m)-[:FILMED_IN]->(x)
```

Средства массовой загрузки данных позволяют проводить эксперименты с примерами наборов данных при проектировании системы, а также обеспечивают интеграцию с другими системами и источниками данных при развертывании. Формат CSV является популярным форматом обмена данными, практически все технологии обработки и интеграции данных поддерживают вывод в CSV-файлы. А это значительно упрощает как разовый, так и периодический импорт данных в базу данных Neo4j.

Итоги

В этой главе мы рассмотрели наиболее важные аспекты разработки приложений графовых баз данных. Мы узнали, как создавать графовые модели, основываясь на потребностях приложений и целях конечных пользователей, и как сделать модели и запросы наглядными и надежными с помощью модульных тестов и тестов производительности. Кроме того, мы оценили плюсы и минусы нескольких разных архитектур приложений и перечислили факторы, которые следует учитывать при планировании эксплуатации.

Наконец, мы познакомились с вариантами быстрой загрузки множества данных в базу данных Neo4j при первоначальном импорте и при периодически выполняемых пакетных добавлениях в уже функционирующую базу данных.

В следующей главе мы рассмотрим, как в настоящее время с помощью графовых баз данных решаются реальные задачи в таких разнообразных областях, как социальные сети, системы рекомендаций, управление нормативно-справочной информацией, управление центром обработки данных, управление доступом и логистика.

Глава 5

Графы в реальном мире

В этой главе мы рассмотрим некоторые типичные случаи практического использования графовых баз данных и выясним причины их выбора вместо реляционных или других NOSQL-хранилищ. Большую часть главы занимает описание трех примеров, включая подробную информацию о соответствующих моделях данных и запросах. Все примеры основываются на системах, существующих в действительности, но в них изменены названия и опущены некоторые технические детали, чтобы сделать акцент на ключевых моментах разработки и скрыть сложности, не имеющие отношения к предмету обсуждения.

Почему выбирают графовые базы данных

В этой книге мы не раз пели дифирамбы графовой модели данных, ее мощи, гибкости и естественной наглядности. Когда дело доходит до применения графовых баз данных для решения практических задач, технических или коммерческих, многие организации обосновывают свой выбор следующими причинами:

- производительность: сжатие минут в миллисекунды. Производительность запросов и время отклика являются основными проблемами многих платформ обработки данных. В настоящее время транзакционные веб-системы, в том числе крупные веб-приложения, должны обеспечивать отклик на запросы конечных пользователей в течение миллисекунд. В реляционных базах данных с увеличением их размеров начинают проявляться недостатки соединения таблиц и ухудшается производительность. Использование смежности без индексов позволяет графовым базам данных быстро перемещаться по сложным соеди-

ниям в графе, обеспечивая миллисекундное время отклика независимо от общего размера набора данных.

- значительное ускорение разработки. Графовые модели данных смягчают несогласованность, десятилетиями преследующую разработчиков программного обеспечения, и снижают накладные расходы на преобразования между объектной и реляционной моделями. Кроме того, графовая модель сглаживает несоответствия между технической и прикладной сферами. Специалисты в прикладной области, проектировщики и разработчики могут обсуждать и представлять область применения, используя единую модель, которая затем будет включена в приложение.
- удивительная интерактивность. Успешные приложения редко остаются неизменными. Изменения в условиях ведения бизнеса, в пожеланиях пользователей, в технической и эксплуатационной инфраструктурах ведут к появлению новых требований. Раньше реорганизация потребовала бы опасных длительных миграций данных, включающих модификацию схем, преобразование данных, а также поддержание избыточности данных, чтобы обеспечить старые и новые возможности. Свободная от схем природа графовых баз данных в сочетании с возможностью одновременного соединения элементов данных множеством разных способов позволяет приложениям не отставать в развитии от прикладной области, снижая риски и сокращая время выхода на рынок.
- готовность к промышленной эксплуатации. Технологии обработки данных в важных коммерческих приложениях должны быть надежными, масштабируемыми и, как правило, транзакционными. Хотя некоторые графовые базы данных созданы недавно и еще не достигли своей зрелости, но уже существуют графовые базы данных, которые отвечают всем требованиям ACID (Atomic – атомарность, Consistent – согласованность, Isolated – изолированность, Durable – долговечность) к транзакциям, обеспечивают высокую надежность, горизонтальную масштабируемость чтения и хранение миллиардов сущностей – все то, что сегодня необходимо крупным компаниям, – а также производительность и гибкость. Все это стало важным фактором использования графовых баз данных не только для каких-либо автономных и специфичных целей, но и для охвата всего производственного процесса.

Типичные примеры использования

В этом разделе будет описано несколько типичных примеров использования графовых баз данных и продемонстрирована ценность некоторых особенностей графовых моделей и графовых баз данных для промышленного применения.

Социальные сети

Мы только начинаем постигать мощь социальной информации. В своей книге «*Connected*» социологи Николас Кристакис (Nicholas Christakis) и Джеймс Фаулер (James Fowler) описали, как предсказывать поведение личности по ее контактам[1].

Социальные приложения позволяют организациям получить преимущество в конкуренции и повысить свою оперативность с помощью информации о связях между людьми. Объединив дискретную информацию о физических лицах со сведениями об их взаимосвязях, можно содействовать их сотрудничеству, управлять их информированностью и прогнозировать их поведение.

Используемый в Facebook термин *социальный граф* подразумевает, что графовая модель данных и графовая база данных являются естественными для этой открытой области, основанной на взаимосвязях. Социальные сети помогают идентифицировать прямые *и* косвенные отношения между людьми, группами людей и характер их взаимодействий, что позволяет пользователям оценивать, анализировать, находить друг друга и то, что их интересует. Разбираясь в том, кто с кем взаимодействует, как люди связаны между собой, что представляет собой группа, чем будет группа занимается и что она выбирает в совокупности, можно понять действие невидимых сил, влияющих на индивидуальное поведение. Более подробно прогнозное моделирование и его роль в анализе социальной сети будет рассмотрена в разделе «Теория графов и прогнозное моделирование» ниже.

Социальные отношения могут быть явными или неявными. Явные отношения возникают там, где социальные объекты сами создают прямые ссылки, например ссылаясь на кого-то в Facebook или указывая на коллег или на бывших коллег в LinkedIn. Неявные отношения

[1] Nicholas Christakis, James Fowler. Connected: The Amazing Power of Social Networks and How They Shape Our Lives (HarperPress, 2011) – Николас Кристакис, Джеймс Фаулер. Связанные одной сетью. Как на нас влияют люди, которых мы никогда не видели. Юнайтед Пресс, 2011. ISBN: 978-5-4295-0022-5.

возникают из других отношений, которые косвенно связывают двух или более особ через посредника. Людей можно связать на основании их мнений, пристрастий, покупок и даже программ, которыми они ежедневно пользуются. Такие косвенные отношения можно использовать как почву для размышлений и строить на ней умозаключения. Можно предположить, что A, *вероятнее всего,* знаком, симпатизирует или связан каким-либо иным способом, с B, основываясь на знакомстве их обоих с одними и теми же лицами. При этом происходит переход от анализа социальной сети к рекомендательным системам.

Рекомендации

Эффективные рекомендации являются ярким примером оценки конечного пользователя с помощью логических выводов или догадок. В бизнес-приложениях, как правило, применяются точные дедуктивные алгоритмы для генерации связанных с пользователями значений, например для расчета заработной платы, налогов и т. д., однако алгоритмы рекомендаций носят индуктивный характер и основаны на предположениях, что люди, товары или услуги, *скорее всего*, заинтересуют определенное лицо или группу лиц.

Рекомендательные алгоритмы устанавливают отношения между людьми и объектами: другими людьми, товарами, услугами, медиаконтентом и всем прочим, имеющим отношение к области рекомендаций. Устанавливаемые отношения основываются на действиях пользователей, выражающихся в их покупках, предложениях, расходах, оценках или обзорах. Рекомендательные алгоритмы позволяют определить ресурсы, представляющие интерес для конкретного лица или конкретной группы лиц, а также отдельных лиц или отдельных групп лиц, которых может заинтересовать конкретный ресурс. В первом случае, т. е. при выявлении ресурсов, представляющих интерес для конкретного пользователя, действия пользователя, такие как покупки, явно выраженное предпочтение и отношение в виде рейтингов и обзоров, коррелируют с действиями других пользователей для выявления похожих пользователей и, следовательно, сходства их интересов. Во втором случае, при выявлении пользователей и групп, интересующихся конкретным ресурсом, основное внимание уделяется характеристикам рассматриваемого ресурса. Затем выполняется идентификация подобных ресурсов и пользователей, связанных с ними.

Как и в примере для социальной сети, эффективность рекомендаций зависит от понимания связей между объектами, а также от учета качеств и прочности этих связей, т. е. всего того, что наилучшим об-

разом может быть представлено в виде графа со свойствами. Запросы сначала находят определенное место в графе, поэтому они и начинаются с идентификации одного или нескольких объектов, людей или ресурсов, а затем обследуют окружающую его графовую область.

Социальные сети и генераторы рекомендаций, применяемые совместно, отлично зарекомендовали себя в области розничной торговли, найма, анализа настроений, поиска и управления знаниями. Графы хорошо подходят для моделирования тесно связанных структур данных, характерных для этих областей. Хранение этих данных и выполнение запросов к ним с помощью графовых баз данных позволяет приложению выдавать конечным пользователям результаты в режиме реального времени, отражающие последние изменения в данных, а не предварительно рассчитанные устаревшие результаты.

Геоинформационные системы

Геоинформационные системы представляют собой оригинальную область использования графов. Эйлер, решив задачу о семи мостах Кенигсберга, сформулировал математическую теорему, которая позже была положена в основу теории графов. Примеры применения графовых баз данных в геоинформационных приложениях охватывают диапазон задач от расчета маршрутов между точками в абстрактной сети, такой как автомобильные дороги, железнодорожные пути, сети воздушного сообщения или логистические сети (пример последней описан ниже в этой же главе) до пространственных операций, таких как поиск всех интересующих точек в ограниченной области, поиск центра области и расчет пересечений между двумя и более областями.

Геоинформационные операции зависят от специфики структур данных, охватывающих диапазон от простых весовых и направленных взаимосвязей до пространственных индексов, таких как R-деревья (https://ru.wikipedia.org/wiki/R-дерево), которые представляют многомерные свойства с помощью древовидных структур данных. Эти структуры данных органично преобразуются в форму графов, как правило, в иерархическую форму, и хорошо подходят для хранения в графовых базах данных. Из-за отсутствия схем в графовых базах данных геоинформационные данные могут храниться вместе с другими видами сетевых данных, например данных из социальной сети, позволяя выполнять сложные многомерные запросы по нескольким прикладным областям[1].

[1] Neo4j Spatial (https://github.com/neo4j/spatial) – открытая библиотека утилит для реализации геопространственных указателей и использования данных Neo4j в геоинструментах.

Использование геоинформационных приложений графовых баз данных особенно актуально в области телекоммуникаций, логистики, путешествий, расписаний и планирования маршрутов.

Управление справочными данными

Справочными называют данные, которые имеют решающее значение для функционирования бизнеса, но сами по себе не являются транзакционными. К справочным данным относятся данные о пользователях, клиентах, товарах, поставщиках, отделах, местоположениях, сайтах, центрах затрат и бизнес-единицах. В крупных организациях эти данные часто находятся в разных местах, нередко при их организации допускаются значительная избыточность и дублирование, они хранятся в нескольких различных форматах, с разной степенью качества, и для работы с ними применяются различные средства. *Управление справочными данными* (*Master Data Management, MDM*) – это набор методик идентификации, очистки, хранения и, что самое главное, манипуляции этими данными. Ключевыми задачами управления являются: обновление данных с течением времени (по мере изменения организационных структур, при слиянии фирм и при смене бизнес-правил), подключение новых источников данных, дополнение существующих данных сведениями из внешних источников, удовлетворение потребностей в отчетности, поддержание согласованности и пригодности для анализа, хранение различных версий данных при изменениях оценок и схем.

Графовые базы данных не всегда способны обеспечить полную поддержку всех задач MDM. Но они идеально подходят для моделирования, хранения и выборки иерархических метаданных и моделей справочных данных. Их моделирование включает определение типов, ограничений, взаимосвязей между сущностями и сопоставление модели с исходной системой. Не привязанная к схемам структура графовой базы данных обеспечивает моделирование специализированных, изменчивых и исключительных структур, содержащих схемные аномалии, обычно возникающие при наличии нескольких резервных источников данных, в то же время позволяя быстрое обновление моделей справочных данных без отставания от быстро меняющихся потребностей.

Сети и управление центром обработки данных

В главе 3 была описана простая модель управления центром обработки данных, демонстрирующая, как легко можно представить в виде

графа физические и виртуальные ресурсы в этом центре. Сетевые коммуникации являются графовыми структурами. Поэтому графовые базы данных хорошо подходят для моделирования, хранения и извлечения данных такого рода прикладных областей. Различие между управлением большой сетью коммуникаций и управлением центром обработки данных в значительной степени определяется тем, с какой стороны от брандмауэра ведется работа. Задачи и цели моделирования этих двух областей схожи.

Представление сетей в виде графа позволяет создать каталог ресурсов, визуализировать их развертывание и выявить взаимосвязи между ними. Графовое моделирование структуры соединений вместе с языком запросов, таким как Cypher, позволяет провести сложный анализ взаимодействий и получить ответы на вопросы, подобные перечисленным ниже:

- какие части сети, т. е. приложения, службы, виртуальные серверы, физические серверы, центры обработки данных, маршрутизаторы, коммутаторы и волоконные коммуникации, являются наиболее важными при обслуживании пользователей? (Анализ сверху вниз.)
- интересы каких приложений и служб, а в конечном итоге и пользователей сети будут затронуты, если конкретный элемент сети, например маршрутизатор или коммутатор, выйдет из строя? (Анализ снизу вверх.)
- предусмотрено ли в сети резервирование для наиболее важных клиентов?

Решения на основе графовых баз данных значительно расширяют набор инструментов управления и анализа сетью. Как и в случае управления справочными данными, для объединения данных с разными системами организации могут использоваться дополнительные инструменты, что позволит достичь единого представления о сети и ее пользователях, охватывающего диапазон от самых незначительных сетевых элементов до приложений, служб и их клиентов. Представление сети с помощью графовой базы данных также может быть использовано для расширенного анализа оперативных данных, основанного на корреляции событий. Каждый раз при выявлении корреляции между событиями (например, Complex Event Processor, http://ru.wikipedia.org/wiki/Обработка_сложных_событий) комплексного события из потока низкоуровневых событий сети можно оценить влияние этого события с помощью графовой модели, а затем вызвать необходимое компенсирующее или смягчающее действие.

Сегодня графовые базы данных успешно используются в таких областях, как телекоммуникации, управление и анализ сетей, управление облачными платформами, центры обработки данных и управление вычислительными ресурсами, а также для анализа влияния неполадок на сети, где они уменьшают время анализа последствий сбоев и решения проблем от дней и часов до минут и секунд. Здесь важными факторами применимости являются производительность и гибкость в условиях часто меняющихся схем сетей.

Авторизация и контроль доступа (для коммуникаций)

Решения, связанные с авторизацией и контролем доступа, предназначены для хранения информации о клиентах (например, администраторах, организационных единицах, конечных пользователях) и ресурсах (например, файлах, общедоступных ресурсах, сетевых устройствах, товарах, услугах, договорах) вместе с правилами, регулирующими доступ к этим ресурсам. Кроме того, они обеспечивают применение этих правил при определении доступности ресурсов. Управление доступом традиционно реализуется либо с помощью служб каталогов, либо с помощью нестандартных решений. Но системы с неиерархической организацией и структуры, зависимые от ресурсов, для которых характерно применение многократно разорванных цепочек, не могут эффективно работать с иерархическими структурами каталогов. Решения с искусственной поддержкой таких структур, в частности разработанные на основе реляционных баз данных, страдают от применения соединений, что при росте размеров набора данных ведет к увеличению времени отклика и в конечном итоге делает невозможным их использование.

Графовые базы данных способны хранить сложные тесно связанные между собой структуры управления доступом, охватывающие миллиарды клиентов и ресурсов. Их модель данных, свободная от схем, поддерживает как иерархические структуры, так и неиерархические, кроме того, свойства, расширяющие модель, позволяют охватить множество сведений о каждом элементе системы. Их механизмы обработки запросов позволяют обойти миллионы взаимосвязей за секунду и находить данные о доступе в больших комплексных структурах за миллисекунды.

Решения для управления и анализа сети, контроля доступа, основанные на графовых базах данных, позволяют получить ответы на следующие вопросы:

- Какими ресурсами (структурными единицами, товарами, услугами, договорами и конечными пользователями) может управлять конкретный администратор? (Сверху вниз.)
- Какой ресурс доступен конечному пользователю?
- Кто может изменять настройки доступа к определенному ресурсу? (Снизу вверх.)

Решения для управления доступом и авторизации, основанные на графовых базах данных, применяются в областях управления контентом, для авторизации в федеральных службах, социальных сетях и платформах как услугах (Software as a Service, SaaS), где позволяют сократить время отклика с минут до миллисекунд, значительно увеличивая производительность, по сравнению со своими кустарными реляционными предшественниками.

Реальные примеры

В этом разделе будут подробно описаны три примера из следующих областей: социальные сети и система рекомендаций, авторизация и управление доступом, логистика. Каждый из приведенных здесь примеров почерпнут из одного или нескольких действующих приложений, основанных на графовых базах данных (конкретно в этих примерах – на Neo4j). Названия компаний, особенности контекста, модели данных и запросы были изменены, чтобы уменьшить влияние мешающих пониманию специфичных факторов и подчеркнуть важные аспекты проектирования и реализации.

Социальные рекомендации (Профессиональная социальная сеть)

Talent.net является приложением для создания социальной сети рекомендаций, дающим пользователям возможность создать собственную профессиональную сеть, а также познакомиться с другими пользователями, имеющими определенный профессиональный опыт. Люди, работая над проектами в компаниях, обладают некими профессиональными интересами и познаниями. Основываясь на этой информации, Talent.net создает профессиональную сеть пользователя, выявляя других пользователей с похожими интересами. Поисковые запросы могут быть ограничены компанией пользователя или расширены до охвата всей базы пользователей. Talent.net может также находить лиц, обладающих конкретными умениями, прямо или косвен-

но связанных с текущим пользователем. Такая возможность может пригодиться при выборе экспертов в определенной области, которым можно предложить сотрудничество.

Talent.net демонстрирует всю широту возможностей графовых баз данных. Хотя многие бизнес-приложения являются, по своей сути, дедуктивными и точными, например расчет налогов или зарплаты, составление бухгалтерского баланса, тем не менее применение индуктивных методов анализа данных позволяет обеспечить конечным пользователям новые возможности. Именно это и делает Talent.net. Основываясь на интересах и знаниях людей, их опыте работы, приложение выявляет вероятных кандидатов для включения в их сети по профессиональным интересам. Результаты работы приложения не являются точными, в смысле не такими точными, каким должен быть расчет заработной платы, но тем не менее они приносят несомненную пользу.

Talent.net *выявляет* связи между людьми. Сравните его с LinkedIn, где пользователи явно указывают, что они знают кого-либо или работали раньше с кем-либо. Это не значит, что LinkedIn использует лишь точные возможности социальных сетей, потому что там также применяется и индуктивный алгоритм для расширенного анализа. Но в Talent.net даже первичные связи `(A)-[:KNOWS]->(B)` (`A` знает `B`) выявляются, а не задаются явно.

Первая версия Talent.net, основываясь на информации, предоставленной пользователями, об их интересах, знаниях и опыте работы, может выявить их профессиональные социальные взаимосвязи. Но, обладая этими базовыми сведениями и возможностями, платформа способна на более глубокий анализ без дополнительных усилий со стороны конечных пользователей. Умения и интересы, например, могут выявляться путем анализа ежедневного участия лица в мероприятиях и созданных им продуктах. Занимаясь разработкой программного кода, подготовкой документов или обмениваясь электронными письмами, пользователь взаимодействует с серверной системой. Перехватывая эти взаимодействия, Talent.net захватывает данные, которые указывают, какие познания есть у определенного лица и в каких мероприятиях оно участвует. Другими источниками данных являются членство в группах и списки встреч. Хотя представленные здесь описания примеров не охватывают функций вывода на верхнем уровне, их реализация потребует в основном лишь интеграции приложений и партнерских соглашений, а не какого-либо существенного изменения в используемых графах или алгоритмах.

Модель данных Talent.net

Чтобы лучше описать модель данных Talent.net, создадим небольшой граф, изображенный на рис. 5.1, который будет использоваться в этом разделе для выполнения Cypher-запросов, демонстрирующих использование Talent.net.

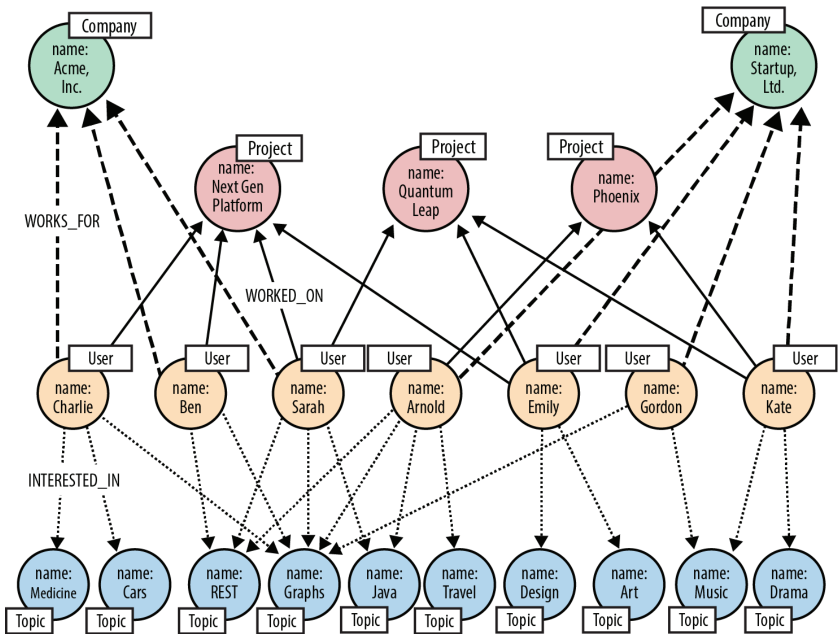


Рис. 5.1 ❖ Пример социальной сети Talent.net

Представленный здесь пример графа включает две компании, с несколькими сотрудниками в каждой. Сотрудники связаны со своими работодателями взаимосвязями WORKS_FOR. Каждый сотрудник интересуется одной или несколькими темами (взаимосвязи INTERESTED_IN) и работает над одним или несколькими проектами (взаимосвязи WORKED_ON). Случается, что сотрудники из разных компаний работают над одним и тем же проектом.

Эта структура предназначена для решения двух важных задач:

- выявление взаимосвязей пользователя, т. е. создание социальной сети по общим профессиональным интересам и знаниям;

- выявление пользователей через рекомендации тех, с кем они работают или работали, которые имеют определенный опыт и навыки.

Решение первой задачи позволяет строить сообщества людей с общими интересами. Решение второй помогает выявлять кандидатов для заполнения определенных вакансий в проекте.

Выявление социальных взаимосвязей

Граф Talent.net позволяет создать профессиональную социальную сеть пользователя, определив людей, разделяющих его профессиональные интересы. Сила рекомендаций зависит от количества общих интересов. Если Сару (`Sarah`) интересуют `Java`, `Graphs` и `REST`, Бена (`Ben`) интересуют `Graphs` и `REST`, а Чарли (`Charlie`) интересуют `Graphs`, `Cars` и `Medicine`, то связь между Сарой и Беном основана на общих интересах `Graphs` и `REST`, а связь между Сарой и Чарли – на общем интересе `Graphs`, следовательно, связь между Сарой и Беном сильнее, чем связь между Сарой и Чарли (два общих интереса против одного).

На рис. 5.2 изображен шаблон для выявления коллег, разделяющих интересы пользователя. Узел `subject` привязывается к субъекту запроса (в предыдущем примере это Сара (`Sarah`)). Этот узел можно найти по индексу. Остальные узлы будут найдены после прикрепления шаблона к узлу субъекта и обхода его окружения.

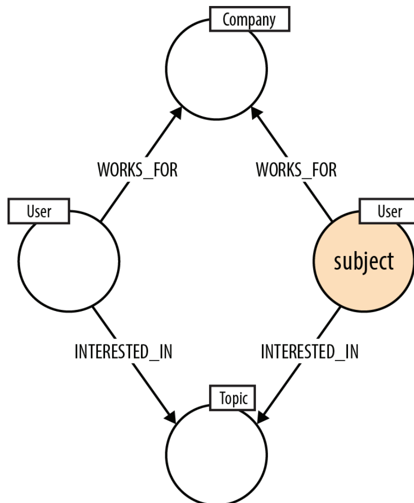


Рис. 5.2 ❖ Шаблон для поиска коллег, разделяющих интересы пользователя

Этот шаблон реализует следующий Cypher-запрос:

```
MATCH (subject:User {name:{name}})
MATCH (subject)-[:WORKS_FOR]->(company:Company)<-[:WORKS_FOR]-(person:User),
      (subject)-[:INTERESTED_IN]->(interest)<-[:INTERESTED_IN]-(person:User)
RETURN person.name AS name,
       count(interest) AS score,
       collect(interest.name) AS interests
ORDER BY score DESC
```

Запрос действует следующим образом:

- первая фраза `MATCH` находит субъекта запроса (`Sarah`) среди узлов с меткой `User` и присваивает ему идентификатор `subject`;
- вторая фраза `MATCH` подбирает узлы с меткой `User`, соответствующие работникам той же компании, имеющим один или несколько общих интересов. Если субъектом запроса является Сара, которая работает в компании `Acme`, для Бена будут выявлены два совпадения: Бен работает в компании `Acme` и интересуется `Graphs` (первое совпадение) и `REST` (второе совпадение). Для Чарли будет выявлено одно совпадение: Чарли работает в компании `Acme` и интересуется `Graphs`;
- фраза `RETURN` создает проекцию данных о совпадениях. Для каждого совпавшего коллеги извлекаются имя и количество интересов, общих с субъектом запроса (этому количеству присваивается псевдоним `score`), затем с помощью функции `collect` создается список общих интересов. Если лицо имеет несколько совпадений, как Бен в нашем примере, функции `count` и `collect` агрегируют его совпадения в возвращаемых результатах в одну строку (фактически для такого агрегирования было бы достаточно и одной из функций `count` или `collect`);
- и наконец, результаты сортируются по значениям `score`, причем по убыванию.

При выполнении этого запроса, к примеру графа, с Сарой в качестве субъекта будут получены следующие результаты:

```
+-------------------------------------+
| name      | score | interests       |
+-------------------------------------+
| "Ben"     | 2     | ["Graphs","REST"] |
| "Charlie" | 1     | ["Graphs"]      |
+-------------------------------------+
2 rows
```

На рис. 5.3 изображена часть графа, содержащая полученные результаты.

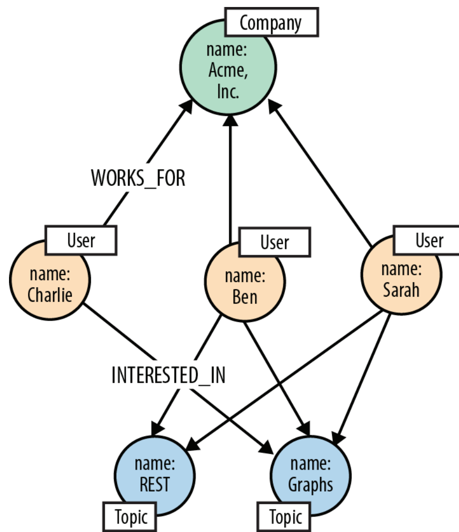


Рис. 5.3 ❖ Коллеги, разделяющие интересы Сары

Обратите внимание, что этот запрос находит только людей, работающих в той же компании, что и Сара. Чтобы распространить поиск на другие компании, следует несколько изменить запрос:

```
MATCH (subject:User {name:{name}})
MATCH (subject)-[:INTERESTED_IN]->(interest:Topic)
        <-[:INTERESTED_IN]-(person:User),
      (person)-[:WORKS_FOR]->(company:Company)
RETURN person.name AS name,
       company.name AS company,
       count(interest) AS score,
       collect(interest.name) AS interests
ORDER BY score DESC
```

Изменения заключаются в следующем:

- из фразы MATCH исключено требование принадлежности лиц к компании субъекта запроса (однако название компании, где работает рассматриваемое лицо, все еще необходимо, потому что эту информацию нужно будет включить в результат запроса);

- фраза `RETURN` теперь включает в результат информацию о компании для каждого рассматриваемого лица.

Выполнение этого запроса для того же примера данных вернет следующие результаты:

```
+------------------------------------------------------------------+
| name      | company         | score | interests                  |
+------------------------------------------------------------------+
| "Arnold"  | "Startup, Ltd"  | 3     | ["Java","Graphs","REST"]   |
| "Ben"     | "Acme, Inc"     | 2     | ["Graphs","REST"]          |
| "Gordon"  | "Startup, Ltd"  | 1     | ["Graphs"]                 |
| "Charlie" | "Acme, Inc"     | 1     | ["Graphs"]                 |
+------------------------------------------------------------------+
4 rows
```

Рисунок 5.4 демонстрирует часть графа, соответствующую этим результатам.

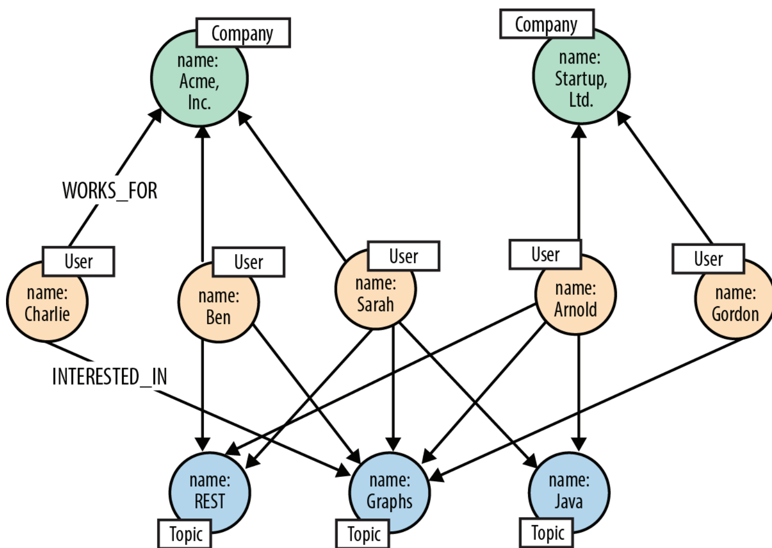


Рис. 5.4 ❖ Лица, разделяющие интересы Сары

Хотя Бен и Чарли по-прежнему присутствуют в результатах, но Арнольд (`Arnold`), работающий в компании `Startup, Ltd.`, имеет больше общих интересов с Сарой: три пункта, по сравнению с одним у Бена и двумя у Чарли.

Поиск коллег с определенными интересами

Во втором варианте Talent.net будет сделан переход от выявления социальных взаимосвязей на общих интересах к поиску лиц, имеющих определенный набор умений, которые либо работали с субъектом запроса, либо работали с людьми, работающими с субъектом запроса. Применяя граф таким способом, можно заполнить вакансии в проекте на основании социальных связей доверенных лиц или, по крайней мере, лиц, с которыми ранее уже работали.

Социальные связи объединяют лиц, работавших над одним и тем же проектом. Сравним это с предыдущим случаем, где социальные связи выявляются на основании общих интересов. Если люди работали над одним проектом, выявляется их социальная связь. Проекты формируют промежуточные узлы, которые связывают вместе двух и более лиц. Иными словами, проект является примером сотрудничества, которое приводит к необходимости контакта нескольких лиц между собой. Любое выявленное таким образом лицо является кандидатом на включение в результаты, так как оно обладает нужными интересами или навыками.

Следующий Cypher-запрос находит коллег и коллег их коллег, обладающих одним или несколькими конкретными интересами:

```
MATCH (subject:User {name:{name}})
MATCH p=(subject)-[:WORKED_ON]->(:Project)-[:WORKED_ON*0..2]-(:Project)
      <-[:WORKED_ON]-(person:User)-[:INTERESTED_IN]->(interest:Topic)
WHERE person<>subject AND interest.name IN {interests}
WITH person, interest, min(length(p)) as pathLength
ORDER BY interest.name
RETURN person.name AS name,
       count(interest) AS score,
       collect(interest.name) AS interests,
       ((pathLength - 1)/2) AS distance
ORDER BY score DESC
LIMIT {resultLimit}
```

Это довольно сложный запрос. Давайте разобьем его на части и подробно рассмотрим каждую:

- первая фраза `MATCH` находит субъекта запроса среди узлов с меткой `User` и присваивает результат идентификатору `subject`;
- вторая фраза `MATCH` находит лиц, связанных с субъектом `subject` работой над одним проектом или работой над одним проектом с лицами, которые работали с `subject`. Для каждого из выявленных лиц фиксируются его интересы. Совпадения, найден-

ные фразой MATCH, дополнительно уточняются во фразе WHERE, которая исключает узлы, не соответствующие субъекту запроса, и гарантирует наличие у людей нужных интересов. Для каждого найденного совпадения его маршруту присваивается идентификатор p, т. е. маршруту от субъекта запроса через подошедшее лицо к его интересу. Более подробно такая фраза MATCH будет описана чуть ниже;

- фраза WITH передает результаты фразе RETURN, отфильтровав лишние маршруты. Ненужные маршруты здесь все еще присутствуют в результатах, потому что коллеги и коллеги коллег часто оказываются связанными разными путями, одни из них длиннее, другие короче. Требуется отбросить эти длинные пути. Это именно то, что делает фраза WITH. Фраза WITH исключает дублирование троек, включающих лицо, интерес и длину маршрута, от субъекта запроса через лицо к его интересам. Учитывая, что любая конкретная комбинация лицо/интерес может появляться в результатах более одного раза, но с разными длинами маршрута, требуется объединить такие несколько строк, свернув их в тройку, содержащую кратчайший путь, что и делается с помощью выражения min(length(p)) as pathLength;
- фраза RETURN создает проекцию данных, выполняя дальнейшую агрегацию. Данные, переданные с помощью фразы WITH во фразу RETURN, содержат по одной записи для лица и интереса. Если лицу соответствуют два интереса, ему будут сопоставлены две записи. Эти записи объединяются с помощью функций countи collect, функция count рассчитывает итоговый балл для лица, функция collect создает разделенный запятыми список совпадающих интересов для этого лица. В результатах также рассчитывается удаленность найденного лица от субъекта запроса. Для этого из длины маршрута для лица вычитается единица (для учета взаимосвязей INTERESTED_IN в конце маршрута), а затем результат делится на два (потому что лицо отделено от субъекта парой взаимосвязей WORKED_ON). И наконец, результаты сортируются по значениям score в порядке убывания и отбрасываются в соответствии с параметром resultLimit задаваемого клиентом запроса.

Во второй фразе MATCH в предыдущем запросе используется маршрут переменной длины [:WORKED_ON*0..2] для выявления лиц, которые работали непосредственно с субъектом запроса, и лиц, которые

работали над одним проектом с лицами, работающими с субъектом запроса. Так как каждое лицо отделено от субъекта запроса одной или двумя парами взаимосвязей WORKED_ON, в приложении Talent.net можно было бы записать эту часть запроса в виде MATCH p=(subject)-[:WORKED_ON*2..4]-(person)-[:INTERESTED_IN]->(interest), с переменной длиной маршрута от двух до четырех взаимосвязей WORKED_ON. Но длинные маршруты переменной длины относительно неэффективны. В таких запросах желательно ограничивать число маршрутов переменной длины, насколько это возможно. Для повышения производительности запроса приложение Talent.net использует исходящие взаимосвязи WORKED_ON фиксированной длины от субъекта запроса к его первому проекту и взаимосвязи фиксированной длины WORKED_ON, соединяющие подобранное лицо с проектом с помощью пути между ними, имеющим наименьшую длину.

Этот запрос для приведенного выше графа, Сары в роли субъекта и при поиске ее коллег и коллег ее коллег, интересующихся Java, Travel или Medicine, вернет следующие результаты:

```
+-----------------------------------------------------+
| name      | score | interests          | distance |
+-----------------------------------------------------+
| "Arnold"  | 2     | ["Java","Travel"] | 2        |
| "Charlie" | 1     | ["Medicine"]      | 1        |
+-----------------------------------------------------+
2 rows
```

Отметим, что результаты отсортированы по рейтингу score, а не по удаленности distance. У Арнольда совпадают два из трех интересов, следовательно, его рейтинг выше, чем у Чарли, у которого совпадает только один интерес, хотя Арнольд удален от Сары на две единицы, в то время как Чарли работал непосредственно с Сарой.

На рис. 5.5 показана часть графа, отобранная при его обходе для получения этих результатов.

А теперь воспользуемся моментом, чтобы подробно разобраться, как работает этот запрос. Рисунок 5.6 демонстрирует три этапа выполнения запроса. (Для наглядности были удалены метки и выделены важные свойства). Первый этап содержит список маршрутов, подобранных фразами MATCH и WHERE. Как можно заметить, он содержит один избыточный путь: имеется прямое совпадение Charlie и Next Gen Platform и косвенное с маршрутом через Quantum Leap и Emily. Второй этап представляет фильтрацию во фразе WITH. Здесь выявляются тройки, содержащие лицо, интерес и длину кратчайшего маршрута

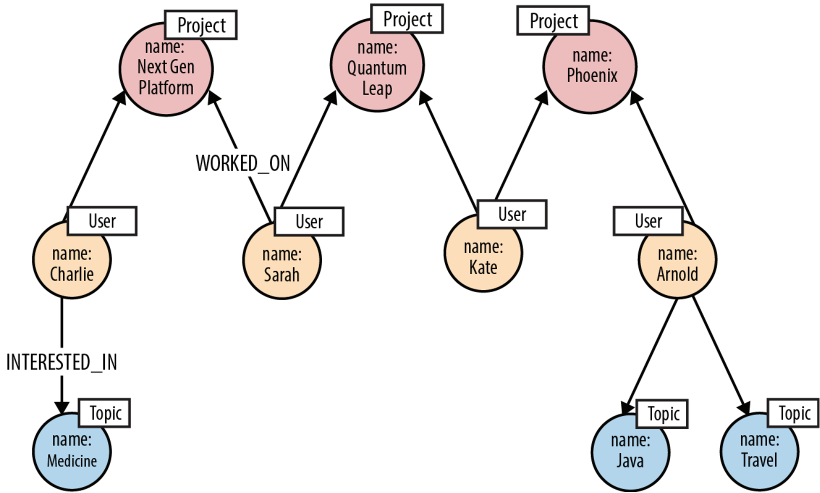


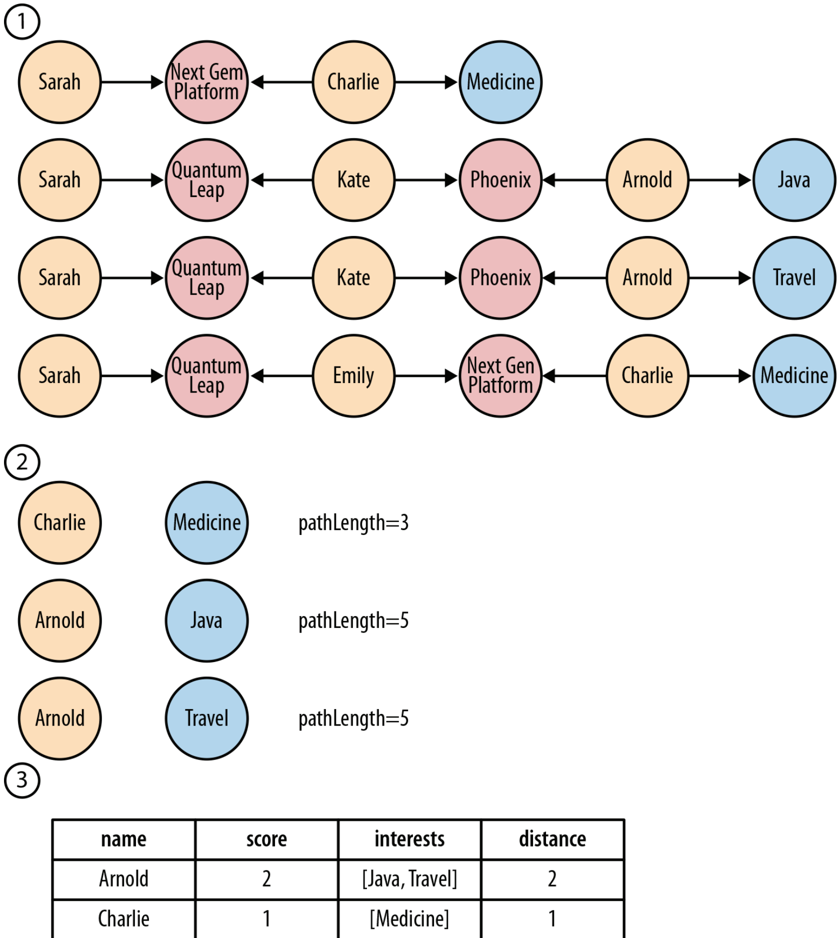
Рис. 5.5 ❖ Поиск людей с заданными интересами

от субъекта через лицо к интересу. Третий этап представляет фразу `RETURN`, где результаты сворачиваются по каждому подошедшему лицу, и вычисляются его рейтинг и удаленность от субъекта.

Добавление взаимосвязей WORKED_WITH

Запрос для поиска коллег и коллег их коллег с конкретными интересами – один из наиболее часто используемых запросов на сайте Talent.net, и успешность сайта в значительной степени зависит от скорости его выполнения. Запрос использует пары взаимоотношений `WORKED_ON` (например, `('Sarah')-[:WORKED_ON]->('Next Gen Platform')<-[:WORKED_ON]-('Charlie')`), чтобы выяснить –работали ли пользователи друг с другом. Он производительный, но все же не самый эффективный, потому что для выявления одной неявной взаимосвязи требует прохода двух явных взаимосвязей.

Чтобы устранить эту неэффективность, разработчики сайта Talent.net решили заранее подготовить взаимосвязи нового вида `WORKED_WITH` и тем самым расширить граф для обработки шаблонов, требующих высокой производительности. Как уже упоминалось в разделе «Итеративная и поэтапная разработка», для оптимизации доступа к графу часто добавляется прямая связь между двумя узлами, которые ранее могли быть связаны только через посредников.



name	score	interests	distance
Arnold	2	[Java, Travel]	2
Charlie	1	[Medicine]	1

Рис. 5.6 ❖ Передача данных в запросе

С точки зрения прикладной области приложения Talent.net взаимосвязи `WORKED_WITH` являются двусторонними. Но в графе они будут реализованы с помощью однонаправленных взаимосвязей. Хотя направления взаимосвязей обычно добавляют полезную семантику в их определение, в этом случае направление не имеет никакого смысла. Это не создаст серьезных проблем, поскольку запросы, работающие со взаимосвязями `WORKED_WITH`, будут игнорировать их направления.

Графовые базы данных поддерживают одинаково быстрый обход взаимосвязей в любом направлении, поэтому нет смысла обсуждать необходимость добавления взаимообратных взаимосвязей прикладной области. Например, нет нужды в одновременном добавлении взаимосвязей PREVIOUS (предыдущий) и NEXT (следующий) в связанный список, но в социальной сети при представлении привязанностей важно явно указать, кто кого любит, потому что это не всегда предполагает взаимность.

Вычисление взаимосвязей WORKED_WITH пользователей и добавление их в граф не является сложной задачей и не приведет к заметным затратам ресурсов. Но при добавлении новой информации в профиль пользователя о проектах такой расчет добавит несколько миллисекунд ко времени ожидания конечных пользователей, поэтому разработчики Talent.net решили выполнять эту операцию асинхронно с обработкой запросов конечных пользователей. При изменении пользователем своей истории работы в проектах Talent.net добавляет задание в очередь. Это задание пересчитывает взаимосвязи WORKED_WITH пользователя. Отдельный поток опрашивает эту очередь и выполняет задание с помощью следующего Cypher-оператора:

```
MATCH (subject:User {name:{name}})
MATCH  (subject)-[:WORKED_ON]->()<-[:WORKED_ON]-(person:User)
WHERE NOT((subject)-[:WORKED_WITH]-(person))
WITH DISTINCT subject, person
CREATE UNIQUE (subject)-[:WORKED_WITH]-(person)
RETURN subject.name AS startName, person.name AS endName
```

На рис. 5.7 изображен пример графа после добавления взаимосвязей WORKED_WITH.

С помощью расширенного графа приложение Talent.net будет выполнять поиск коллег и коллег их коллег с конкретными интересами с помощью более простого варианта запроса, чем был рассмотрен выше:

```
MATCH (subject:User {name:{name}})
MATCH p=(subject)-[:WORKED_WITH*0..1]-(:Person)-[:WORKED_WITH]-(person:User)
      -[:INTERESTED_IN]->(interest:Topic)
WHERE person<>subject AND interest.name IN {interests}
WITH person, interest, min(length(p)) as pathLength
RETURN person.name AS name,
       count(interest) AS score,
       collect(interest.name) AS interests,
       (pathLength - 1) AS distance
ORDER BY score DESC
LIMIT {resultLimit}
```

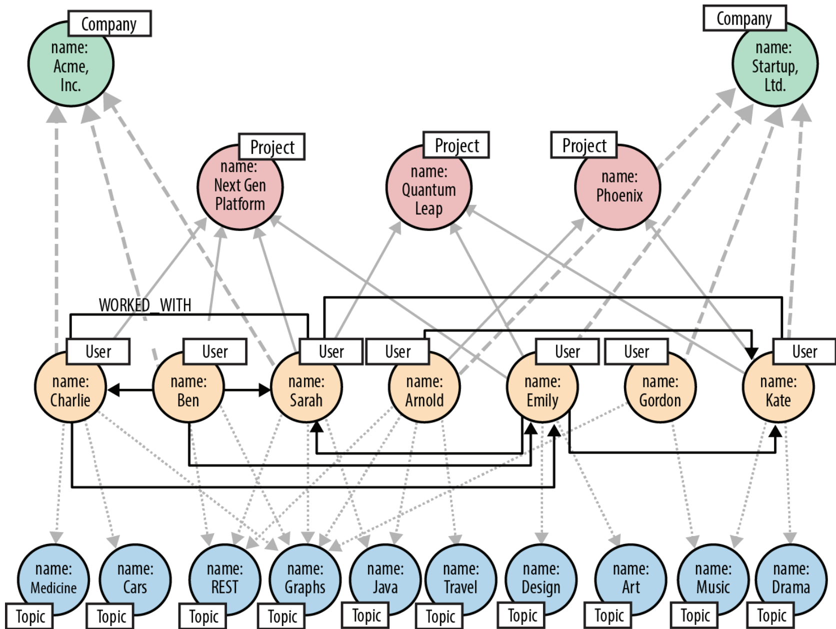


Рис. 5.7 ❖ Граф Talent.net с добавленными взаимосвязями WORKED_WITH

Авторизация и контроль доступа

TeleGraph Communications – это международная компания, предоставляющая услуги связи. Миллионы физических лиц и организаций пользуются ее услугами. В течение нескольких лет она предоставляла своим крупнейших клиентам возможность самим обслуживать свои учетные записи. С помощью приложения, работающего в браузере, администраторы этих организаций имели возможность добавлять и удалять услуги от имени своих сотрудников. Чтобы пользователи и администраторы имели возможность вносить изменения только в услуги, которыми они имеют право управлять, приложение использует сложную систему контроля доступа, которая назначает права доступа многим миллионам пользователей к десяткам миллионов экземпляров услуг.

Компания TeleGraph решила заменить существующую систему контроля доступа на решение, основанное на графовой базе данных. Решающими факторами при этом стали увеличение производительности и интерактивности.

Проблемы с производительностью преследовали приложение самообслуживания компании TeleGraph уже в течение нескольких лет. Существующая система была основана на реляционной базе данных с рекурсивными соединениями для моделирования сложных организационных структур и иерархии товаров, а также хранимые процедуры для реализации бизнес-логики управления доступом. Из-за интенсивного применения соединений, присущих модели данных, многие из важнейших запросов выполнялись с неприемлемо медленной скоростью. Для крупных компаний предоставление среды управления администратору занимало до нескольких минут. Это вызывало недовольство пользователей и препятствовало предоставлению услуг самообслуживания, приносящих доход.

Компания TeleGraph вынашивала амбициозные планы захвата новых регионов и рынков, позволяющие увеличить ее клиентскую базу на порядок. Но проблемы с производительностью в существующем приложении уже сегодня делали его использование невозможным, не говоря уже о будущих потребностях. Решение, основанное на графовой базе данных, напротив, обеспечивало производительность, масштабируемость и адаптивность, так необходимую для успешной работы в быстро меняющейся среде.

Модель данных компании TeleGraph

На рис. 5.8 изображен пример модели данных компании TeleGraph. (Для ясности: метки изображены над группами узлов, а не прикреплены к каждому из них. В действительности каждый узел снабжен, по крайней мере, одной меткой.)

Модель включает две иерархии. Первая иерархия объединяет администраторов из каждой организации в группы. Для групп задаются различные разрешения, согласно организационной структуре организаций:

- взаимосвязь `ALLOWED_INHERIT` соединяет административную группу с организационной единицей, отражая привилегию администраторов этой группы управлять данной организационной единицей. Эта привилегия наследуется дочерними узлами родительской организационной единицы. Пример наследования привилегий можно найти в модели данных TeleGraph, обратив внимание на взаимосвязь группы администраторов `Group 1` с организационной единицей `Acme`, имеющей дочерний узел `Spinoff`. Административная группа `Group 1` подключена к `Acme` с помощью взаимосвязи вида `ALLOWED_INHERIT`. Администратор

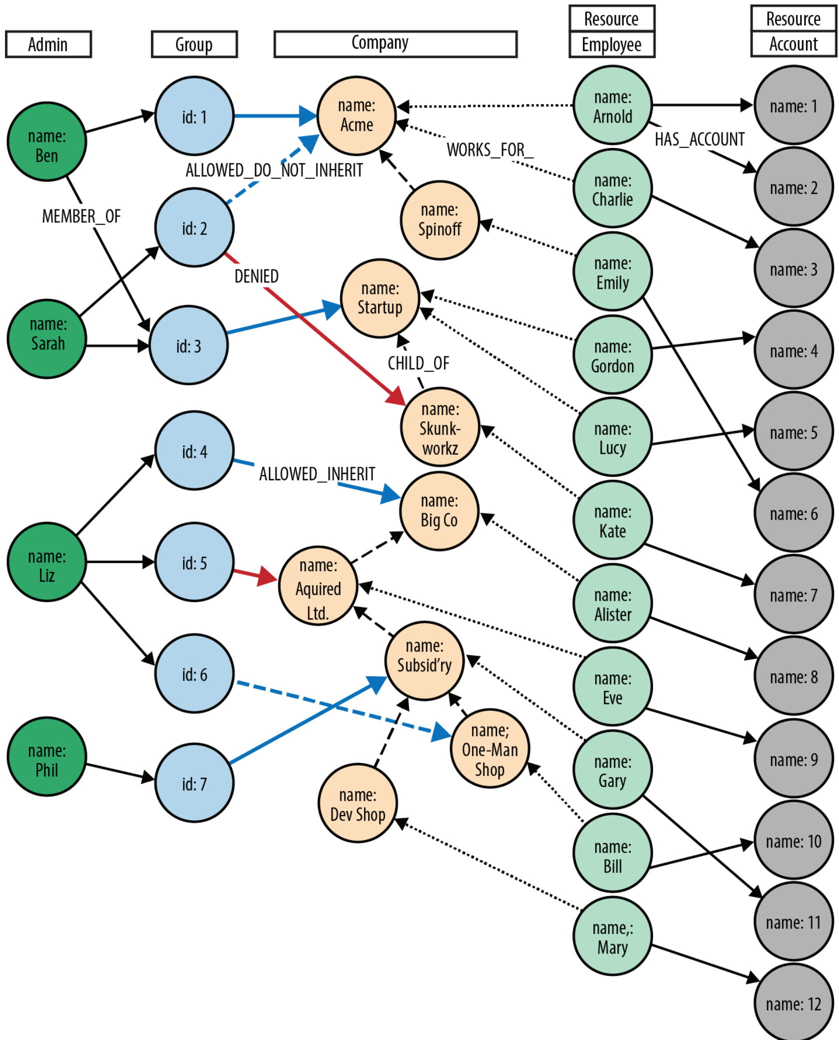


Рис. 5.8 ❖ Граф контроля доступа

Ben как член группы Group 1 может управлять сотрудниками Acme и Spinoff благодаря взаимосвязи ALLOWED_INHERIT;

- взаимосвязь ALLOWED_DO_NOT INHERIT соединяет административную группу с организационной единицей Acme, отражая тот факт, что администраторам этой группы разрешено управлять

только этой организационной единицей, но не ее дочерними организационными единицами. Администратор Sarah как член группы Group 2 может управлять организационной единицей Acme, но не ее дочерней организационной единицей Spinoff, потому что Group 2 подключена к Acme с помощью взаимосвязи ALLOWED_DO_NOT_INHERIT, а не взаимосвязи ALLOWED_INHERIT;

- взаимосвязь DENIED указывает на запрет доступа администраторам к организационным единицам. Этот запрет наследуется дочерними организационными единицами от родительской. На схеме TeleGraph эта взаимосвязь лучше всего видна на примере администратора Liz и его разрешений в отношении организационных единиц Big Co, Acquired Ltd, Subsidiary и One-Map Shop. В результате членства администратора Liz в группе Group 4 и его взаимосвязи вида ALLOWED_INHERIT с организационной единицей Big Co администратор Liz может управлять организационной единицей Big Co. Но, несмотря на эту обеспечивающую наследование взаимосвязь, администратор Liz не может управлять организационными единицами Acquired Ltd и Subsidiary, потому что группе Group 5, членом которой является администратор Liz, отказано в доступе (взаимосвязь DENIED) к организационной единице Acquired Ltd и ее дочерним организационным единицам (включающим в себя Subsidiary). Но администратор Liz может управлять организационной единицей One-Map Shop благодаря разрешению ALLOWED_DO_NOT_INHERIT, предоставляемому членством в группе Group 6, последней группе, к которой принадлежит администратор Liz.

Взаимосвязь DENIED имеет приоритет над взаимосвязью ALLOWED_INHERIT, но находится в подчиненном положении относительно взаимосвязи ALLOWED_DO_NOT_INHERIT. То есть если администратор связан с организационной единицей взаимосвязями ALLOWED_DO_NOT_INHERIT и DENIED, предпочтение отдается взаимосвязи ALLOWED_DO_NOT_INHERIT.

Подробные имена взаимосвязей или взаимосвязи со свойствами?

Обратите внимание, что модель данных контроля доступа компании TeleGraph использует подробные имена взаимосвязей (ALLOWED_INHERIT, ALLOWED_DO_NOT_INHERIT и DENIED), а не один тип взаимосвязей, например PERMISSION, с уточняющими булевыми свойствами, определяющими разрешение и наследование. Разработчики TeleGraph проверили оба подхода и определили, что использование взаимосвязей с подробными именами в два раза увеличивает быстродействие, по сравнению с использованием свойств. Более подробно проектирование взаимосвязей описано в главе 4.

Поиск доступных администратору ресурсов

Приложение TeleGraph использует множество различных Cypher-запросов. Рассмотрим некоторые из них.

Начнем с поиска всех ресурсов, к которым администратор может получить доступ. При входе администратора в систему ему представляется список всех сотрудников и их учетных записей, которыми он может управлять. Этот список формируется на основании результатов, возвращенных следующим запросом:

```
MATCH (admin:Admin {name:{adminName}})
MATCH paths=(admin)-[:MEMBER_OF]->(:Group)-[:ALLOWED_INHERIT]->(:Company)
          <-[:CHILD_OF*0..3]-(company:Company)<-[:WORKS_FOR]-(employee:Employee)
          -[:HAS_ACCOUNT]->(account:Account)
WHERE NOT ((admin)-[:MEMBER_OF]->(:Group)
        -[:DENIED]->(:Company)<-[:CHILD_OF*0..3]-(company))
RETURN employee.name AS employee, account.name AS account
UNION
MATCH (admin:Admin {name:{adminName}})
MATCH paths=(admin)-[:MEMBER_OF]->(:Group)-[:ALLOWED_DO_NOT_INHERIT]->(:Company)
          <-[:WORKS_FOR]-(employee:Employee)-[:HAS_ACCOUNT]->(account:Account)
RETURN employee.name AS employee, account.name AS account
```

Как и все другие запросы в этом разделе, этот запрос включает в себя два отдельных запроса, объединенных оператором `UNION`. Запрос перед оператором `UNION` обрабатывает взаимосвязи `ALLOWED_INHERIT`, не конфликтующие со взаимосвязями `DENIED`. Запрос после оператора `UNION` обрабатывает все взаимосвязи `ALLOWED_DO_NOT_INHERIT`. Такой шаблон – `ALLOWED_INHERIT` минус `DENIED`, затем плюс `ALLOWED_DO_NOT_INHERIT` – повторяется во всех приведенных примерах запросов управления доступом.

Первый запрос перед оператором `UNION` можно разбить на части следующим образом:

- первая фраза `MATCH` выбирает определенного администратора из узлов с метками `Administrator` и связывает результат с идентификатором `admin`;
- фраза `MATCH` определяет все группы, к которым принадлежит заданный администратор, а затем на основании этих групп все родительские организации, присоединенные с помощью взаимосвязей `ALLOWED_INHERIT`. Далее фраза `MATCH` использует маршрут переменной длины (`[:CHILD_OF*0..3]`) для нахождения всех дочерних организационных единиц этих организаций, сотрудников и их учетных записей, присоединенных ко всем орга-

низационным единицам (как родительским, так и дочерним). В этой точке запрос находит все соответствия организационных единиц, сотрудников и учетных записей, доступных через взаимосвязи ALLOWED_INHERIT;
- фраза WHERE отсеивает совпадения, в которых организационная единица company или родительская по отношению к ней организационная единица подсоединена с помощью взаимосвязи DENIED к группе администраторов. Эта фраза WHERE выполняется для каждого совпадения. Если где-то между узлом admin и узлом company существует взаимосвязь DENIED, совпадение отбрасывается;
- фраза RETURN создает проекцию выбранных данных в виде списка имен сотрудников и их учетных записей.

Второй запрос после оператора UNION несколько проще:
- первая фраза MATCH выбирает определенного администратора из узлов с метками Administrator и связывает результат с идентификатором admin;
- вторая фраза MATCH просто выбирает организации (плюс сотрудников и их учетные записи), непосредственно связанные с группами администраторов посредством взаимосвязей ALLOWED_DO_NOT_INHERIT.

Оператор UNION объединяет результаты этих двух запросов вместе, исключая дубликаты. Обратите внимание, что фраза RETURN в обоих запросах должна содержать одинаковую проекцию результатов. Другими словами, имена столбцов двух результирующих наборов должны совпадать.

Рисунок 5.9 демонстрирует, как этот запрос отбирает все ресурсы, доступные пользователю Sarah, в виде графа приложения TeleGraph. Обратите внимание, что из-за взаимосвязи DENIED между группой администраторов Group 2 и организационной единицей Skunkworkz администратор Sarah не может управлять пользователем Kate и ее учетной записью Account 7.

Язык запросов Cypher поддерживает операторы UNION и UNION ALL. Оператор UNION удаляет дубликаты в окончательном наборе результатов, а оператор UNION ALL включает в набор результатов все дубликаты.

Определение доступности ресурса администратору

Рассмотренный выше запрос возвращает список сотрудников и их учетных записей, которыми может управлять выбранный админист-

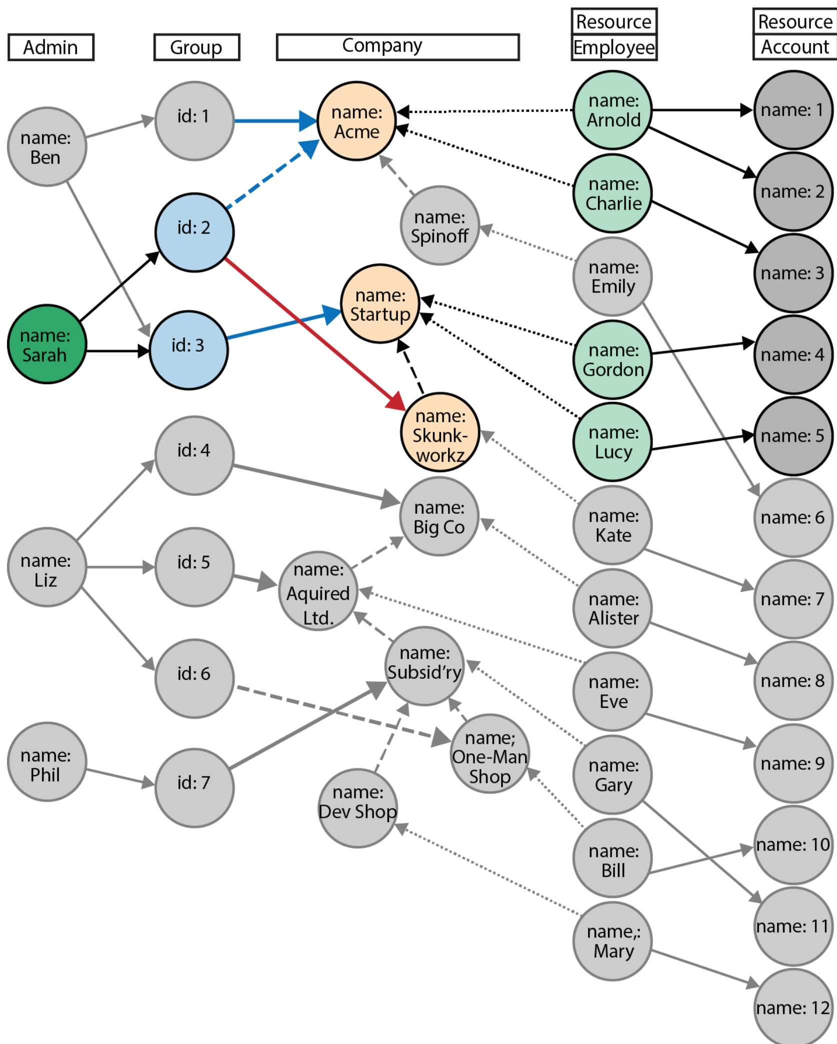


Рис. 5.9 ❖ Поиск доступных администратору ресурсов

ратор. В веб-приложении каждый из этих ресурсов (сотрудник, учетная запись) доступен через свой URI-адрес. Учитывая дружелюбный вид URI-адресов (например, http://TeleGraph/accounts/5436), ничто не помешает злоумышленнику взломать URI-адрес и получить незаконный доступ к учетной записи.

Итак, нужен запрос, который позволит определить, имеет ли администратор доступ к определенному ресурсу. Вот этот запрос:

```
MATCH (admin:Admin {name:{adminName}}),
      (company:Company)-[:WORKS_FOR|HAS_ACCOUNT*1..2]
        -(resource:Resource {name:{resourceName}})
MATCH p=(admin)-[:MEMBER_OF]->(:Group)-[:ALLOWED_INHERIT]->(:Company)
        <-[:CHILD_OF*0..3]-(company)
WHERE NOT ((admin)-[:MEMBER_OF]->(:Group)-[:DENIED]->(:Company)
            <-[:CHILD_OF*0..3]-(company))
RETURN count(p) AS accessCount
UNION
MATCH (admin:Admin {name:{adminName}}),
      (company:Company)-[:WORKS_FOR|HAS_ACCOUNT*1..2]
        -(resource:Resource {name:{resourceName}})
MATCH p=(admin)-[:MEMBER_OF]->()-[:ALLOWED_DO_NOT_INHERIT]->(company)
RETURN count(p) AS accessCount
```

Он определяет, имеет ли администратор доступ к организационной единице, к которой принадлежит сотрудник или его учетная запись. На основании сотрудника или учетной записи определяется компания, с которой связан этот ресурс, а затем проверяется – имеет ли администратор доступ к ресурсам этой компании.

Как идентифицировать компанию по сотруднику или учетной записи? И те, и другие снабжены метками `Resource` (а также метками `Company` или `Account`). Сотрудники подключены к ресурсу компаний `Company` с помощью взаимосвязей `WORKS_FOR`. Учетные записи связаны с компаниями через сотрудников. Взаимосвязи `HAS_ACCOUNT` соединяют сотрудников с учетными записями. Взаимосвязи `WORKS_FOR` связывают сотрудников с компаниями. Иными словами, работник удален на один шаг от компании, в то время как учетная запись отстоит от компании на два шага.

Несложно догадаться, что проверка права доступа к ресурсу похожа на запрос для поиска всех компаний, сотрудников и учетных записей с некоторыми небольшими различиями:

- первая фраза `MATCH` находит компанию, с которой связан сотрудник или учетная запись. Она использует оператор ИЛИ (`|`) для выявления одной или двух взаимосвязей `WORKS_FOR` или `HAS_ACCOUNT`;
- фраза `WHERE` в запросе перед оператором `UNION` отсеивает соответствия, где компания связана с одним из администраторов группы посредством взаимосвязи `DENIED`;
- фразы `RETURN` в запросе до и после оператора `UNION` возвращают подсчет количества совпадений. Чтобы администратор имел

доступ к ресурсу, хотя бы одно из этих значений accessCount должно быть больше 0.

Поскольку оператор UNION устраняет дубликаты в результатах, полный набор результатов запроса может содержать одно или два значения. Алгоритм определения доступности ресурса на стороне клиента легко можно выразить на Java:

```
private boolean isAuthorized( Result result )
{
    Iterator<Long> accessCountIterator = result.columnAs( "accessCount" );
    while ( accessCountIterator.hasNext() )
    {
        if (accessCountIterator.next() > 0L)
        {
            return true;
        }
    }
    return false;
}
```

Поиск администраторов по учетной записи

Предыдущие два запроса представляют собой обзор графа «сверху вниз». Последний запрос из приведенных здесь реализует обзор данных «снизу вверх». Он отвечает на вопрос: кто может управлять ресурсом, т. е. сотрудником или учетной записью? Вот этот запрос:

```
MATCH (resource:Resource {name:{resourceName}})
MATCH  p=(resource)-[:WORKS_FOR|HAS_ACCOUNT*1..2]-(company:Company)
      -[:CHILD_OF*0..3]->()<-[:ALLOWED_INHERIT]-()<-[:MEMBER_OF]-(admin:Admin)
WHERE NOT ((admin)-[:MEMBER_OF]->(:Group)-[:DENIED]->(:Company)
         <-[:CHILD_OF*0..3]-(company))
RETURN admin.name AS admin
UNION
MATCH (resource:Resource {name:{resourceName}})
MATCH  p=(resource)-[:WORKS_FOR|HAS_ACCOUNT*1..2]-(company:Company)
        <-[:ALLOWED_DO_NOT_INHERIT]-(:Group)<-[:MEMBER_OF]-(admin:Admin)
RETURN admin.name AS admin
```

Как и предыдущие запросы, этот так же состоит из двух независимых запросов, соединенных оператором UNION. Особо следует отметить следующие моменты:

- первая фраза MATCH использует метку Resource, что позволяет ей идентифицировать как сотрудников, так и учетные записи;
- вторая фраза MATCH содержит выражение маршрута переменной длины, которое с помощью оператора | определяет маршрут,

содержащий одну или две взаимосвязи вида `WORKS_FOR` или `HAS_ACCOUNT`. Это выражение учитывает тот факт, что субъектом запроса может быть либо сотрудник, либо учетная запись.

На рис. 5.10 изображена часть графа, отобранная в запросе при поиске администраторов для учетной записи `Account 10`.

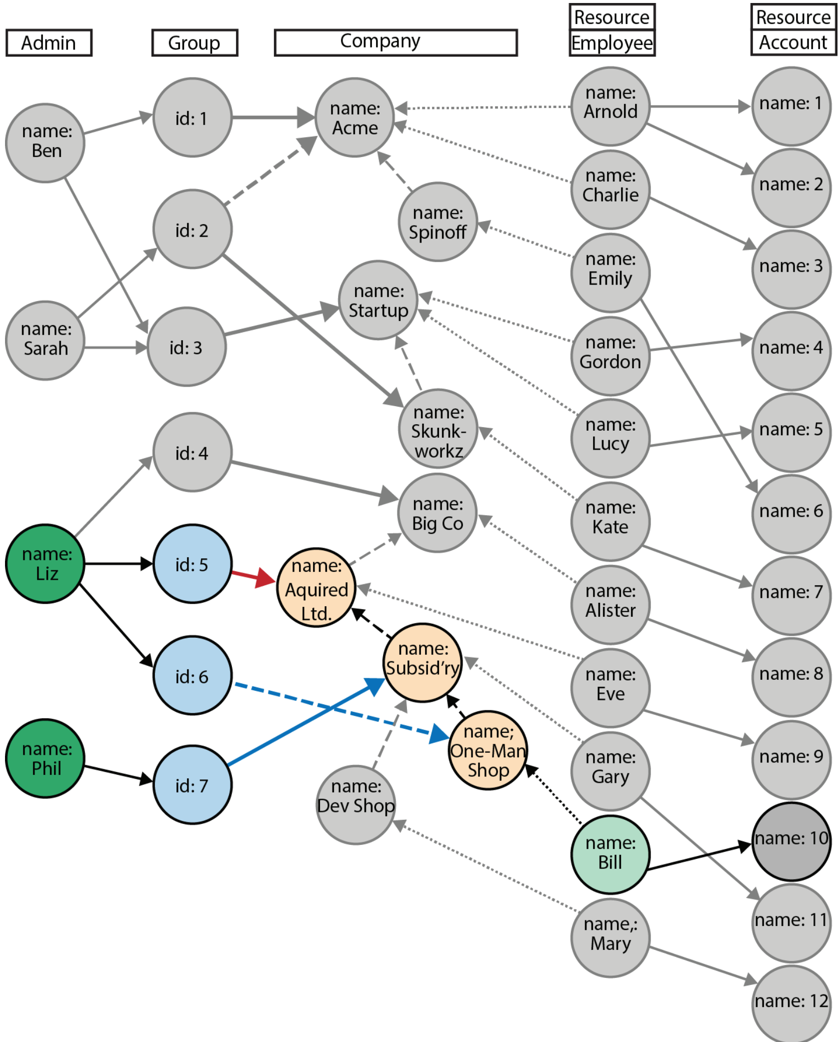


Рис. 5.10 ❖ Поиск администратора для определенной учетной записи

Геоинформационные системы и логистика

Компания Global Post – это международная служба доставки, ежедневно доставляющая миллионы посылок по более чем 30 миллионам адресов. В последние годы из-за роста числа интернет-магазинов количество отправлений значительно увеличилось. В настоящее время на долю Amazon и eBay приходится более половины посылок, доставляемых Global Post.

С учетом роста числа посылок и в условиях жесткой конкуренции с другими службами доставки Global Post решила реализовать программу изменения всех аспектов сети доставки посылок, в том числе обновление зданий, оборудования, систем и процессов.

Одним из самых важных и критичных к быстродействию компонентов в сети доставки посылок является средство прокладки маршрутов. В сеть каждую секунду поступает от одной до трех тысяч посылок. После поступления посылок в сеть они механически сортируются в соответствии с их пунктом назначения. Для поддержания требуемого темпа этого процесса средство прокладывания маршрута должно рассчитать маршрут посылки до того, как она достигнет точки, где сортировочное оборудование должно сделать выбор, что составляет всего несколько секунд после поступления посылки в сеть, отсюда и такие строгие требования к прокладыванию маршрутов.

Необходимо не только, чтобы прокладывание маршрута посылки занимало миллисекунды, но оно должно быть сделано в соответствии с маршрутами, запланированными на конкретный период времени. Маршруты посылок постоянно меняются и зависят от количества задействованного транспорта, персонала и загрузки, которая, например, на Рождество значительно отличается от загрузки летом. Средство прокладывания маршрутов должно, следовательно, согласовывать свои расчеты с доступностью маршрутов в определенный период времени.

И вместе с учетом разных маршрутов и уровней трафика посылок новая сеть доставки должна также предусматривать возможность внесений существенных изменений и значительного расширения. Платформа, закладываемая сегодня, должна стать основой деятельности компании Global Post на ближайшие 10 и более лет. В течение этого времени компания ожидает обновления больших участков сети, в том числе оборудования, помещений и транспортных маршрутов, для соответствия изменениям во внешней среде. Модель данных, лежащая в основе средства прокладывания маршрутов, должна, следовательно, предусматривать быструю и значительную эволюцию схем.

Модель данных Global Post

На рис. 5.11 приведен упрощенный пример модели сети доставки посылок Global Post. Сеть включает центры обработки посылок, которые соединены с базами доставки, каждая из которых охватывает несколько областей доставки. Области доставки, в свою очередь, подразделяются на сегменты, охватывающие множество мест доставки. Существует около 25 национальных центров обработки посылок и примерно 2 миллиона мест доставки (соответствующих почтовым индексам).

С течением времени маршруты доставки меняются. На рис. 5.12, 5.13 и 5.14 приведены три различные схемы для разных периодов вре-

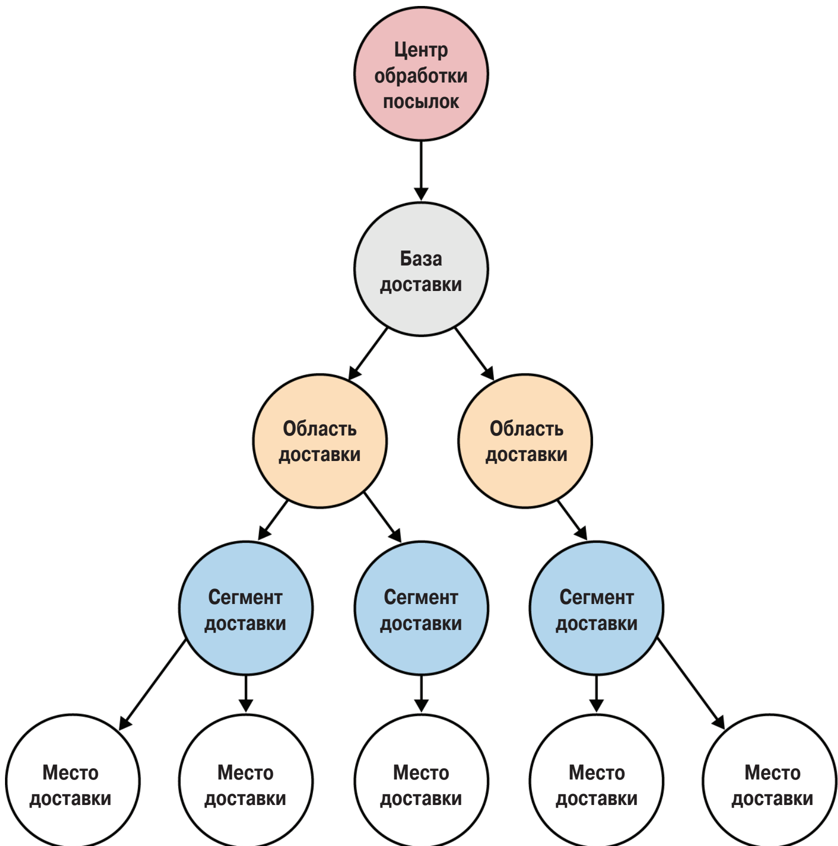


Рис. 5.11 ❖ Элементы в сети Global Post

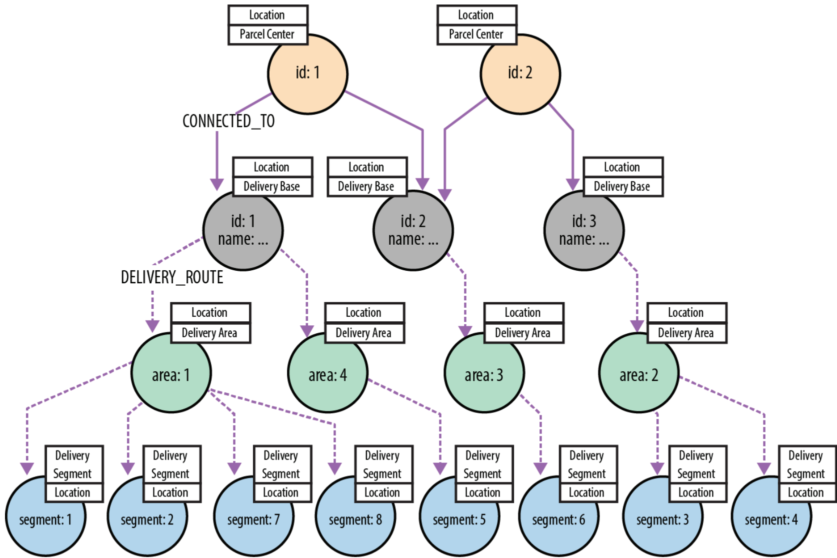


Рис. 5.12 ❖ Структура сети доставки для периода 1

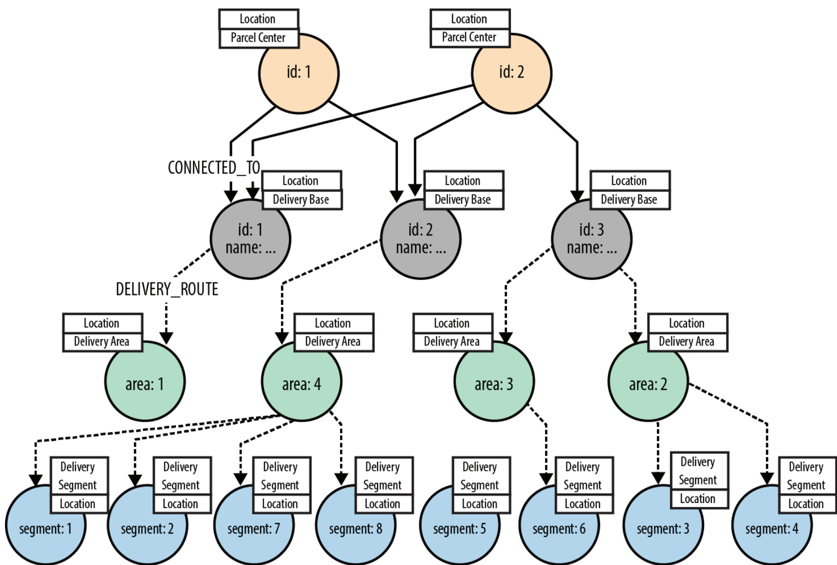


Рис. 5.13 ❖ Структура сети доставки для периода 2

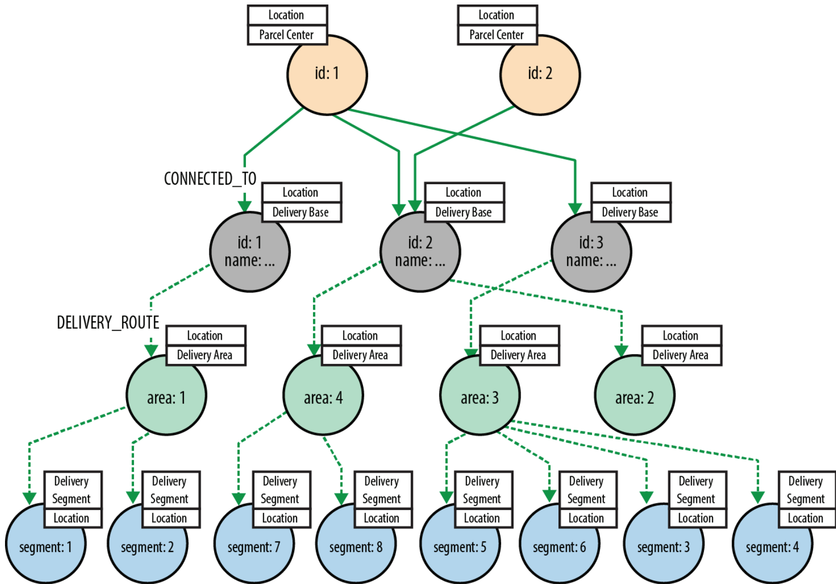


Рис. 5.14 ❖ Структура сети доставки для периода 3

мени. Для любого выбранного периода существует не более одного маршрута от базы доставки до любой области доставки или сегмента доставки. Но имеется несколько маршрутов между базами доставки посылок и центрами обработки посылок. Поэтому для любого заданного момента времени нижние части графа (отдельные подграфы ниже баз доставки) представляют собой простые древовидные структуры, в то время как верхние части графа, состоящие из баз доставки и центров обработки посылок, в большей мере взаимосвязаны.

Обратите внимание, что места доставки не включены в модель данных. Это связано с тем, что каждое место доставки всегда связано с одним и тем же сегментом независимо от текущего периода. Такая инвариантность делает возможным использовать один сегмент вместо нескольких мест доставки. Для прокладки маршрута к определенному месту доставки системе достаточно лишь проложить маршрут к соответствующему сегменту, название которого можно извлечь из почтового индекса, указанного для места доставки. Такая оптимизация позволяет уменьшить размер графа и количество проходов при расчете маршрутов.

Действующая база данных содержит информацию о различных периодах доставки. Как показано на рис. 5.15, присутствие взаимосвязей для нескольких периодов делает граф весьма тесно связанным.

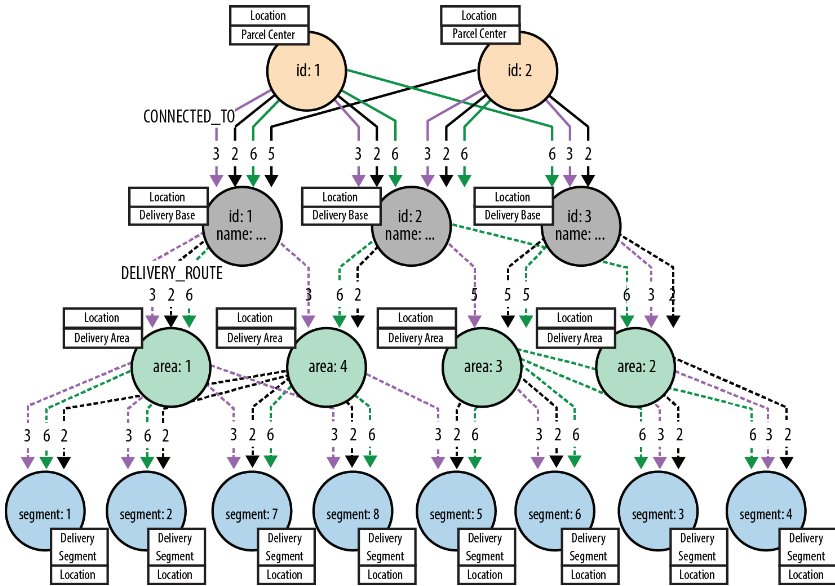


Рис. 5.15 ❖ Пример сети Global Post

В действующей модели данных узлы соединены несколькими взаимосвязями, каждая из которых снабжена свойствами `start_date` (начало) и `end_date` (окончание), содержащими дату и время. Используются взаимосвязи двух типов – `CONNECTED_TO`, соединяющие центры обработки посылок с базами доставки, и `DELIVERY_ROUTE`, соединяющие базы доставки с областями доставки и области доставки с сегментами доставки. Применение двух видов взаимосвязей позволяет разделить граф на верхнюю и нижнюю части, что обеспечивает высокую эффективность прокладки маршрутов. На рис. 5.16 приведены три ограниченные временными рамками взаимосвязи `CONNECTED_TO`, соединяющие центр обработки посылок с базой доставки.

Расчет маршрута

Как уже упоминалось в предыдущем разделе, взаимосвязи `CONNECTED_TO` и `DELIVERY_ROUTE` разбивают граф на *верхнюю* и *нижнюю*

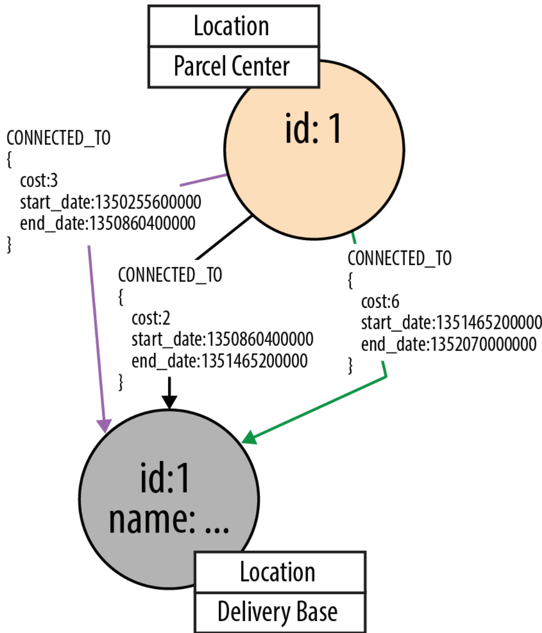


Рис. 5.16 ❖ Свойства временных отрезков взаимосвязей

части. Верхняя часть состоит из сложно связанных между собой центров обработки посылок и центров доставки. Нижняя – включает базы, области и сегменты доставки, причем эта часть в любой период времени является обычной древовидной структурой.

Расчет маршрута включает поиск кратчайшего пути между двумя точками в нижней части графа. Начальной точкой обычно является сегмент или область доставки, а конечной – всегда сегмент. Сегмент доставки, как описано выше, несложно определить по месту доставки. Независимо от расположения начальной и конечной точек проложенный маршрут обязательно должен проходить, по меньшей мере, через один центр обработки посылок из верхней части графа.

Если рассматривать расчет как проход графа, его можно разделить на три шага. Шаги 1 и 2, показанные на рис. 5.17, представляют собой направленные вверх перемещения от начальной и конечной точек соответственно до баз доставки. Поскольку для любого периода существует не более одного маршрута между любыми двумя элемен-

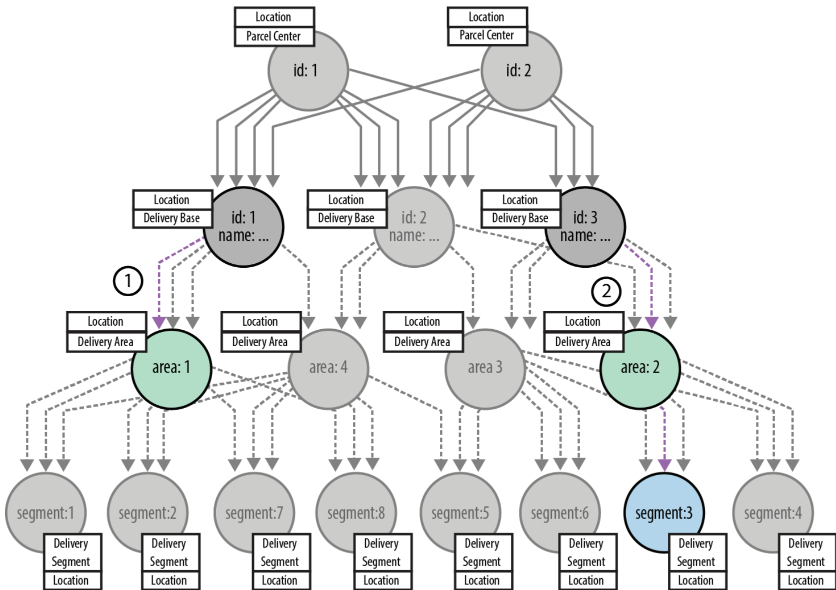


Рис. 5.17 ❖ Кратчайший путь к базам доставки из начальной и конечной точек

тами в нижней части графа, прокладка маршрута от одного элемента к другому сводится к простому поиску исходящих взаимосвязей `DELIVERY ROUTE`, временные границы которых *охватывают* текущий период доставки. Выполнение шагов 1 и 2, состоящее в одном или двух перемещениях по паре ветвей древовидной структуры, приводит к двум разным базам доставки. Эти две базы доставки являются начальной и конечной точками шага 3, проходящего через верхнюю часть графа.

В шаге 3, изображенном на рис. 5.18, так же как в шагах 1 и 2, выполняется поиск взаимосвязей, но на этот раз взаимосвязей `CONNECTED_TO`, чьи временные границы *охватывают* текущий период доставки. Даже при наличии фильтрации по периоду времени потенциально может существовать несколько маршрутов, соединяющих две любые базы доставки в верхней части графа. Следовательно, в шаге 3 следует оценить протяженность каждого из маршрутов и выбрать самый короткий из них, что делает его *расчетом кратчайшего пути*.

Чтобы получить полный маршрут от начальной точки до конечной точки, нужно просто объединить шаги 1, 2 и 3.

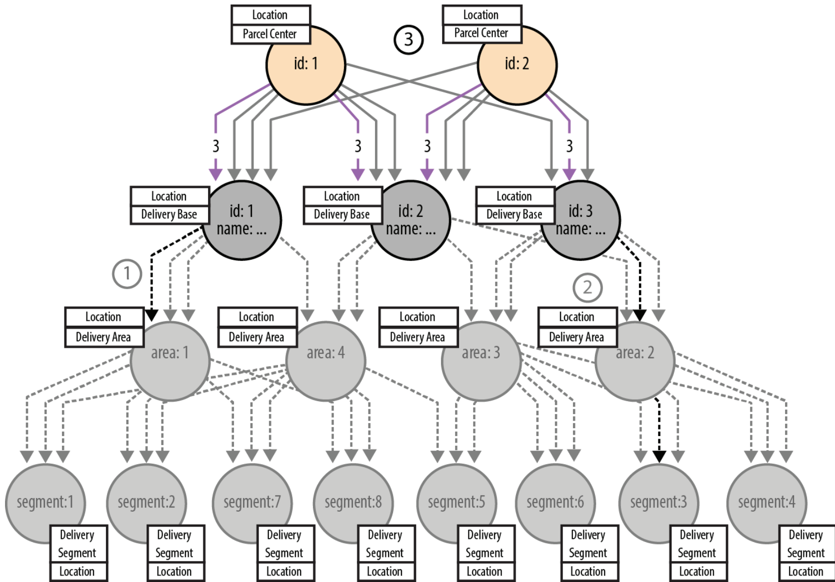


Рис. 5.18 ❖ Кратчайший путь между базами доставки

Поиск кратчайшего маршрута с помощью Cypher

Ниже приводится Cypher-запрос для расчета маршрута доставки:

```
MATCH (s:Location {name:{startLocation}}),
      (e:Location {name:{endLocation}})
MATCH upLeg = (s)<-[:DELIVERY_ROUTE*1..2]-(db1)
WHERE all(r in relationships(upLeg)
          WHERE r.start_date <= {intervalStart}
          AND r.end_date >= {intervalEnd})
WITH e, upLeg, db1
MATCH downLeg = (db2)-[:DELIVERY_ROUTE*1..2]->(e)
WHERE all(r in relationships(downLeg)
          WHERE r.start_date <= {intervalStart}
          AND r.end_date >= {intervalEnd})
WITH db1, db2, upLeg, downLeg
MATCH topRoute = (db1)<-[:CONNECTED_TO]-()-[:CONNECTED_TO*1..3]-(db2)
WHERE all(r in relationships(topRoute)
          WHERE r.start_date <= {intervalStart}
          AND r.end_date >= {intervalEnd})
WITH upLeg, downLeg, topRoute,
     reduce(weight=0, r in relationships(topRoute) | weight+r.cost) AS score
     ORDER BY score ASC
     LIMIT 1
RETURN (nodes(upLeg) + tail(nodes(topRoute)) + tail(nodes(downLeg))) AS n
```

На первый взгляд запрос кажется довольно сложным. Однако он состоит из четырех более простых запросов, соединенных фразами WITH. Рассмотрим каждый из них по отдельности.

Первый подзапрос:

```
MATCH (s:Location {name:{startLocation}}),
      (e:Location {name:{endLocation}})
MATCH upLeg = (s)<-[:DELIVERY_ROUTE*1..2]-(db1)
WHERE all(r in relationships(upLeg)
          WHERE r.start_date <= {intervalStart}
          AND r.end_date >= {intervalEnd})
```

Этот запрос выполняет расчет шага 1-го маршрута. Его можно разбить на следующие части:

- первая фраза MATCH находит начальную и конечную точки в подгруппе узлов с метками Location, присоединяя их к идентификаторам sи e соответственно;
- вторая фраза MATCH находит маршрут от начальной точки s к базе доставки с помощью определения пути, состоящего из переменного числа взаимосвязей DELIVERY_ROUTE. Этому маршруту присваивается идентификатор upLeg. Так как базы доставки всегда являются корневыми узлами деревьев взаимосвязей DELIVERY_ROUTE и, следовательно, не имеют входящих взаимосвязей DELIVERY_ROUTE, узел db1 в конце такого пути переменной длины всегда представляет базу доставки, а не другой сетевой элемент;
- фраза WHERE накладывает дополнительные ограничения на путь upLeg, гарантируя, что маршрут доставки будет состоять только из взаимосвязей DELIVERY_ROUTE, чьи свойства start_date и end_date охватывают заданный период доставки.

Второй подзапрос выполняет расчет шага 2-го маршрута и определяет путь от конечной точки вверх до базы доставки, чье дерево взаимосвязей DELIVERY_ROUTE включает конечную точку в качестве подчиненного узла. Этот запрос очень похож на первый:

```
WITH e, upLeg, db1
MATCH downLeg = (db2)-[:DELIVERY_ROUTE*1..2]->(e)
WHERE all(r in relationships(downLeg)
          WHERE r.start_date <= {intervalStart}
          AND r.end_date >= {intervalEnd})
```

Фраза WITH здесь связывает первый подзапрос со вторым, передавая конечную точку маршрута шага 1 и базу во второй подзапрос.

Второй подзапрос использует во фразе MATCH только конечную точку e, прочие входные данные будут переданы в следующие подзапросы.

Третий подзапрос выявляет *все* пути для шага 3-го маршрута, т. е. пути между базами доставки db1и db2:

```
WITH db1, db2, upLeg, downLeg
MATCH topRoute = (db1)<-[:CONNECTED_TO]-()-[:CONNECTED_TO*1..3]-(db2)
WHERE all(r in relationships(topRoute)
          WHERE r.start_date <= {intervalStart}
          AND r.end_date >= {intervalEnd})
```

Этот подзапрос можно разбить на следующие составные части:

- фраза WITH присоединяет этот подзапрос к предыдущему, передавая базы доставки db1 и db2 вместе с путями, найденными при выполнении шагов 1 и 2 в текущем запросе;
- фраза MATCH определяет *все* пути между первой и второй базами доставки, состоящие не более чем из четырех взаимосвязей, и присваивает им идентификатор topRoute;
- фраза WHERE отсеивает пути topRoute, чьи свойства start_date и end_date не охватывают заданного периода доставки.

Четвертый и последний подзапрос выбирает кратчайший маршрут для шага 3 и вычисляет полный маршрут:

```
WITH upLeg, downLeg, topRoute,
     reduce(weight=0, r in relationships(topRoute) | weight+r.cost) AS score
     ORDER BY score ASC
     LIMIT 1
RETURN (nodes(upLeg) + tail(nodes(topRoute)) + tail(nodes(downLeg))) AS n
```

Это подзапрос работает следующим образом:

- фраза WITH передает одну или несколько троек, включающих пути upLeg, downLeg и topRoute, в текущей подзапрос. Количество таких троек равно количеству маршрутов, полученных третьим подзапросом (маршруты, связанные с идентификаторами upLeg и downLeg, будут одинаковыми во всех тройках, так как первый и второй подзапросы вернут только один маршрут). Каждой тройке соответствует оценка пути topRoute в этой тройке. Оценка рассчитывается с помощью Cypher-функции reduce, которая для каждой тройки суммирует значения свойств cost взаимосвязей, составляющих маршрут topRoute. Затем тройки сортируются по полученным значениям score в порядке возрастания, и в отсортированном списке сохраняется только первая тройка;

- фраза `RETURN` собирает вместе узлы путей `upLeg`, `topRoute` и `downLeg` для получения окончательного результата. Функция `tail` отбрасывает первые узлы маршрутов `topRoute` и `downLeg`, так как их уже содержат предыдущие пути.

Реализация расчета маршрутов с помощью фреймворка Traversal

Временные характеристики прокладки маршрутов крайне важны, и это накладывает жесткие требования на инструмент расчета маршрутов. Так как выполнение единичного запроса не занимает много времени, всегда можно горизонтально распределить нагрузку для увеличения пропускной способности. Решение на языке Cypher работает быстро, но несколько тысяч посылок, поступающих в сеть каждую секунду или даже каждую миллисекунду, создают значительную нагрузку на кластер. Поэтому разработчики Global Post решили применить альтернативный подход, реализовав расчет маршрутов с помощью фреймворка Traversal базы данных Neo4j.

При реализации прокладки маршрутов, основанной на фреймворке Traversal, необходимо решить две задачи: поиск кратчайших путей и фильтрацию путей в зависимости от заданного периода времени. Начнем рассмотрение с фильтрации путей в зависимости от периода времени.

Маршруты должны включать взаимосвязи, доступные в течение указанного периода доставки. Другими словами, при обходе графа нужно использовать только взаимосвязи, соответствующие указанному периоду, чьи свойства `start_date` и `end_date` охватывают указанный период доставки.

Такая фильтрация взаимосвязей реализуется с помощью класса `PathExpander`. Для заданного пути от начального узла обхода к текущему метод `expand()` класса `PathExpander` возвращает взаимосвязи, которые можно использовать для продолжения обхода. Этот метод вызывается фреймворком Traversal для перемещения в каждый следующий узел графа. При необходимости клиент может предоставить некое начальное состояние, называемое состоянием ветви, при выполнении обхода. Метод `expand()` может использовать (и даже изменять) состояние ветви, выбирая, какие взаимосвязи вернуть. Реализация калькулятора маршрутов `ValidPathExpander` использует эти состояния ветвей для учета периода доставки.

Ниже приводится код класса ValidPathExpander:

```
private static class ValidPathExpander implements PathExpander<Interval>
{
  private final RelationshipType relationshipType;
  private final Direction direction;
  private ValidPathExpander( RelationshipType relationshipType,
                             Direction direction )
  {
    this.relationshipType = relationshipType;
    this.direction = direction;
  }

  @Override
  public Iterable<Relationship> expand( Path path,
                                        BranchState<Interval> deliveryInterval )
  {
    List<Relationship> results = new ArrayList<Relationship>();
    for ( Relationship r : path.endNode()
                               .getRelationships( relationshipType, direction ) )
    {
      Interval relationshipInterval = new Interval(
         (Long) r.getProperty( "start_date" ),
         (Long) r.getProperty( "end_date" ) );
      if ( relationshipInterval.contains( deliveryInterval.getState() ) )
      {
        results.add( r );
      }
    }
    return results;
  }
}
```

Конструктор класса ValidPathExpander принимает два аргумента: relationshipType и direction. Это позволяет использовать класс для разных типов взаимосвязей. В случае с графом Global Post класс используется для фильтрации взаимосвязей CONNECTED_TOи DELIVERY_ROUTE.

Метод expand() принимает путь path от начального узла к текущему, где остановился обход, и состояние ветви deliveryInterval, заданное клиентом. В каждом вызове метод expand() выполняет перебор соответствующих взаимосвязей текущего узла (текущий узел определяется с помощью метода path.endNode()) и сравнивает интервал взаимосвязи с заданным интервалом доставки. Если интервал взаимосвязи содержит интервал доставки, взаимосвязь добавляется

в результат.

Познакомившись с классом ValidPathExpander, можно переходить к классу реализации самого калькулятора маршрутов ParcelRouteCalculator. Этот класс инкапсулирует всю логику вычисления маршрута от места поступления посылки в сеть до конечного пункта доставки. Он использует тот же подход, что и рассмотренный выше Cypher-запрос. То есть рассчитывает два пути от начального и от конечного узлов до двух разных баз доставки. А затем находит кратчайший путь между этими двумя базами.

Ниже приводится первая часть кода класса ParcelRouteCalculator:

```
public class ParcelRouteCalculator
{
    private static final PathExpander<Interval> DELIVERY_ROUTE_EXPANDER =
       new ValidPathExpander( withName( "DELIVERY_ROUTE" ),
                              Direction.INCOMING );
    private static final PathExpander<Interval> CONNECTED_TO_EXPANDER =
       new ValidPathExpander( withName( "CONNECTED_TO" ),
                              Direction.BOTH );
    private static final TraversalDescription DELIVERY_BASE_FINDER =
        Traversal.description()
            .depthFirst()
            .evaluator( new Evaluator()
            {
            private final RelationshipType DELIVERY_ROUTE =
                withName( "DELIVERY_ROUTE");

            @Override
            public Evaluation evaluate( Path path )
            {
                if ( isDeliveryBase( path ) )
                {
                    return Evaluation.INCLUDE_AND_PRUNE;
                }
                return Evaluation.EXCLUDE_AND_CONTINUE;
            }

            private boolean isDeliveryBase( Path path )
            {
                return !path.endNode().hasRelationship(
                    DELIVERY_ROUTE, Direction.INCOMING );
            }
        } );
    private static final CostEvaluator<Double> COST_EVALUATOR =
        CommonEvaluators.doubleCostEvaluator( "cost" );
```

```
    public static final Label LOCATION = DynamicLabel.label("Location");
    private GraphDatabaseService db;

    public ParcelRouteCalculator( GraphDatabaseService db )
    {
        this.db = db;
    }
    ...
}
```

Здесь определены два класса `PathExpander` – один для взаимосвязей `DELIVERY_ROUTE` и один для взаимосвязей `CONNECTED_TO` – и класс `TraversalDescription` для поиска двух шагов маршрута. Поиск заканчивается, когда встречается узел, не имеющий входящих взаимосвязей `DELIVERY_ROUTE`. Так как базы доставки расположены в корне деревьев маршрутов доставки, можно утверждать, что узел без входящих взаимосвязей `DELIVERY_ROUTE` представляет базу доставки.

Каждый инструмент расчета маршрутов создает экземпляр этого калькулятора. Экземпляр может обслуживать несколько запросов. Для расчета каждого маршрута клиент вызывает метод калькулятора `calculateRoute()`, передает ему названия начальной и конечной точек и период, для которого рассчитывается маршрут:

```
public Iterable<Node> calculateRoute( String start,
                                      String end,
                                      Interval interval )
{
    try ( Transaction tx = db.beginTx() )
    {
        TraversalDescription deliveryBaseFinder =
            createDeliveryBaseFinder( interval );
        Path upLeg = findRou teToDeliveryBase( start, deliveryBaseFinder );
        Path downLeg = findRouteToDeliveryBase( end, deliveryBaseFinder );
        Path topRoute = findRouteBetweenDeliveryBases(
            upLeg.endNode(),
            downLeg.endNode(),
            interval );
        Set<Node> routes = combineRoutes(upLeg, downLeg, topRoute);
        tx.success();
        return routes;
    }
}
```

Метод `calculateRoute()` сначала получает `deliveryBaseFinder` для указанного интервала, который затем использует для поиска двух

шагов. Далее находит маршрут между базами доставки и последними узлами каждого из шагов. И наконец, объединяет эти маршруты для получения окончательного результата.

Вспомогательный метод `CreateDeliveryBaseFinder()` создает описание обхода, настроенного под передаваемый интервал:

```
private TraversalDescription createDeliveryBaseFinder( Interval interval )
{
    return DELIVERY_BASE_FINDER.expand( DELIVERY_ROUTE_EXPANDER,
        new InitialBranchState.State<>( interval, interval ) );
}
```

Статический метод `ParcelRouteCalculator` создает описание обхода `DELIVERY_BASE_FINDER` с помощью `DELIVERY_ROUTE_EXPANDER`. Здесь состояние ветви инициализируется интервалом, указанным клиентом. Это позволяет использовать один и тот же экземпляр базового описания обхода (`DELIVERY_BASE_FINDER`) в нескольких запросах. Эта базовое описание расширяется и параметризируется для каждого запроса.

Описание обхода с соответственно настроенным интервалом передается в метод `findRouteToDeliveryBase()`, который находит стартовый узел в индексе местоположений, а затем выполняет обход:

```
private Path findRouteToDeliveryBase( String startPosition,
    TraversalDescription deliveryBaseFinder )
{
  Node startNode = IteratorUtil.single(
    db.findNodesByLabelAndProperty(LOCATION, "name", startPosition));
  return deliveryBaseFinder.traverse( startNode ).iterator().next();
}
```

Все вышеперечисленное относится к получению двух шагов. Последняя часть расчетов связана с поиском кратчайшего пути между базами доставки. Метод `calculateRoute()` принимает в качестве аргументов последние узлы в каждом шаге и соединяет их с учетом переданного клиентом интервала, вызывая метод `findRouteBetweenDeliveryBases()`. Реализация метода `findRouteBetweenDeliveryBases()` приведена ниже:

```
private Path findRouteBetweenDeliveryBases( Node deliveryBase1,
                                            Node deliveryBase2,
                                            Interval interval )
{
    PathFinder<WeightedPath> routeBetweenDeliveryBasesFinder =
        GraphAlgoFactory.dijkstra(
            CONNECTED_TO_EXPANDER,
            new InitialBranchState.State<>( interval, interval ),
```

```
            COST_EVALUATOR );
        return routeBetweenDeliveryBasesFinder
            .findSinglePath( deliveryBase1, deliveryBase2 );
}
```

Для поиска кратчайшего маршрута между двумя узлами вместо описания обхода этот метод использует алгоритм поиска кратчайших путей из библиотеки графовых алгоритмов Neo4j, в данном случае алгоритм Дейкстры (подробнее об алгоритме Дейкстры рассказывается в разделе «Поиск путей с помощью алгоритма Дейкстры» ниже). Этот алгоритм настраивается с помощью статического метода `ParcelRouteCalculator` статического свойства `CONNECTTED_TO_EXPANDER`, который, в свою очередь, инициализируется интервалом состояния ветви. Алгоритму также передается критерий оценки (другое статическое свойство), который определяет свойство взаимосвязей, указывающее вес или стоимость этой взаимосвязи. Вызов метода `findSinglePath` возвращает кратчайший путь между двумя базами доставки.

Самое сложное уже позади. Остается только соединить все эти маршруты и сформировать окончательный результат. Это относительно просто, единственная хитрость состоит в том, что перед добавлением в результат путь `downLeg` необходимо обратить (он проложен от пункта назначения вверх, а в конечный результат должен быть добавлен путь от базы доставки вниз):

```
private Set<Node> combineRoutes( Path upLeg,
                                 Path downLeg,
                                 Path topRoute )
{
    LinkedHashSet<Node> results = new LinkedHashSet<>();
    results.addAll( IteratorUtil.asCollection( upLeg.nodes() ));
    results.addAll( IteratorUtil.asCollection( topRoute.nodes() ));
    results.addAll( IteratorUtil.asCollection( downLeg.reverseNodes() ));
    return results;
}
```

Итоги

В этой главе было рассмотрено несколько случаев практического использования графовых баз данных и подробно описаны три реализации графовых баз данных для построения социальной сети, осуществления контроля доступа и управления сложными логистическими расчетами.

Следующая глава будет посвящена погружению во внутреннюю

структуру графовых баз данных. В заключительной главе будут рассмотрены несколько аналитических технологий и алгоритмов обработки графовых данных.

Глава 6

Внутреннее устройство графовых баз данных

Эта глава посвящена внутренней реализации графовых баз данных и ее отличиям от других средств хранения и извлечения с помощью запросов сложных, тесно связанных между собой данных с переменной структурой. Хотя не существует некого универсального эталона структуры графовых баз данных, тем не менее в этой главе описаны наиболее общие закономерности их архитектуры и наиболее часто встречающиеся в них компоненты.

Для иллюстрации тем, обсуждаемых в данной главе, используется графовая база данных Neo4j, и на это есть несколько веских причин. Neo4j – это графовая база данных с нативными средствами обработки, в частности с нативным хранилищем графов (нативные средства обработка и хранилища графов описаны в главе 1). Кроме того, эта база данных была наиболее распространенной графовой базой данных на момент написания книги, она поставляется с открытыми исходными кодами, что позволяет при необходимости глубже разобраться в некоторых вопросах, обратившись к ее исходным кодам. И наконец, эта база данных хорошо знакома авторам книги.

Нативная обработка графов

На протяжении книги неоднократно упоминались графовые модели со свойствами. Читатель должен быть уже хорошо знаком с понятием узлов, соединенных с помощью именованных и направленных взаимосвязей, и их свойствами. Несмотря на то что сама модель является

достаточно устоявшейся для всех графовых баз данных, существует множество способов программирования и представления графов в разных системах управления базами данных. Системы управления базами данных с нативной обработкой графов отличаются от прочих архитектур применением *смежности без индексов* (*index-free adjacency*).

В системах управления базами данных, использующих смежность без индексов, каждый узел содержит прямые ссылки на смежные с ним узлы. Каждый узел, таким образом, представляет собой как бы микроиндекс, ссылающийся на соседние узлы, что значительно снижает затраты, по сравнению с использованием глобальных индексов. Это значит, что время выполнения запроса будет зависеть не от общего размера графа, а от размера части графа, участвовавшей в поиске.

Ненативные системы управления графовыми базами данных, напротив, используют (глобальные) индексы для связывания узлов, как это показано на рис. 6.1. Обработка индексов при каждом обходе связана с большими затратами вычислительных ресурсов. Сторонники нативной обработки утверждают, что смежность без индексов является решающим фактором быстроты и эффективности обхода графов.

Чтобы понять, почему нативная обработка графов гораздо эффективнее обработки, основанной на глобальных индексах, рассмотрим следующее. Поиск по глобальному индексу имеет алгоритмическую сложность $O(\log n)$, а непосредственный поиск по взаимосвязям – алгоритмическую сложность $O(1)$. Для обхода сети за m шагов временные затраты при индексном подходе составляют $O(m \log n)$, что значительно превышает затраты $O(m)$ при смежности без индексов.

На рис. 6.1 изображен ненативный подход к обработке графов. Чтобы найти, с кем дружит Алиса (`Alice`), нужно сначала выполнить поиск по индексу, имеющий алгоритмическую сложность $O(\log n)$. Это еще приемлемо, когда поиск выполняется редко или поверхностно, но становится слишком затратным при смене направления обхода. Чтобы найти, не с кем дружит Алиса, а кто дружит с Алисой, потребуется выполнить несколько операций поиска по индексу для каждого узла, который потенциально может дружить с Алисой. Это сделает затраты очень обременительным. Поиск, с кем дружит Алиса, имеет вычислительную сложность $O(\log n)$, но поиск тех, кто дружит с Алисой, имеет сложность $O(m \log n)$.

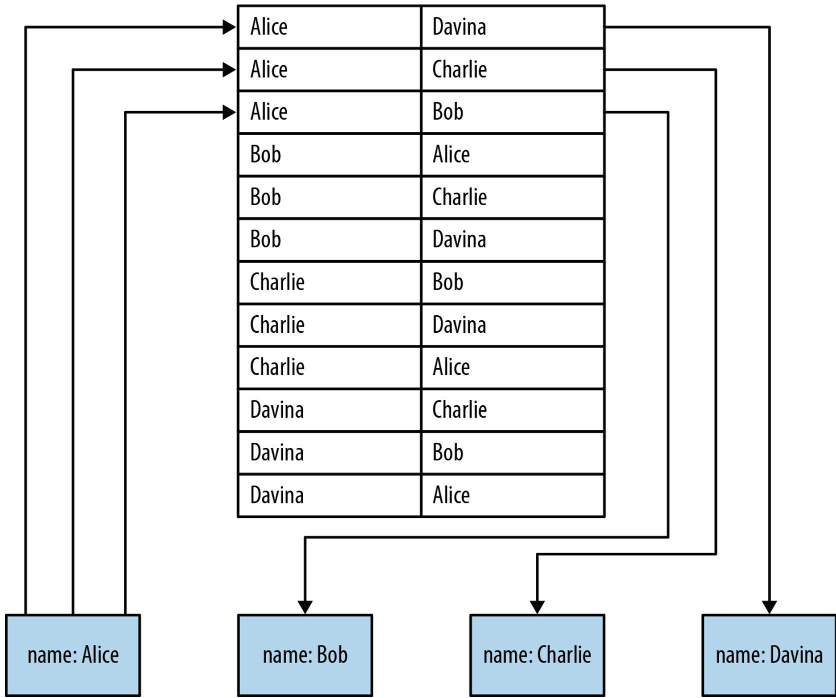


Рис. 6.1 ❖ Механизмы с ненативной обработкой используют индекс при обходе узлов

Смежность без индексов снижает затраты на «соединение»

При использовании смежностей без индекса двунаправленные соединения подготовлены к использованию заранее и хранятся в базе данных как взаимосвязи. Напротив, при использовании индексов для имитации соединения между записями в базе данных ничего не хранится. Исходя из этого, возникают две проблемы.

Во-первых, алгоритм поиска с использованием глобального индекса обычно является более затратным, чем обход физических взаимосвязей. Временные затраты при использовании индексов обычно оцениваются как $O(\log(n))$, тогда как, по крайней мере в Neo4j, временные затраты при обходе взаимосвязей оцениваются как $O(1)$. Даже в теории, уже при скромных значениях n, затраты, выражаемые логарифмом, будут в несколько раз превышать постоянные временные затраты. На практике все еще хуже, потому что граф и его глобальные индексы борются за такие ресурсы, как кэш и ввод/вывод (например, когда содержимое страницы зависит от данных индекса и графа).

Во-вторых, при имитации взаимосвязей с помощью индексов обход в направлении, «обратном» тому, для которого был построен индекс, весьма

проблематичен. Следовательно, необходимо либо создать индексы обратного поиска для каждого варианта обхода, либо выполнять полный обход с помощью оригинального индекса, являющийся операцией со сложностью $O(n)$. В этом случае, учитывая плохую алгоритмическую производительность, соединение будут занимать слишком много времени, чтобы его можно было использовать в оперативных системах.

Поиск по индексу еще можно использовать в небольших сетях, подобных приведенной на рис. 6.1, но станет выполняться слишком медленно на больших графах. Вместо поиска по индексу, подменяющего взаимосвязи при выполнении запросов, графовые базы данных с нативной обработкой применяют смежности без индексов, чтобы обеспечить высокую производительность. Рисунок 6.2 демонстрирует, как взаимосвязи делают индексы ненужными для поиска.

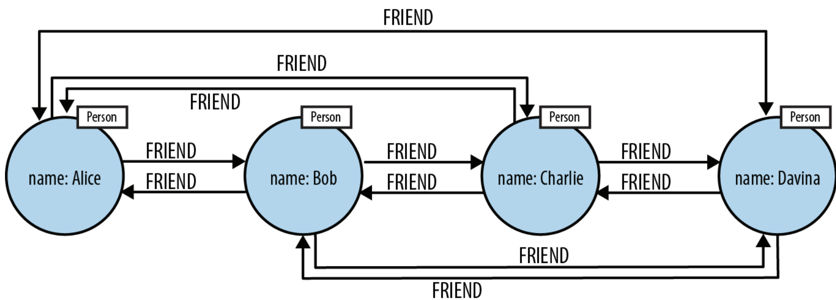


Рис. 6.2 ❖ Neo4j использует для обхода взаимосвязи, а не индексы

Вспомните, что в графовой базе данных обход взаимосвязей в любом направлении (от начала к концу или от конца к началу) не требует особых временных затрат. Как показано на рис. 6.2, чтобы найти тех, с кем дружит Алиса (`Alice`), нужно просто перебрать ее исходящие взаимосвязи `FRIEND` с затратами $O(1)$ на каждую. Чтобы найти тех, кто дружит с Алисой (`Alice`), нужно просто перебрать ее входящие взаимосвязи `FRIEND`, также с затратами $O(1)$ на каждую.

Такие затраты явно указывают, по крайней мере теоретически, что подобный обход графа чрезвычайно эффективен. Но этот обход может быть действительно высокопроизводительным, только когда выполняется специально предназначенной для этой цели архитектурой.

Нативное хранилище графов

Если смежность без индексов является ключом к высокой производительности обходов, запросов и записи, то одним из ключевых аспектов проектирования графовых баз данных является подход к хранению графов. Эффективный, нативный формат хранения графов, поддерживающий чрезвычайно быстрый способ обхода для любых алгоритмов, является важнейшей причиной использования графов. Рассмотрим архитектуру графовых баз данных на примере Neo4j.

Начнем с высокоуровневой архитектуры Neo4j, изображенной на рис. 6.3. А затем будем двигаться снизу вверх от файлов на диске через программные интерфейсы к языку запросов Cypher. Попутно обсудим эффективность и надежность Neo4j и проектные решения, которые делают Neo4j производительной и надежной графовой базой данных.

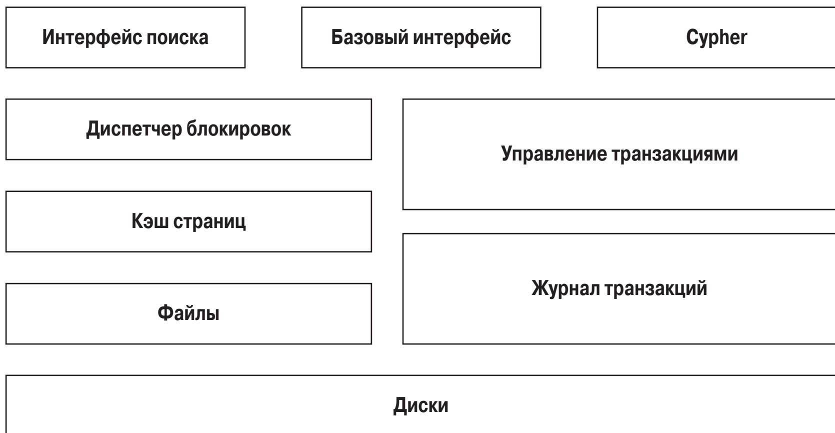
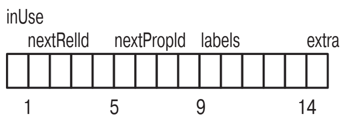


Рис. 6.3 ❖ Архитектура Neo4j

Neo4j хранит графовые данные в нескольких разных *файлах хранилища* (*store files*). Каждый файл содержит данные из определенной части графа (например, отдельные хранилища предназначены для узлов, взаимосвязей, меток и свойств). Разделение обязанностей при хранении, и особенно отдельное хранение структуры графа и значений свойств, улучшает производительность обхода графа, даже при том, что реальная структура графа в файле отличается от того, что ви-

дят пользователи. Для начала рассмотрим структуру хранения узлов и взаимосвязей в файлах, изображенную на рис. 6.4[1].



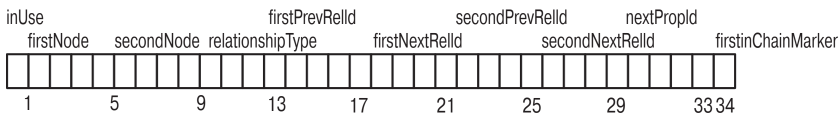


Рис. 6.4 ❖ Структура хранения узлов и взаимосвязей в файлах Neo4j

Файл для хранения узлов содержит записи с данными об узлах. Создание каждого узла в графе заканчивается сохранением соответствующей записи в физический файл `neostore.nodestore.db`. Как и большинство файлов в Neo4j, хранилище узлов состоит из записей фиксированного размера, где каждая запись занимает 15 байт. Фиксированный размер записи обеспечивает быстрый поиск узлов в файле. При поиске узла, имеющего идентификатор `100`, заранее известно, что содержащая о нем сведения запись располагается в файле, начиная с 1500-го байта. Основываясь на таком формате, база данных имеет возможность непосредственно вычислить местоположение записи, что означает выполнение поиска с затратами $O(1)$ вместо $O(\log n)$.

Первый байт в записи узла содержит признак актуальности `inUse`. Если он установлен, значит, запись базы данных в настоящее время задействована для хранения узла, если нет – в нее можно поместить сведения о новом узле (для отслеживания неиспользуемых записей в Neo4j служат файлы с расширением `.id`). Следующие четыре байта содержат идентификатор первой взаимосвязи `nextRelId`, подключенной к узлу, а следующие четыре байта – идентификатор первого свойства узла `nextPropId`. Следующие пять байтов содержат указатель на метку узла `label` в хранилище меток (несколько меток соединяются вместе). Последний байт `extra` зарезервирован для признаков. Один

[1] Структура записи приведена для версии 2.2 Neo4j, для других версий размер записи может отличаться.

из таких признаков идентифицирует тесно связанные узлы, а прочая часть байта зарезервирована для использования в будущем. Запись узла достаточно легковесна, потому что содержит лишь несколько указателей на списки взаимосвязей, меток и свойств.

Соответственно, взаимосвязи хранятся в файле взаимосвязей `neostore.relationshipstore.db`. Как и хранилище узлов, хранилище взаимосвязей тоже состоит из записей фиксированного размера. Каждая запись взаимосвязей содержит идентификаторы начального `firstNode` и конечного `secondNode` узлов взаимосвязи, указатель на вид взаимосвязи `relationshipType` (сами виды хранятся в хранилище видов взаимосвязей), указатели на следующую `firstPrevrelId` и предыдущую `firstNextRelId` взаимосвязи для начального и конечного узлов (`firstPrevrelId, firstNextRelId, secondPrevrelId` и `secondNextRelId`) и признак `firstInChainMarker`, указывающий, является ли текущая запись первой в том, что часто называют *цепочкой взаимосвязей* (*relationship chain*).

Хранилища узлов и взаимосвязей предназначены исключительно для хранения структуры графа, а не значений его свойств. Оба хранилища состоят из записей фиксированного размера, что позволяет быстро найти любую запись в файле по ее идентификатору. Эти важнейшие проектные решения и обеспечивают высокую производительность Neo4j.

Рисунок 6.5 демонстрирует взаимодействие различных файлов хранения. Каждая из двух записей с информацией об узлах содержит указатель на первое свойство узла и первую взаимосвязь в цепочке взаимосвязей. Чтобы прочитать свойства узла, следует обойти однонаправленный список, начав с указателя на первое свойство. Чтобы найти взаимосвязи узла, нужно перейти по указателю на первую взаимосвязь (такую как взаимосвязь `LIKES` в данном примере) и затем проследовать по двусвязному списку для данного конкретного узла (т. е. от начального узла двунаправленного списка или от конечного узла двунаправленного списка) до нужной взаимосвязи. Получив запись взаимосвязи, можно прочитать свойства этой взаимосвязи (если таковые имеются) с помощью такого же односвязного списка, как для свойств узла, или перейти к записям двух связанных узлов по идентификаторам начального и конечного узлов. Умножение этих идентификаторов на размер записи узлов дает в результате смещение узла в файле.

Двусвязные списки в хранилище взаимосвязей

Не волнуйтесь, если структура хранения взаимосвязей покажется вам сложной. Она действительно сложнее структуры хранения узлов или свойств.

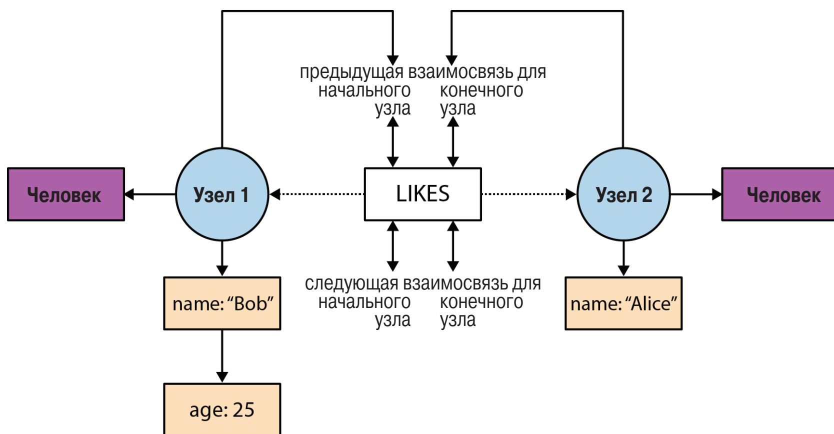


Рис. 6.5 ❖ Физическое хранение графа в Neo4j

Воспринимайте запись взаимосвязи как «принадлежащую» двум узлам, начальному и конечному. Очевидно, что хранение двух записей со сведениями об одной и той же взаимосвязи выглядит расточительным. Следовательно, столь же очевидно, что запись взаимосвязи должна каким-то образом привязываться к обоим узлам, начальному и конечному.

Вот почему указатели (они же идентификаторы) соединяются в двусвязные списки. Одно направление представляет список взаимосвязей с точки зрения начального узла. Другое направление – с точки зрения конечного узла. Двусвязный список просто позволяет быстро обойти список в любом направлении и эффективно добавлять и удалять взаимосвязи.

Выбор взаимосвязи предполагает обход списка взаимосвязей до получения необходимой взаимосвязи (например, нужного вида или имеющей нужное значение некоторого свойства). После получения нужной взаимосвязи вновь применяется умножение идентификатора на размер записи и получение указателя.

Благодаря фиксированному размеру записей и идентификаторам в форме указателей обход структуры данных по указателям выполняется очень быстро. При переходе по взаимосвязи от одного узла к другому база данных выполняет лишь несколько быстрых вычислений идентификаторов (эти вычисления выполняются гораздо быстрее, чем поиск в глобальных индексах, который необходим при имитации графов в неграфовых базах данных):

1. По заданной записи узла находится первая запись в цепочке взаимосвязей посредством расчета смещения в хранилище взаимосвязей, т. е. умножения идентификатора на фиксированный размер записи. Это позволяет сразу перейти к нужной записи.

2. Из записи взаимосвязи выбирается поле для *второго узла*, содержащее его идентификатор. Умножение значения поля на размер записи дает смещение записи узла в хранилище.

Если требуется ограничить обход лишь взаимосвязями определенного вида, следует добавить поиск взаимосвязи нужного вида в хранилище взаимосвязей. И снова, чтобы получить смещение записи взаимосвязи определенного типа в хранилище, понадобится лишь умножить идентификатор на размер записи. Аналогично, чтобы ввести ограничение по меткам, достаточно выполнить те же операции с хранилищем меток.

Кроме хранилищ узлов и взаимосвязей, формирующих структуру графа, имеются также и файлы свойств, хранящие данные в виде пар ключ-значение. Neo4j как графовая база данных со свойствами позволяет присоединять свойства, т. е. пары имя-значение, к узлам и взаимосвязям. Как следствие записи с узлами и взаимосвязями содержат ссылки на хранилище свойств.

Записи свойств хранятся в файле `neostore.propertystore.db`. Подобно записям узлов и взаимосвязей, записи свойств имеют фиксированный размер. Каждая запись содержит четыре блока сведений о свойстве и идентификатор следующего свойства в цепочке (для хранения цепочек свойств используется односвязный список, в отличие от двусвязного, применяемого для хранения цепочек взаимосвязей). Каждое свойство занимает от одного до четырех блоков, следовательно, каждая запись может содержать до четырех свойств. Запись свойства содержит тип свойства (Neo4j поддерживает все простые типы JVM, а также строки и массивы простых типов JVM) и указатель на индексный файл (`neostore.propertystore.db.index`) с именами свойств. Для каждого значения свойства запись содержит либо указатель на запись в динамическом хранилище, либо встроенное значение. Динамические хранилища предназначены для хранения сложных значений. Существуют два динамических хранилища: динамическое хранилище строк (`neostore.propertystore.db.strings`) и динамическое хранилище массивов (`neostore.propertystore.db.arrays`). Динамические хранилища содержат связанные списки записей фиксированного размера, следовательно, очень длинная строка или большой массив может занимать более одной динамической записи.

Встраивание и оптимизация использования хранилища свойств

База данных Neo4j поддерживает оптимизацию хранилищ посредством встраивания некоторых свойств непосредственно в файл хранения свойств (`neostore.propertystore.db`). Это происходит, когда значения свойств можно представить так, чтобы уместить их в одном или нескольких из четырех блоков записи. Практически это значит, что такие данные, как номера телефонов или почтовые индексы, могут встраиваться непосредственно в файл хранения свойств, а не помещаться в динамические хранилища. Это ведет к уменьшению количества операций ввода и вывода и увеличению пропускной способности, так как требуется доступ лишь к одному файлу.

В дополнение к встраиванию некоторых значений свойств база данных Neo4j поддерживает экономную организацию имен свойств. Например, в социальном графе наверняка имеется множество узлов со свойствами `first_name` (имя) и `last_name` (фамилия). Было бы слишком расточительным хранить на диске все многократно повторяющиеся имена свойств, поэтому вместо них используются ссылки на хранилище свойств через индексный файл свойств. Индексирование позволяет хранить для всех свойств с одним и тем же именем одну-единственную запись, и, таким образом, для графов с часто повторяющимися именами свойств база данных Neo4j позволяет достичь значительного уменьшения объема хранимых данных и ощутимого сокращения числа операций ввода и вывода.

Но эффективное использование хранилища – это только полдела. Другим фактором, кроме оптимизации структуры хранения для быстрого обхода, оказывающим значительное влияние на производительность, является применяемое оборудование. За последние годы объемы используемой памяти значительно возросли, но очень большие графы по-прежнему невозможно целиком разместить в памяти. Жесткие диски тратят миллисекунды на позиционирование, что, конечно, очень быстро по человеческим меркам, но слишком медленно по компьютерным критериям. Твердотельные диски (SSD) обеспечивают гораздо лучшие показатели (потому что в них отсутствуют движущиеся части), но длительность прохождения данных между центральным процессором и диском много больше времени, затрачиваемого при обмене данными с кэшем L2 или оперативной памятью, использование которых идеально при работе с графами.

Для смягчения влияния эксплуатационных характеристик механических или электронных запоминающих устройств многие графовые базы данных применяют кэширование, чтобы обеспечить с высокой вероятностью быстрый доступ к графу. Начиная с версии 2.2, база данных Neo4j использует кэширование кучи для достижения такого прироста производительности.

База данных Neo4j версии 2.2 использует *страничное кэширование* LRU-K. Страничное кэширование – это применение ориентированного на страницы кэша LRU-K, т. е. кэширование, разделяющее каждое из хранилищ на отдельные области, а затем помещающее в кэш определенное количество фрагментов из файла хранения. Удаление страниц из кэша выполняется на основе политики частоты их использования (Least Frequently Used, LFU), делающей акцент на популярность страницы. То есть непопулярные страницы будут удалены из кэша и заменены более популярными страницами, даже если к ним недавно обращались. Эта политика обеспечивает статистически оптимальное кэширование ресурсов.

Программные интерфейсы

Хотя инфраструктуры файловой системы и кэширования интересны сами по себе, разработчики редко работают с ними напрямую. Обычно вместо этого они воздействуют на базу данных с помощью языка графовых запросов, который может быть либо императивным, либо декларативным. В примерах для этой книги использовался язык запросов Cypher – встроенный декларативный язык запросов базы данных Neo4j, – потому что этот язык прост в изучении и использовании. Существуют также другие программные интерфейсы, и выбор между ними зависит от того, какие задачи требуется решить и что является приоритетом. Начиная работать с новым проектом, важно разобраться с набором интерфейсов и их возможностями. Самое главное в этом разделе – понимание, что эти интерфейсы нужно рассматривать в виде стека, изображенного на рис. 6.6, верхняя часть которого обеспечивает наглядность и декларативность, а нижняя – точность императивного стиля и (самый нижний слой) возможности «голого железа».

Рис. 6.6 ❖ Обзор пользовательских программных интерфейсов Neo4j

Подробному описанию языка Cypher была посвящена глава 3. В следующих разделах будут рассмотрены остальные программные интерфейсы, причем их описание будет упорядочено снизу вверх. Это лишь ознакомительный тур по программным интерфейсам. Не все графовые базы данных обладают одинаковым количеством слоев, и совсем необязательно одни и те же их слои ведут себя и взаимодействуют с пользователями одинаково. Каждый программный интерфейс имеет свои преимущества и недостатки, которые нужно учесть при принятии решения.

Программный интерфейс ядра

На самом низшем уровне стека программных интерфейсов находятся обработчики транзакционных событий (http://docs.neo4j.org/chunked/stable/transactions-events.html). Их использование позволяет отслеживать выполнение транзакций ядром и реагировать (или игнорировать) на транзакции в зависимости от данных и этапа жизненного цикла.

Обработчики событий выполнения транзакций

Типичным случаем использования обработчиков событий транзакций является предотвращение физического удаления записей. Обработчик можно настроить для перехвата события удаления узла и вместо удаления отмечать узлы как удаленные (или, в более сложном случае, как «устаревшие», путем создания взаимосвязей архивации, содержащих дату и время).

Базовый интерфейс

Базовый программный интерфейс Neo4j является императивом Java API, который предоставляет доступ к графовым примитивам узлов, взаимосвязей, свойств и меток. При использовании базового интерфейса для чтения он действует в отложенном режиме, т. е. обход взаимосвязей происходит только при вызове метода получения следующего узла. Данные извлекаются из графа по мере их затребования клиентом программного интерфейса. Имеется возможность в любой момент прервать обход. При записи базовый программный интерфейс предоставляет возможность управления транзакциями для обеспечения атомарности, согласованности, изолированности и долговременности.

Следующий код – это фрагмент, позаимствованный из руководства Neo4j (https://github.com/jimwebber/neo4j-tutorial) и предназначенный для поиска гуманоидных компаньонов Доктора из сериала «Doctor Who»[1]:

[1] Сериал «*Doctor Who*» является самым продолжительным научно-фантастическим сериалом и фаворитом команды Neo4j.

```
// Для краткости поиск узла, представляющего самого доктора, здесь опущен

Iterable<Relationship> relationships =
            doctor.getRelationships( Direction.INCOMING, COMPANION_OF );

for ( Relationship rel : relationships )
{
    Node companionNode = rel.getStartNode();
    if ( companionNode.hasRelationship( Direction.OUTGOING, IS_A ) )
    {
        Relationship singleRelationship = companionNode
                                                .getSingleRelationship( IS_A,
                                                    Direction.OUTGOING );
        Node endNode = singleRelationship.getEndNode();
        if ( endNode.equals( human ) )
        {
            // Найден!
        }
    }
}
```

Этот код действует императивно: он выполняет обход компаньонов доктора и проверяет, связан ли каждый из узлов компаньонов взаимосвязью `IS_A` с узлом, представляющим гуманоидный вид. Если такая связь имеется, с ним производятся некоторые действия.

Базовый программный интерфейс является императивным, и поэтому требуется его тонкая настройка под структуру графа. Но он может работать очень быстро. Однако его код будет жестко привязан к конкретной прикладной области. По сравнению с программными интерфейсами более высокого уровня (в частности, с Cypher), его применение потребует большего объема кода для решения той же задачи. Тем не менее близость базового интерфейса к хранилищу записей позволяет в полной мере использовать структуру хранилищ и кэширование в пользовательском коде.

Фреймворк Traversal

Фреймворк Traversal (интерфейс поиска) – это декларативный программный интерфейс на языке Java. Он дает возможность пользователю задать набор условий, ограничивающих область обхода графа. Имеется возможность задавать доступные виды взаимосвязей, их направления (фактически создавая фильтры взаимосвязей). Так же можно указать, нужно ли сначала выполнить обход в ширину или

в глубину, задать условие оценки маршрута, которое будет проверяться при достижении каждого узла. С помощью такой оценки на каждом шаге определяется, нужно ли продолжать обход. Следующий фрагмент демонстрирует обход при помощи программного интерфейса Traversal:

```
Traversal.description()
  .relationships( DoctorWhoRelationships.PLAYED, Direction.INCOMING )
  .breadthFirst()
  .evaluator( new Evaluator()
  {
    public Evaluation evaluate( Path path )
    {
    if ( path.endNode().hasRelationship(
                          DoctorWhoRelationships.REGENERATED_TO ) )
    {
      return Evaluation.INCLUDE_AND_CONTINUE;
    }
    else
    {
      return Evaluation.EXCLUDE_AND_CONTINUE;
    }
  }
} );
```

Этот фрагмент явно демонстрирует преимущественно декларативный характер фреймворка Traversal. Метод `relationships()` объявляет, что обход должен производиться только по входящим `INCOMING` взаимосвязям `PLAYED`. Затем с помощью метода `breadthFirst()` объявляется, что обход должен выполняться сначала в ширину, т. е. первыми следует посетить всех ближайших соседей, а затем уже двигаться вглубь.

Фреймворк Traversal декларирует лишь порядок навигации по графу. Для реализации оценки нужно обратиться к императивному базовому интерфейсу. То есть базовый интерфейс используется, чтобы на основании анализа пути к текущему узлу решить, нужно ли продолжать обход графа (базовый интерфейс можно также использовать для изменения графа из модуля, выполняющего оценку). И снова нативные графовые структуры базы данных, представленные примитивами узлов, связей и свойств, занимают центральное место в программном интерфейсе.

Базовый программный интерфейс, фреймворк Traversal или язык Cypher?

Какой из этих разных подходов к выполнению графовых запросов следует выбрать?

Базовый интерфейс позволяет тонко настроить запросы, чтобы они были тесно связаны с графом. Хорошо написанные запросы для базового интерфейса часто выполняются быстрее, чем запросы, основанные на других подходах. Их недостатком является большой размер, что требует значительных усилий от разработчиков при их написании. Кроме того, близость к графу делает их жестко привязанными к его структуре. При изменении структуры графа они могут перестать работать. Язык Cypher гораздо терпимее относится к изменениям структуры, такие его возможности, как переменные длины путей, помогают значительно смягчить последствия внесения изменений в структуру.

Фреймворк Traversal не только имеет большую гибкость, чем базовый интерфейс (так как позволяет объявлять информационные цели), но и более лаконичен, поэтому написание запросов с помощью фреймворка Traversal, как правило, требует меньше усилий от разработчиков, чем написание эквивалентного запроса с помощью базового интерфейса. Но в связи с тем, что фреймворк Traversal имеет общее назначение, написанные на его основе запросы выполняются несколько медленнее, чем хорошо написанные запросы для базового интерфейса.

Причиной применения базового программного интерфейса или фреймворка Traversal (и, следовательно, отказа от применения языка Cypher и его наглядности) может послужить программирование нестандартной ситуации, когда требуется реализация алгоритма, который не может быть эффективно реализован с помощью соответствующего шаблона языка Cypher. Выбор между базовым программным интерфейсом и фреймворком Traversal определяется достаточностью уровня абстракции фреймворка Traversal; если и этого недостаточно, следует обратиться к самому низкому уровню базового интерфейса, необходимому для реализации алгоритма в соответствии с заданными требованиями к его производительности.

На этом завершается краткий обзор программных интерфейсов графов на примере встроенных интерфейсов Neo4j. Мы увидели, как эти интерфейсы отражают структуры, используемые на нижних уровнях стека Neo4j, и как они согласуются с идиоматическим и быстрым обходом графов.

Для базы данных недостаточно быть быстрой, она также должна быть надежной. Это подводит к рассмотрению нефункциональных характеристик графовых баз данных.

Нефункциональные характеристики

На данный момент уже были рассмотрены архитектура нативных графовых баз данных и реализация некоторых из встроенных в них возможностей на примере Neo4j. Но, чтобы считаться *надежной*, лю-

бая технология хранения данных должна обеспечивать определенный уровень гарантий в отношении долговременности и доступности хранимых данных[1].

Одной из самых распространенных методик оценки реляционной базы данных является оценка по числу обрабатываемых транзакций в секунду. В реляционном мире предполагается, что транзакции обеспечивают требования ACID (даже при наличии сбоев), что позволят сохранять согласованность данных и предоставляет возможность их восстановления. При непрерывной обработке и управлении большими объемами данных реляционная база данных масштабируется таким образом, что для обработки запросов и внесения изменений постоянно доступно множество экземпляров и потеря отдельного экземпляра не слишком влияет на работу кластера в целом.

По крайней мере, на высоком уровне это же относится и к графовым базам данных. Они должны гарантировать согласованность, возможность восстановления при сбоях и предотвращение повреждения данных. Кроме того, они должны предусматривать масштабирование, чтобы обеспечить высокий уровень доступности и расширяемости. Следующие разделы будут посвящены обсуждению, что каждое из этих требований значит для архитектуры графовых баз данных. И здесь определенные моменты будут проиллюстрированы конкретными примерами применительно к архитектуре Neo4j. Следует отметить, что не все графовые базы данных полностью поддерживают требования ACID. Это важно для понимания специфики модели транзакций конкретной базы данных. Поддержка в Neo4j транзакций в стиле ACID обеспечивает ту же высокую надежность, которая является привычной для корпоративных систем управления реляционными базами данных.

Транзакции

Надежность вычислительных систем уже десятилетия базируется на транзакциях. Хотя многие NOSQL-хранилища не поддерживают транзакций, в частности потому, что существуют непроверенные предположения, что поддержка транзакций затрудняет масштабирование системы, транзакции остаются фундаментальной основой на-

[1] Формальным определением надежности является следующее определение: «надежной является вычислительная система, которая обеспечивает доверие к предоставляемым ей услугам», размещенное на http://www.dependability.org/.

дежности современных графовых баз данных, включая Neo4j. (В утверждении, что транзакции ограничивают масштабируемость, есть доля правды, поскольку реализация двухфазной фиксации может вызвать проблемы с готовностью в некоторых патологических случаях, но в целом эффект гораздо менее заметен, чем это обычно считается.)

Транзакции в Neo4j семантически идентичны традиционным транзакциям баз данных. Запись выполняется в контексте транзакций, а в целях согласованности всех узлов и взаимосвязей, участвующих в транзакции, применяется блокировка записи. После успешного *завершения* транзакции изменения фиксируются на диске и блокировка записи сбрасывается. Эти действия позволяют гарантировать атомарность транзакций. Если транзакция по каким-то причинам терпит крах, записи будут удалены и блокировка записи сброшена, в результате граф остается в предыдущем согласованном состоянии.

Если две или более транзакций пытаются параллельно изменить одни и те же элементы графа, Neo4j обнаружит ситуацию, потенциально ведущую в тупик, и упорядочит транзакции. Записи внутри контекста одной транзакции не будут видны другим транзакциям, тем самым сохраняя их изоляцию.

Реализация транзакций в Neo4j

Реализация транзакций в Neo4j концептуально проста. Отдельную транзакцию можно представить как объект в памяти, содержащий данные, которые будут записаны в базу данных. Этот объект поддерживается менеджером блокировок, который обеспечивает блокировку записи узлов и взаимосвязей при их создании, изменении и удалении. При откате транзакции транзакционный объект отбрасывается и блокировка записи освобождается, а при ее подтверждении транзакционный объект записывается на диск.

Neo4j при подтверждении записи данных на диск использует *предварительную регистрацию записи* (*Write Ahead Log*) для добавления изменений в виде сведений о действиях в активный журнал транзакций. При подтверждении транзакции (при условии получения разрешения на этапе подготовки) сведения о фиксации будут внесены в журнал. Это приведет к сбросу журнала на диск, тем самым делая изменения долговечными. После сброса на диск изменения будут применены собственно к графу. После применения всех изменений к графу все блокировки, связанные с транзакцией, освобождаются.

После завершения транзакции система оказывается в состоянии, когда изменения гарантированно применены к базе данных и никакие сбои не смогут на это повлиять. Ясно, что это обеспечивает возможность восстановления, и, следовательно, гарантирует непрерывность предоставления услуг.

Восстанавливаемость

Базы данных ничем не отличаются от любого другого программного обеспечения – их надежность зависит от ошибок, допущенных при реализации, от оборудования, на котором они работают, в том числе сбоев в аппаратных системах питания, охлаждения и подключения к сети. Несмотря на усердные старания инженеров свести к минимуму возможность отказов, все равно в какой-то момент база данных терпит крах, хотя среднее время между отказами должно быть действительно большим.

В хорошо спроектированной системе крах сервера базы данных не должен повлиять на доступность, хотя и может уменьшить пропускную способность. А при возобновлении сервером работы после сбоев он не должен повредить данные пользователей независимо от характера или сроков аварии.

При восстановлении после аварийного завершения, вызванного сбоем или даже ошибкой оператора, Neo4j проверяет последнее содержимое активного журнала транзакций и повторяет найденные транзакции. Вполне возможно, что некоторые из этих транзакций уже были применены к хранилищу, но их повторное применение не изменит конечного результата, а это гарантирует, что после восстановления состояние хранилища будет соответствовать всем успешно совершенным до отказа транзакциям.

В случае единственного экземпляра базы данных локального восстановления вполне достаточно. Однако обычно базы данных объединяются в кластер (подробнее мы остановимся на этом в ближайшее время) для обеспечения высокого уровня доступности для клиентских приложений. К счастью, кластеризация дает дополнительные преимущества при восстановлении. Мало того, что экземпляр станет соответствовать всем транзакциям, успешно завершенным до отказа, как это было упомянуто выше, – он также сможет быстро догнать другие экземпляры кластера, тем самым достигнув соответствия всем транзакциям, успешно совершенным и *после* его отказа. То есть после завершения локального восстановления репликатор может обратиться к другим членам кластера, обычно к главному из них, для получения сведений о новых транзакциях. Затем он может применить новые транзакции к собственному набору данных с помощью повтора транзакций.

Восстанавливаемость связана с возможностью базы данных устранять ошибки, возникшие непосредственно после сбоев. В дополнение

к обеспечению восстановления хорошая база данных должна иметь высокой уровень доступности для удовлетворения потребностей приложений, тесно связанных с данными.

Доступность

Предоставляющие немалые преимущества сами по себе, транзакции и возможности восстановления Neo4j также положительно влияют на показатели доступности. Возможности базы данных, связанные с определением сбоев и, при необходимости, восстановлением экземпляра после аварии, обеспечивают быстрый возврат доступности без вмешательства человека. И конечно же, увеличение количества экземпляров, находящихся в рабочем состоянии, повышает общую доступность базы данных при обработке запросов.

Отдельные, не соединенные между собой экземпляры базы данных очень редко используются в промышленной эксплуатации. Чаще для обеспечения высокой доступности экземпляры базы данных объединяются в кластер. Для достижения полной идентичности графов, хранящихся на разных серверах, в Neo4j используется кластерный механизм ведущий-ведомый. Записи реплицируются из главного на подчиненные серверы через определенные промежутки времени. В любой момент времени главный сервер и некоторые из подчиненных будут иметь актуальную копию полного графа, в то время как другие подчиненные серверы будут догонять их (обычно отставание составляет миллисекунды).

Наиболее популярной топологией записи является классическая модель записи на главный сервер и чтения с подчиненных серверов. При этом все требования на запись в базу данных направляются на главный сервер, а все операции чтения – на подчиненные. Это обеспечивает асимптотическое масштабирование записи (ограниченное мощностью одного сервера), но позволяет примерно линейный рост масштабирования чтения (с учетом скромных накладных затрат на управление кластером).

Хотя запись на главный сервер и чтение с подчиненных серверов являются классической топологией развертывания, Neo4j также поддерживает запись через подчиненные серверы. В этом случае подчиненный сервер, в который клиент выполняет запись, в первую очередь должен обеспечить согласование с главным сервером (ведь он «догоняющий»), после этого происходит синхронизация данных. Это полезно при необходимости немедленного сохранения записи в двух экземплярах базы данных. Кроме того, возможность записи в любой

экземпляр предлагает дополнительную гибкость при развертывании. Это происходит за счет более высокой латентности записи из-за форсирования фазы синхронизации, что не означает появления возможности распределения записей по системе – всем записям необходимо будет пройти через главный сервер.

Другие варианты репликаций в Neo4j

В версии Neo4j 1.8 можно было указать, что запись на главном сервере должна быть реплицирована определенным количеством репликаторов, прежде чем транзакция будет считаться завершенной. Это обеспечивает альтернативу закреплению на уровне «не менее чем на двух», достигаемую при записи через подчиненный сервер. Более подробную информацию можно найти в разделе «Репликация» в главе 4.

Еще одним аспектом доступности является конкуренция за доступ к ресурсам. Операция, которая требует эксклюзивного доступа (например, для записи) к определенной части графа, может пострадать от достаточно высокой степени защиты, оказавшись недоступной. Подобную конкуренцию можно наблюдать при грубой блокировке на уровне таблицы в реляционных базах данных, где доступ на запись может быть ограничен, даже если по логике вещей никакого конфликта не существует.

К счастью, графовой модели доступа, как правило, характерна большая гибкость, особенно при выполнении идиоматических графовых локальных запросов. Локализация графовой операции начинается с закрепления одной или нескольких заданных точек графа, а затем выполняется обход соседних подграфов. Исходными точками таких запросов являются обычно сущности, определяемые особенностями прикладной области, например пользователи или товары. Наличие таких исходных точек приводит к локальности распространения области запроса. В свою очередь, клиенты получают малое время отклика и высокую доступность.

Преимущества идиоматических запросов

Джеки Стюарт (Jackie Stewart), пилот Формулы-1, как-то сказал, что для того, чтобы хорошо водить автомобиль, не нужно быть инженером, но нужно чувствовать машину. То есть хорошего результата можно добиться лишь при гармоничном взаимодействии водителя и автомобиля.

Примерно так же графовые запросы чувствуют базу данных, если они сформулированы в виде идиоматических, локализованных запросов, которые начинают обход от одной или нескольких начальных точек. Обеспечивающая их выполнение инфраструктура, в том числе кэширование и доступ к хранилищу, оптимизирована для такого рода нагрузок.

Идиоматические запросы обладают полезными побочными эффектами. Например, так как кэширование совместимо с идиоматическим поиском, идиоматические запросы лучше используют кэш и быстрее выполняются, чем неидиоматические. В свою очередь, быстрые запросы быстрее освобождают базу данных для следующих запросов, что способствует увеличению пропускной способности и ощущению лучшей доступности у клиентов, так как уменьшается общее время ожидания.
Неидиоматические запросы (например, выполняющие случайный выбор узлов или взаимосвязей, а не обход) обладают прямо противоположными свойствами: они не учитывают низкоуровневых особенностей кэширования и работают медленнее, потому что для их выполнения требуется большое количество дисковых операций ввода-вывода. Так как запросы выполняются медленно, база данных может обрабатывать за секунду лишь небольшое их количество, а это означает уменьшение производительности базы данных с точки зрения клиентов.
Независимо от используемой базы данных понимание ее инфраструктуры хранения и кэширования поможет созданию идиоматических, а следовательно, чувствующих машину, запросов, которые позволят максимизировать производительность.

Наше последнее наблюдение говорит о положительном влиянии горизонтального масштабирования внутрикластерных репликаций не только на отказоустойчивость, но и на время отклика. Благодаря наличию множества серверов, доступных для обработки заданной нагрузки, задержки при обработке запросов невелики, и обеспечивается высокая доступность. Далее мы рассмотрим собственно масштабирование, которое характеризуется большим количеством нюансов, а не сводится просто к количеству привлеченных серверов.

Масштабирование

Важность темы масштабирования возрастает вместе с объемом данных. И в самом деле, проблемы масштабирования, которые сложно было решить с помощью реляционных баз данных, стали мощным толчком для развития NOSQL-движения. В каком-то смысле графовые базы данных ничем не выделяются, в конце концов, они также должны масштабироваться для удовлетворения потребностей современных приложений. Но масштабирование нельзя свести к некоторой простой величине, например к числу транзакций в секунду. Оно больше походит на агрегированную величину и определяется значениями на нескольких осях.

Для графовых баз данных разделим рассмотрение масштабирования на три ключевые темы:

1) мощность (размер графа);
2) задержка (время отклика);
3) производительность чтения и записи.

Мощность

Некоторые производители графовых баз данных решили не ограничивать размера графа в ущерб производительности и затратам при хранении. База данных Neo4j изначально заняла в этом вопросе уникальную позицию, сохранив высокую производительность и низкие затраты при хранении (и следовательно, уменьшение объема требуемой памяти и количества операций ввода-вывода) путем такого ограничения максимальных размеров графа, которое охватывает около 95 процентов случаев их применения. Причина такого компромисса кроется в использовании записей фиксированных размеров и указателей, которые (как это было описано в разделе «Нативное хранилище графов») широко используются в хранилищах. На момент написания книги текущая версия базы данных Neo4j способна поддерживать графы, содержащие десятки миллиардов узлов, связей и свойств. Это позволяет работать с графами, моделирующими набор данных социальной сети примерно того же размера, что и Facebook.

В своих планах на будущее команда Neo4j публично выразила намерение достичь поддержки более 100 миллиардов узлов, взаимосвязей и свойств одного графа.

Насколько большим должен быть набор данных, чтобы почувствовать все преимущества графовых баз данных? На самом деле меньше, чем вы могли бы подумать. Для запросов второй или третьей степени преимущества производительности проявляются уже на наборах данных, имеющих несколько тысяч узлов. Чем выше степень запроса, тем четче прослеживается разница. Преимущество, заключающееся в упрощении разработки, естественно, не зависит от объема данных и присуще базе данных любого размера. Авторы наблюдали за множеством приложений в диапазоне от нескольких десятков тысяч узлов и нескольких сотен тысяч взаимосвязей до миллиардов узлов и взаимосвязей.

Задержки

Графовые базы данных не страдают от проблем, приводящих к задержкам отклика в традиционных реляционных базах данных, где увеличение размера таблиц данных и, следовательно, увеличение индексов значительно замедляют операции соединения (этот простой, обыденный факт является одной из ключевых причин того, что настройка производительности всегда будет основной заботой администраторов реляционных баз данных). В графовой базе данных

большинство запросов по указанному шаблону находит начальный узел (или узлы) посредством индекса. Далее выполняется обход, использующий комбинацию направления и шаблона сопоставления, для поиска данных. Это означает, что производительность, в отличие от реляционных баз данных, зависит не от общего объема набора данных, а лишь от объема самих запрашиваемых данных. И это приводит к достижению практически постоянной производительности (определяемой только размером результирующего набора), не зависящей от роста размера набора данных (хотя, как было описано в главе 3, по-прежнему остается актуальной тонкая настройка структуры графа под потребности запросов, даже если объем данных не слишком велик).

Производительность

Можно предположить, что графовые базы данных используют тот же подход к масштабированию, что и другие базы данных. Но это не так. Наблюдая за поведением приложений, интенсивно использующих операции ввода-вывода, можно заметить, что при выполнении одной бизнес-операции происходят массовое чтение и запись связанных между собой данных. Другими словами, приложение выполняет несколько операций внутри логического подграфа общего набора данных. В графовой базе данных такое множество операций можно свернуть в набор более крупных, тесно связанных операций. Кроме того, при нативном хранении графа для выполнения каждой из операций требуется меньше вычислительных затрат, чем для эквивалентной ей реляционной операции. Масштабирование графов при меньших затратах дает тот же результат.

Для примера представьте базу данных издательства, в которой нужно найти последнее произведение заданного автора. В реляционной СУБД для этого нужно использовать соединение таблицы авторов с таблицей произведений по идентификатору автора, а затем упорядочить произведения по дате публикации и ограничить результат последней записью. Эту операцию можно рассматривать как операцию с уровнем сложности $O(\log(n))$, что не так уж и плохо.

Тем не менее, как показано на рис. 6.7, эквивалентная графовая операция имеет сложность $O(1)$, что означает постоянную производительность независимо от размера набора данных. В графе достаточно просто пройти по исходящей взаимосвязи `WROTE` от автора к работе в начале списка (или дерева) опубликованных статей. Чтобы найти более старые публикации, нужно пройти по взаимосвязям `PREV` и пе-

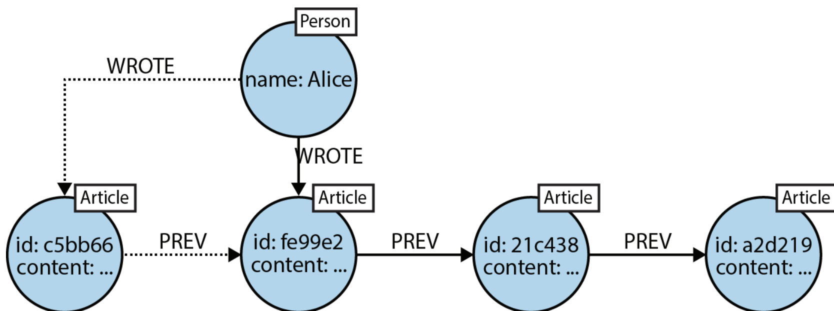


Рис. 6.7 ❖ Постоянное время выполнения операций для системы издательства

ребрать связанный список (или, в качестве альтернативы, рекурсивно обойти дерево). Запись выполняется аналогично, так как новые публикации добавляются в начало списка (или в корень дерева), что тоже обеспечивает постоянство временных затрат. Это выгодное отличие от реляционных СУБД, в частности потому, что естественно поддерживает постоянную производительность при чтении.

Конечно же, требования могут превысить возможности выполнения запросов одним сервером, т. е. пропускную способность операций ввода-вывода. Когда это произойдет, следует просто создать кластер Neo4j для горизонтального масштабирования, обеспечения высокой доступности и высокой пропускной способности чтения. Для типичной рабочей загрузки графа, когда объем чтения намного превышает объем записи, такая архитектура решения станет идеальной.

При превышении емкости кластера можно распределить граф между экземплярами баз данных и включить в приложение логику *сегментирования*. Сегментирование предполагает использование синтетического идентификатора для соединения записей из разных экземпляров базы данных на уровне приложения. Насколько хорошо это будет работать, в значительной степени зависит от формы графа. Некоторые графы отлично подходят для этого. Например, Mozilla использует графовую базу данных Neo4j как часть облачного браузера следующего поколения Pancake. При этом хранение одного большого графа заменяется хранением большого количества маленьких независимых графов, каждый из которых связан с конечным пользователем. Это значительно упрощает масштабирование.

Конечно, не все графы имеют такие удобные границы. Если граф настолько велик, что его следует разбить, но при этом в нем не сущест-

вует никаких естественных границ, нужно использовать тот же подход, который используют NOSQL-хранилища, подобные MongoDB: создание синтетических ключей и привязка записей на уровне приложения с помощью этих ключей плюс некие алгоритмы разрешения ссылок на уровне приложений. Основным отличием от подхода MongoDB является сам характер нативных графовых баз данных, который позволяет увеличить производительность при обходе внутри экземпляра базы данных, в то время как часть обхода, связанная с переходом между экземплярами, будет работать примерно с той же скоростью, что и соединения MongoDB. Но в целом производительность заметно возрастет.

Поиски чаши Грааля масштабируемости графов

Будущее большинства графовых баз данных связано с возможностью раздельного хранения графа на нескольких серверах без участия приложений, т. е. с поддержкой горизонтального масштабирования чтения и записи графа. В общем случае при этом возникает известная проблема NP Hard (проблема недетерминированной полиномиальной сложности), которую не удается преодолеть на практике.

Упрощенное решение проблемы может привести к непредсказуемым скачкам времени выполнения запросов, вызванным неожиданными (медленными) переходами между серверами сети при обходе графа. Разумной реализацией будет определение *минимальных разделяемых частей* в контексте конкретной прикладной области, что сведет к минимуму переходы между серверами[1]. Сейчас в этой области ведется захватывающая исследовательская работа, но на момент написания книги пока нет баз данных, поддерживающих такой механизм.

Итоги

В этой главе мы показали, что графы со свойствами являются отличным выбором для моделирования практических данных. Была рассмотрена архитектура графовых баз данных, причем особый акцент был сделан на архитектуру Neo4j, нефункциональные характеристики реализации и надежность графовых баз данных.

[1] Более подробно на http://en.wikipedia.org/wiki/Cut_(graph_theory).

Глава 7

Интеллектуальный анализ с помощью теории графов

В этой главе будут рассмотрены некоторые из аналитических методов и алгоритмов обработки графовых данных. Теория графов и графовые алгоритмы являются зрелыми и хорошо проработанными областями знаний, и здесь будет продемонстрировано их применение для сложного извлечения информации из графовых баз данных. Учитывая, что читатель, несомненно, знаком с теоретическими основами информатики, следует заметить, что рассмотрение алгоритмов и технологий в этой главе не будет содержать экскурсов в математику, в то же время самостоятельное удовлетворение любопытства только поощряется.

Поиск в глубину и ширину

До рассмотрения сложных аналитических технологий необходимо познакомиться с фундаментальным алгоритмом *поиска в ширину* (*breadth-first search*), который является основным при обходе всего графа. Большинство приведенных в этой книге запросов было по своей природе направлено *в глубину* (*depth-first*), а не *в ширину* (*breadth-first*). То есть они выполняли обход от начального узла к конечному перед повторением аналогичного поиска по другому пути из того же начального узла. Поиск в глубину хорошо подходит для получения отдельных фрагментов информации.

Осмысленный поиск в глубину

Классический алгоритм поиска в глубину является *неосмысленным*. То есть такой алгоритм просто выполняет поиск маршрута, пока не дойдет до конца графа. После этого он возвращается к начальному узлу и пыта-

ется найти другой маршрут. Поскольку графовые базы данных семантически богаты, в них имеется возможность прервать проход по определенной ветви (например, когда найден узел без нужных исходящих взаимосвязей или когда уже пройденная часть маршрута стала «слишком протяженной»). Такой *осмысленный* поиск выполняется быстрее. Именно подобные виды поиска выполнялись Cypher-запросами и модулями обхода на Java в предыдущих главах.

Хотя поиск в глубину и является основной стратегией обхода графа, многие интересные алгоритмы обхода целого графа основаны на поиске *в ширину* (*breadth-first*). При таком подходе граф обследуется по слоям, сначала посещаются узлы, отстоящие от начального узла на 1 взаимосвязь, а затем на 2, на 3 и т. д., пока не будет обойден весь граф. Такую последовательность легко проиллюстрировать, начав с узла с маркировкой `0` (от англ. *origin* – начало) и удаляясь от него слой за слоем, как это показано на рис. 7.1.

Прекращение поиска зависит от конкретного алгоритма, большинство полезных алгоритмов является не только поиском в ширину, но и содержит некоторый элемент осмысленности. Поиск в ширину

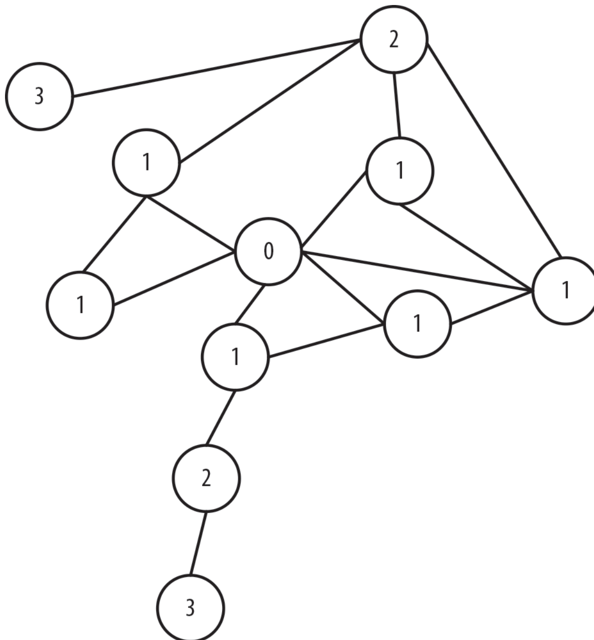


Рис. 7.1 ❖ Порядок выполнения поиска в ширину

часто применяется в алгоритмах поиска маршрутов или систематического обхода всего графа (такие *глобальные графовые* алгоритмы были рассмотрены в главе 3).

Поиск маршрутов с помощью алгоритма Дейкстры

Поиск в ширину поддерживает множество классических графовых алгоритмов, в том числе алгоритм Дейкстры. Дейкстра (для краткости слово «алгоритм» часто опускается) используется для нахождения кратчайшего пути между двумя узлами в графе. Алгоритм Дейкстры не нов, он впервые опубликован в 1956 году, а позднее глубоко исследован и оптимизирован компьютерными специалистами. Алгоритм можно описать следующим образом:

1. Выберите начальный и конечный узлы и добавьте начальный узел в набор *разрешенных* (т. е. во множество узлов, составляющих кратчайший путь, начинающийся с начального узла) со значением 0 (начальный узел удален на расстояние 0 от самого себя).
2. Выполните обход в ширину из начального узла до ближайших соседей и запишите расстояния до каждого из соседних узлов.
3. Выберите соседа с кратчайшим расстоянием (при равных расстояниях выбирается любой из соседей) и отметьте узел как разрешенный, т. е. известен кратчайший путь от начального узла к соседнему.
4. Из всех узлов, входящих в набор разрешенных, перейдите в ближайшие соседние узлы (обратите внимание на использование подхода в ширину) и запишите расстояния от начального узла до новых соседних узлов. Пропускайте соседние узлы, которые уже были разрешены, потому что кратчайший путь до них уже известен.
5. Повторяйте шаги 3 и 4, пока узел назначения не будет помечен как разрешенный.

Эффективность алгоритма Дейкстры

Алгоритм Дейкстры достаточно эффективен, поскольку вычисляет только длины относительно небольшого подмножества возможных маршрутов в графе. После разрешения узла кратчайший маршрут от начального узла до него уже известен, что позволяет безопасно строить последующие маршруты, основываясь на этих сведениях.

Фактически наихудшая из всех известных реализация алгоритма Дейкстры имеет производительность $O(|R|+|N|\,log|N|)$. То есть затраты времени пропорциональны количеству связей в графе плюс количество узлов, умноженное на логарифм количества узлов. Время выполнения оригинального алгоритма $O(|R|\hat{}2)$, т. е. на выполнение затрачивается время, пропорциональное квадрату количества взаимосвязей в графе.

Алгоритм Дейкстры часто используется для поиска кратчайший путей в реальных задачах (например, навигационных). Приведем пример. На рис. 7.2 изображена логистическая карта Австралии. Задача заключается в прокладке кратчайшего маршрута между городами Сидней на восточном побережье (метка *SYD*) и Перт на западном побережье (метка *PER*) через весь континент. Другие крупные города отмечены кодами их аэропортов, маршрут пройдет по многим из них.

Начнем с узла, представляющего Сидней на рис. 7.3, естественно, что кратчайший путь до Сиднея занимает 0 часов, поскольку мы уже там. С точки зрения алгоритма Дейкстры, Сидней уже разрешен, по-

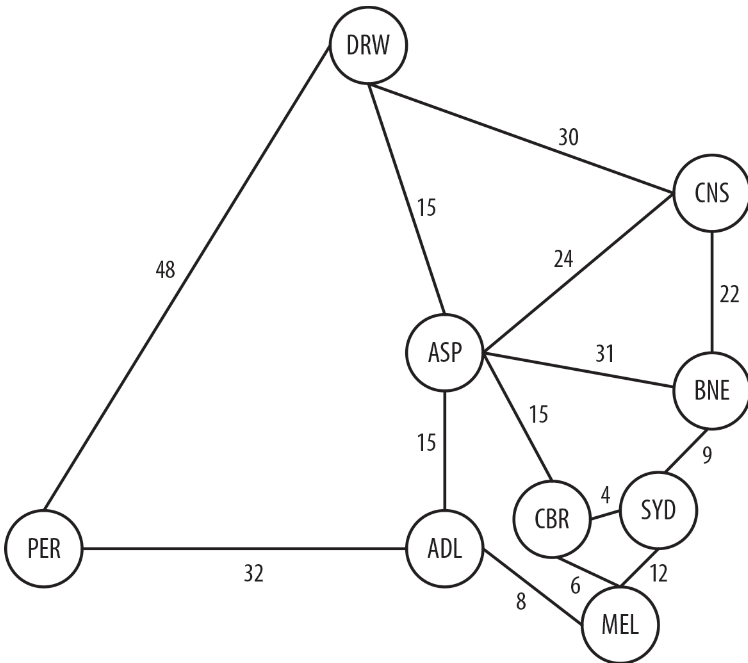


Рис. 7.2 ❖ Логистическое представление Австралии и сети ее магистралей

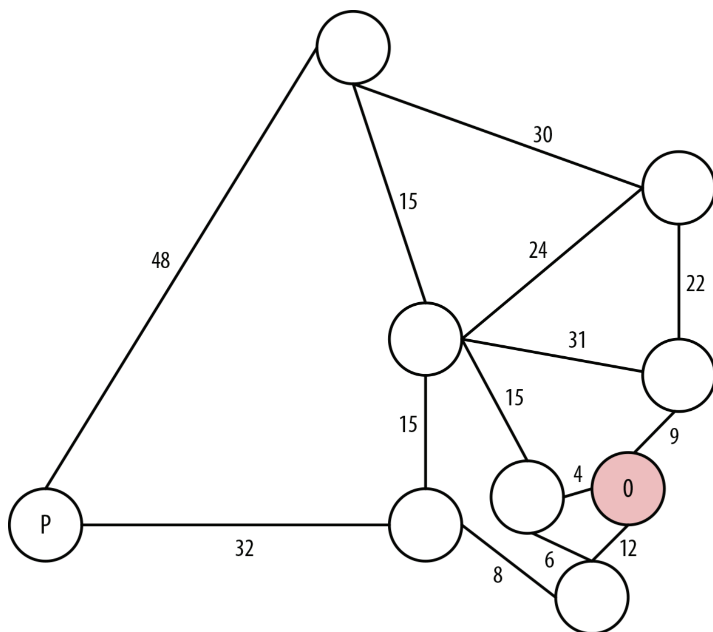


Рис. 7.3 ❖ Кратчайший путь из Сиднея в Сидней занимает, естественно, 0 часов

сколько известен кратчайший путь от Сиднея до Сиднея. Соответственно, отмечаем узел, представляющий Сидней, серым цветом, проставляем длину пути (0) и делаем жирнее границу этого узла, чтобы он оставался выделенным до завершения этого примера.

Продвинемся на один уровень от Сиднея и определимся с городами-кандидатами: Брисбен, в 9 часах пути к северу, Канберра, столица Австралии, в 4 часах пути на запад, и Мельбурн, в 12 часах пути южнее.

Кратчайшим оказывается путь до Канберры, занимающий 4 часа, следовательно, Канберра разрешена, как показано на рис. 7.4.

Следующим доступным из разрешенных узлов будет Мельбурн, в 10 часах пути от Сиднея через Канберру, или в 12 часах от Сиднея напрямую, как уже было известно. Также имеется Алис-Спрингс, в 15 часах пути от Канберры и 19 часах пути от Сиднея, Брисбен, в 9 часах пути прямо из Сиднея.

Соответственно, кратчайшим будет путь из Сиднея в Брисбен, который займет 9 часов, и Брисбен станет разрешенным узлом со значением 9 часов, как показано на рис. 7.5.

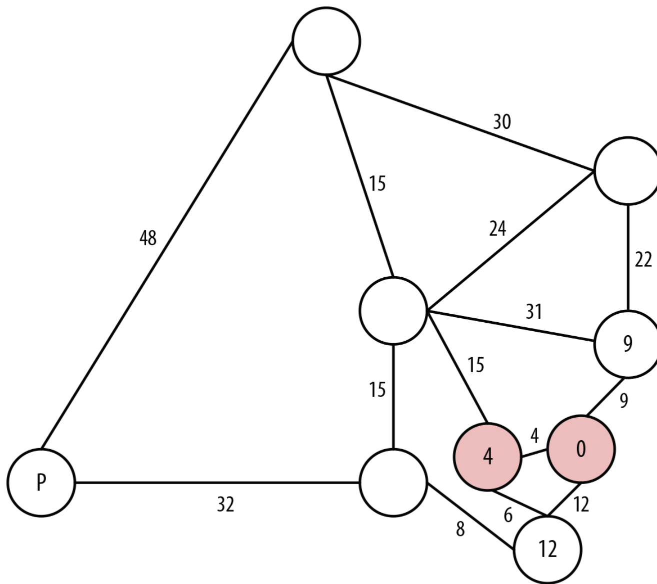


Рис. 7.4 ❖ Канберра является ближайшим к Сиднею городом

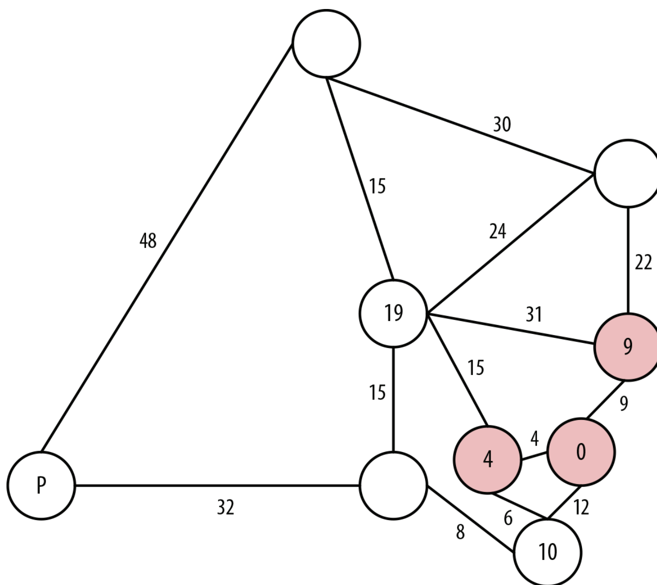


Рис. 7.5 ❖ Брисбен является следующим ближайшим городом

Следующими соседними узлами являются Мельбурн, в 10 часах пути через Канберру или в 12 часах пути прямо от Сиднея по другой магистрали, Кэрнс, в 31 часе пути от Сиднея через Брисбен, и Алис-Спрингс, находящийся в 40 часах пути через Брисбен или 19 часах пути через Канберру.

Соответственно, самый короткий маршрут пролегает через Мельбурн и составляет 10 часов от Сиднея через Канберру. Он короче, чем 12-часовой прямой маршрут. Мельбурн теперь станет разрешенным, как показано на рис. 7.6.

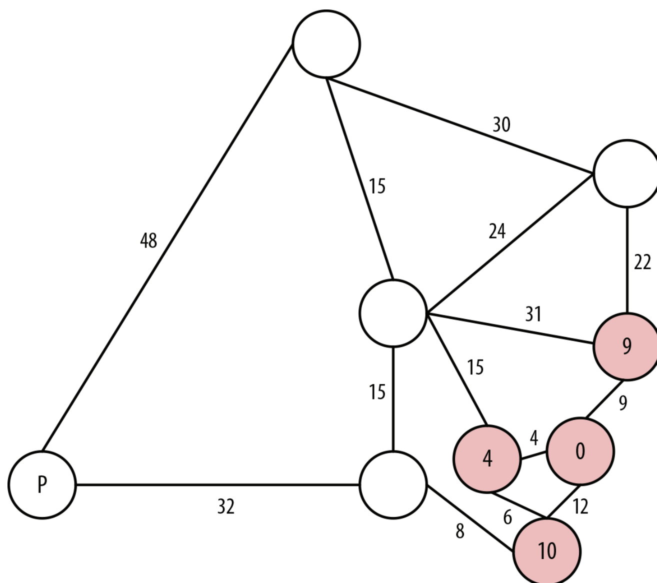


Рис. 7.6 ❖ Достижение Мельбурна, третьего ближайшего города от начального узла Сиднея

На рис. 7.7 следующий слой узлов, соседних с разрешенными, содержит Аделаиду в 18 часах пути от Сиднея (через Канберру и Мельбурн), Кэрнс, в 31 часе пути от Сиднея (через Брисбен), Алис-Спрингс, в 19 часах пути от Сиднея через Канберру или в 40 часах пути через Брисбен. Выбираем Аделаиду и считаем ее разрешенной со значением 18 часов.

Путь Мельбурн → Сидней пропущен, потому что его пунктом назначения является уже разрешенный узел, в данном случае это начальный узел Сиднея.

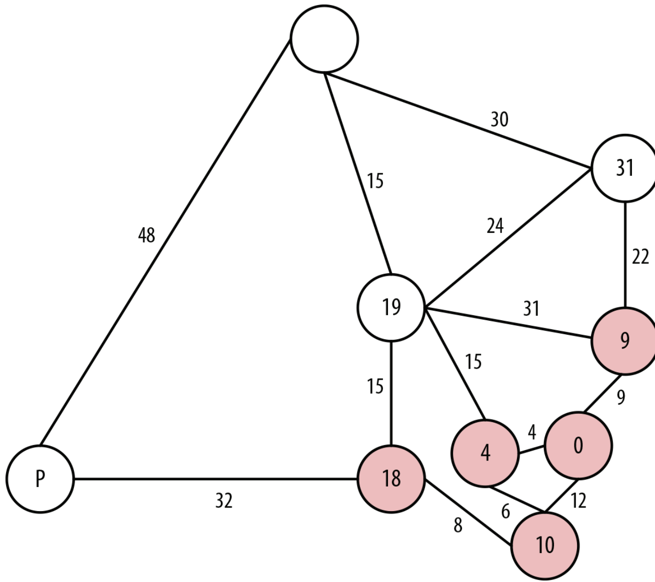


Рис. 7.7 ❖ Разрешение Аделаиды

Следующий слой узлов, соседних с разрешенными, содержит Перт, конечный пункт назначения, в 50 часах пути от Сиднея через Аделаиду, Алис-Спрингс, в 19 часах пути от Сиднея через Канберру или в 33 часах пути через Аделаиду, и Кэрнс, в 31 часе пути от Сиднея через Брисбен.

В этом случае выбирается Алис-Спрингс, потому что ему соответствует текущий кратчайший маршрут, хотя с высоты птичьего полета можно увидеть, что на самом деле окончательным кратчайшим маршрутом станет маршрут из Аделаиды в Перт, и, чтобы убедиться в этом, достаточно спросить дорогу у любого проходящего мимо местного жителя. Соответствующее ему значение будет равно 19 часам, как показано на рис. 7.8.

По рис. 7.9 можно определить следующий слой соседних с разрешенными узлов, состоящий из городов: Кэрнс в 31 часе пути через Брисбен или в 43 часах пути через Алис-Спрингс, Дарвин в 34 часах пути через Алис-Спрингс, или Перт через Аделаиду в 50 часах пути. То есть выберем маршрут в Кэрнс через Брисбен, и Кэрнс станет разрешенным с кратчайшим путем из Сиднея, занимающим 31 час.

Следующий слой узлов, соседних с разрешенными, будет содержать узлы городов: Дарвин в 34 часах пути от Элис-Спрингс, в 61 часе

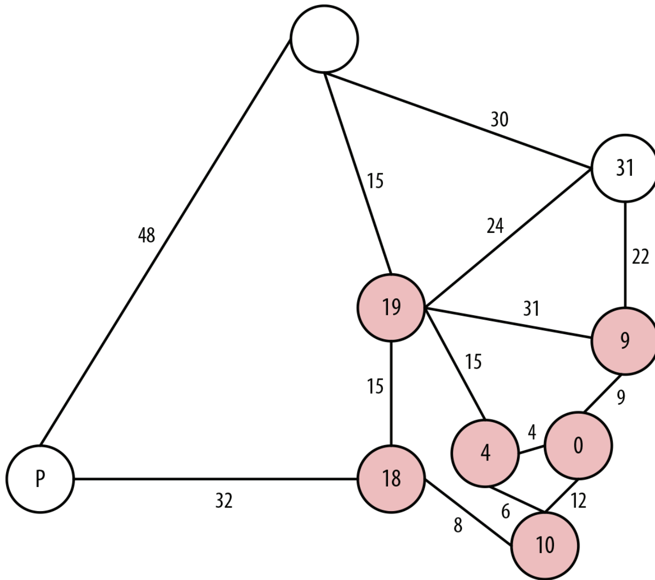


Рис. 7.8 ❖ Выбор «окружного пути» через Алис-Спрингс

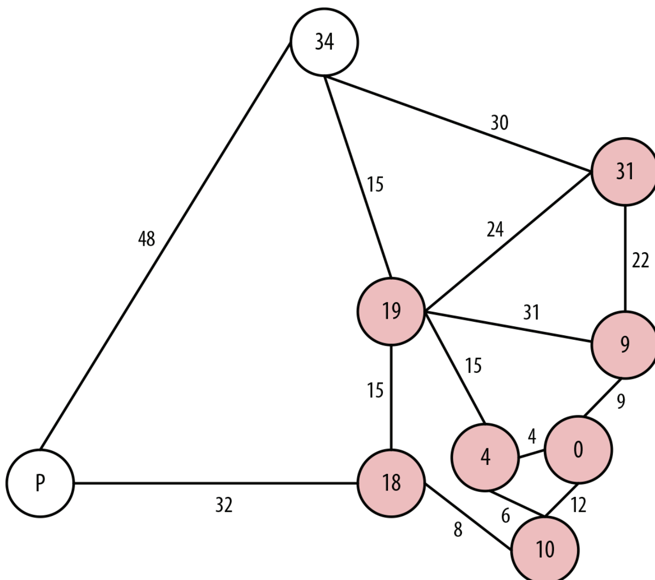


Рис. 7.9 ❖ Возвращение в Кэрнс на восточном побережье

пути через Кэрнс и Перт в 50 часах пути через Аделаиду. Соответственно, выбираем маршрут на Дарвин от Алис-Спрингс со значением 34 часа и считаем Дарвин разрешенным, как показано на рис. 7.10.

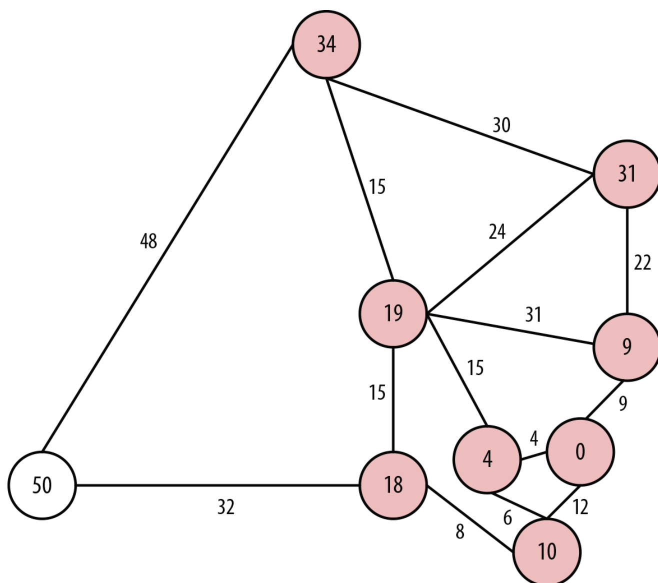


Рис. 7.10 ❖ Разрешение города Дарвин в верхней части Австралии

Наконец, остался только один соседний узел, представляющий сам город Перт, как показано на рис. 7.11. В него можно попасть через Аделаиду, затратив 50 часов, или через Дарвин, затратив 82 часа. Соответственно, выбираем маршрут через Аделаиду и считаем узел города Перт разрешенным с кратчайшим маршрутом из Сиднея в 50 часов пути.

Алгоритм Дейкстры хорошо справляется с задачей, но, так как он выполняет ненаправленный обход, существуют некоторые патологические топологии графов, которые могут вызвать проблемы с производительностью. Тогда будет выполнен обход явно не подходящей области графа, а в некоторых случаях и всего графа. Поскольку каждый из узлов оценивается по отдельности, в относительной изоляции, алгоритм не пытается интуитивно определить кратчайший маршрут.

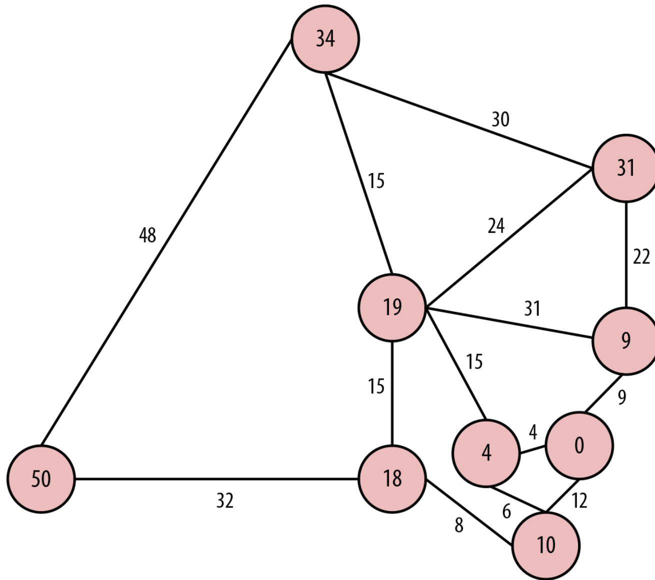


Рис. 7.11 ❖ И наконец, достигнут Перт, всего в 50 часах пути от Сиднея

Несмотря на то что алгоритм Дейкстры успешно справился с прокладкой кратчайшего маршрута между городами Сидней и Перт, любой человек, рассматривая карту, чисто интуитивно не стал бы брать в расчет маршрут к северу от Аделаиды, потому что он явно *выглядит* более протяженным. Если бы имелся некий эвристический механизм, позволяющий начать с поиска в самом перспективном направлении (например, предпочесть запад востоку и юг северу), в этом примере можно было бы избежать обхода городов Брисбен, Кэрнс, Алис-Спрингс и Дарвин. Но поиск по наилучшему направлению выполняется чересчур настойчиво и пытается двигаться вперед, к узлу назначения, даже если имеется препятствие (например, грунтовая дорога) на пути. Мы можем сделать лучше.

Алгоритм A*

Алгоритм A* (произносится как «А со звездочкой») улучшает классический алгоритм Дейкстры. Он основан на наблюдении, что некоторые поиски дают некоторое *осмысление,* и благодаря этому спо-

собен выбирать более удачные пути обхода графа. В нашем примере осмысленный поиск не будет принимать в расчет маршрут из Сиднея в Перт через Дарвин, проходящий через весь континент. Алгоритм A* похож на алгоритм Дейкстры в том, что потенциально может обойти большую часть графа, но он также похож на поиск с учетом наилучшего направления, поскольку использует эвристику при выборе направления. Алгоритм A* сочетает в себе черты алгоритма Дейкстры, отдавая предпочтение узлам, наиболее близким к текущей отправной точке, и черты поиска с выбором наилучшего направления, отдавая предпочтение узлам, ближе всего расположенным к месту назначения, чтобы гарантированно обеспечить оптимальное решение задачи нахождения кратчайшего маршрута в графе.

При применении алгоритма A* затраты времени на путь делятся на две части: часть $g(n)$, затраты на путь от начальной точки до какого-то узла n, и часть $h(n)$, расчетные затраты на путь от узла n к узлу назначения, вычисляемые эвристическим способом (приблизительный расчет). Алгоритм A* балансирует $g(n)$ и $h(n)$ при обходе графа, выбирая в каждой итерации узел с минимальными общими затратами $f(n) = g(n) + h(n)$.

Как видите, алгоритмы поиска в ширину хорошо подходят для прокладывания маршрутов. Но у них также имеются другие применения. Познакомившись с поиском в ширину как с методом обхода всех элементов графа, теперь можно перейти к рассмотрению ряда интересных алгоритмов высокого порядка из теории графов, которые дают представление об интеллектуальном анализе связанных данных.

Теория графов и прогнозное моделирование

Теория графов – зрелая и хорошо изученная область. Ее можно применить к исследованию сетей (или, с нашей точки зрения, связанных данных). Аналитические методы, разработанные специалистами по теории графов, можно использовать в ряде интересных задач. Вооружившись пониманием механизмов обхода низкого уровня, таких как поиск в ширину, приступим к рассмотрению анализа высшего порядка.

Методы теории графов активно применяются в широком диапазоне задач. Они особенно полезны при получении первого представления о новой прикладной области или даже при интуитивном осмыслении прикладной области. Для таких ситуаций существует

ряд методов теории графов и социальных наук, которые можно использовать непосредственно.

В следующих нескольких разделах будут введены некоторые ключевые понятия теории социальных графов. Эти понятия будут приведены в контексте социальной прикладной области и основываться на работах социологов Марка Грановеттера (Mark Granovetter), Дэвида Исли (David Easley) и Джона Клейберга (Jon Kleinberg)[1].

Свойства в графах и теория графов

В большей части работ по теории графов используются несколько иные модели свойств в графах, чем те, что были описаны в этой книге. Эти работы обычно игнорируют направления и метки в графах, заменяя их неориентированными взаимосвязями с одной меткой, взятой из прикладной области (например, `Friend` (друг) или `Colleague` (коллега)).

Мы предпочитаем использовать гибридный подход и далее будем добавлять имена взаимосвязей там, где они помогут прояснить специфичный для прикладной области смысл. Во многих случаях мы будем игнорировать направление взаимосвязей. Но это никак не повлияет на анализ. Графовые структуры придерживаются одних и тех же закономерностей независимо от их конструкции.

Триадические замыкания

Триадические замыкания (triadic closure) – известная особенность социальных графов, согласно которой, если два узла соединены с третьим узлом, высока вероятность создания между ними прямой связи. Это известная социальная закономерность. Если кто-то дружит с двумя людьми, которые не знают друг друга, высока вероятность, что эти два человека в будущем также подружатся. Сам факт, что с ними обоими дружит один и тот же человек, является средством и мотивом к установлению непосредственных дружеских отношений. То есть велика вероятность их встречи при общении с одним и тем человеком и установления доверия между ними на основе доверия и дружбы с одним и тем же лицом. Факт их дружбы с одним и тем же человеком является показателем, что они, скорее всего, похожи в социальном смысле.

Проведя такой анализ, Грановеттер заметил, что подграф имеет *сильное свойство триадического замыкания*, если в нем имеется узел *A*, связанный сильными взаимосвязями с двумя другими узлами *B* и *C*. Узлы *B* и *C* должны быть связаны, по крайней мере, *слабыми* взаимос-

[1] Например, фундаментальная работа Грановеттера (Granovetter) о прочности слабых связей в социальных сетях: http://stanford.io/17XjisT. Работы Исли (Easley) и Клейнберга (Kleinberg): http://bit.ly/13e0ZuZ.

вязями, которые потенциально стремятся стать *сильными*. Это смелое предположение, и оно не всегда выполняется во всех подграфах графа. Тем не менее оно проявляется достаточно часто, особенно в социальных сетях, чтобы быть надежным индикатором для прогнозирования.

> **Сильные и слабые взаимосвязи**
>
> Мы не собираемся определять *сильные* и *слабые* взаимосвязи, потому что они специфичны для каждой прикладной области. В социальной коммуникационной сети можно сказать, что между друзьями существует сильная социальная взаимосвязь, если они обменивались телефонными звонками в течение последней недели, в то время как слабая социальная взаимосвязь объединяет друзей, которые просто отслеживают Facebook-статус друг друга.

Рассмотрим, как сильное свойство триадического замыкания помогает прогнозированию возникновения рабочих взаимоотношений в графе. Начнем с простой организационной иерархии, в которой Алиса (`Alice`) руководит Бобом (`Bob`) и Чарли (`Charlie`) , но между ее подчиненными отсутствует какая-либо взаимосвязь, как показано на рис. 7.12.

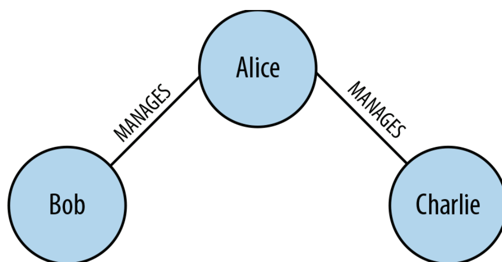


Рис. 7.12 ❖ Алиса руководит Бобом и Чарли

Это довольно странная ситуация для рабочих отношений, потому что весьма маловероятно, что Боб и Чарли незнакомы друг с другом. Как показано на рис. 7.13, были бы они администраторами высокого уровня и, следовательно, занимали одинаковые должности под руководством администратора Алисы, или рабочими по сборке и, следовательно, близкими коллегами, подчиняющимися Алисе как бригадиру, в любом случае следовало бы ожидать, что Боба и Чарли что-то связывает, пусть даже неофициально.

Так как Боб и Чарли работают с Алисой, высока вероятность, что в конечном итоге они будут работать вместе, как показано на рис. 7.13.

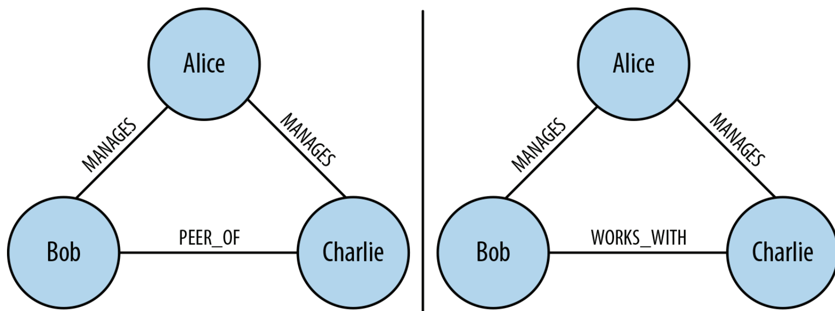


Рис. 7.13 ❖ Боб и Чарли работают вместе под руководством Алисы

Это согласуется с сильным свойством триадического замыкания, которое предполагает, что либо Боб и Чарли занимают одинаковые посты (назовем это *слабой* взаимосвязью), либо Боб работает с Чарли (что означает *сильную* взаимосвязь). Добавление третьей взаимосвязи `WORKS_WITH` (работает вместе с) или `PEER_OF` (сотрудник одного ранга) между Бобом и Чарли замыкает треугольник, отсюда и термин *триадическое замыкание*.

Эмпирические данные из многих областей, в том числе социологии, здравоохранения, психологии, антропологии и даже техники (например, Facebook, Twitter, LinkedIn), указывают, что тенденция триадического замыкания является реальностью и существует. Она подтверждается фактами и аргументами. Но здесь не все сводится к простой геометрии, качество взаимосвязей в графе также оказывает значительное влияние на формирование *стабильных* триадических замыканий.

Структурный баланс

Если вернуться к рис. 7.12, интуитивно можно догадаться, что Боб и Чарли могут стать коллегами (или занимать равнозначные должности) под руководством Алисы. Для примера предположим, что взаимосвязь `MANAGES` имеет негативную окраску (в конце концов, люди не любят, когда ими руководят), тогда как взаимосвязи `PEER_OF` и `WORKS_WITH` являются позитивными (потому что люди, как правило, испытывают симпатию к лицам, занимающим равные с ними должности или работающим вместе с ними).

Из приведенного выше обсуждения принципа сильного триадического замыкания вытекает, что в примере на рис. 7.12, где Алиса

связана взаимосвязями MANAGES с Бобом и Чарли, должно быть выполнено триадическое замыкание. То есть при отсутствии каких-либо прочих ограничений следует ожидать формирования взаимосвязей PEER_OF или WORKS_WITH, а может быть, даже взаимосвязи MANAGES между Бобом и Чарли.

Аналогичная тенденция к созданию триадического замыкания существует, если Алиса связана с Бобом взаимосвязью MANAGES, который, в свою очередь, связан с Чарли взаимосвязью WORKS_WITH, как показано на рис. 7.14. Это звучит вполне правдоподобно: если Боб и Чарли работают вместе, скорее всего, у них общий руководитель – было бы странно, если бы организация позволила Чарли работать без руководящего надзора.

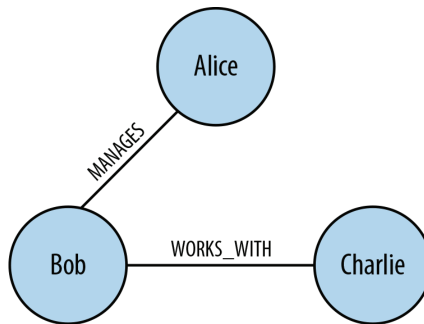


Рис. 7.14 ❖ Алиса управляет Бобом, который работает с Чарли

Однако слепое применение принципа сильного триадического замыкания может привести к некоторым организационным иерархиям, выглядящим довольно странно и неубедительно. Например, если Алиса связана взаимосвязью MANAGES с Бобом и Чарли и Боб связан взаимосвязью MANAGES с Чарли, у нас есть повод для недовольства моделью. Никто не хотел бы быть на месте Чарли, потому что им управляет и его босс, и босс его босса, как показано на рис. 7.15.

Аналогично Боб также чувствует себя неудобно, если им управляет Алиса, хотя он работает вместе с Чарли, который также работает вместе с Алисой. Это не вписывается в организационные уровни, как показано на рис. 7.16. Это также означает, что Боб никогда не сможет спокойно выпустить пар, высказав все, что думает о стиле руководства Алисы, находясь среди лиц, равных ей по должности.

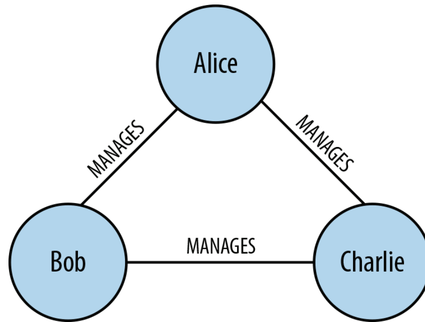


Рис. 7.15 ❖ Алиса управляет Бобом и Чарли, при этом Боб также управляет Чарли

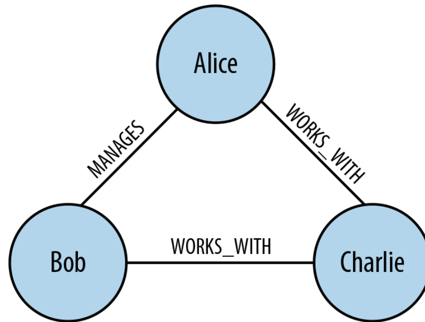


Рис. 7.16 ❖ Алиса управляет Бобом, который работает с Чарли, и в то же время сама работает вместе с Чарли

Неуклюжая иерархия на рис. 7.16, где Чарли – одновременно и ровня боссу, и ровня другому рабочему, вряд ли будет социально правильной, так как Чарли и Алиса будут против нее (желая быть либо боссом, либо рабочим). Аналогично Боб не знает, как обращаться к Чарли – как к руководителю (потому что Чарли и Алиса занимают равные должности) или как к равному себе.

Понятно, что триадические замыкания на рис. 7.15 и 7.16 вряд ли будут выглядеть естественно, поскольку противоречат симметричности структуры и рациональности разделения на слои, которым следует отдавать предпочтение. Такое предпочтение называется в теории графов *структурным балансом*.

Интересно, что имеется и более приемлемое, структурно сбалансированное триадическое замыкание, в котором Алиса связана с Бобом и Чарли взаимосвязями `MANAGES`, а Боб и Чарли являются коллегами и связаны взаимосвязями `WORKS_WITH`, как показано на рис. 7.17.

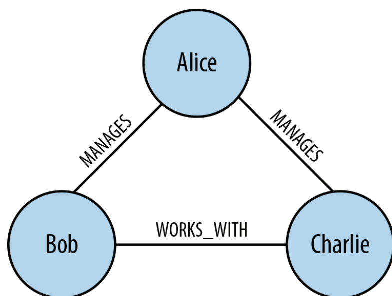


Рис. 7.17 ❖ Коллегами Бобом и Чарли управляет Алиса

Аналогичный структурный баланс наблюдается в триадическом замыкании, где Алиса, Боб и Чарли являются коллегами. В этом варианте они работают вместе, между ними могут возникнуть дружба в социальном смысле, которую порождает товарищество по работе, как показано на рис. 7.18.

На рис. 7.17 и 7.18 триадические замыкания являются идиоматическими и построены либо с помощью трех взаимосвязей `WORKS_WITH`, либо двух взаимосвязей `MANAGES` и одной взаимосвязи `WORKS_WITH`. Все они являются *сбалансированными* триадическими замыканиями. Чтобы разобраться, что именно означают сбалансированные и несбалансированные триадические замыкания, добавим в модель семан-

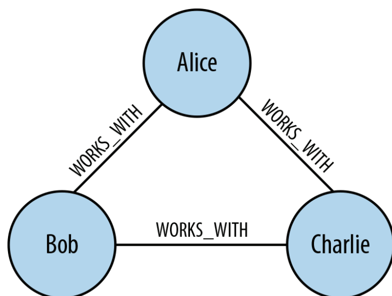


Рис. 7.18 ❖ Алиса, Боб и Чарли коллеги

тику. Определим, что взаимосвязи WORKS_WITH являются позитивными в социальном смысле (так как коллеги тратят много времени на взаимодействие), в то время как взаимосвязи MANAGES являются негативными, потому что руководители тратят меньше времени на взаимодействие с подчиненными.

С учетом этого нового измерения, позитивности и негативности уже можно ответить на вопрос: «Что же такого особенного в сбалансированных структурах?» Очевидно, что сильные триадические замыкания срабатывают, но не всегда. Здесь свою лепту вносит понятие *структурного баланса*. Структурно сбалансированное триадическое замыкание состоит из всех позитивных взаимосвязей (в данном случае WORKS_WITHили PEER_OF) или двух негативных взаимосвязей (в данном случае MANAGES) и одной позитивной взаимосвязи.

Это можно часто наблюдать в реальном мире. Если у кого-то есть два хороших друга, социальный прессинг будет устремлен на превращение их в хороших друзей. Весьма маловероятно, что эти два друга станут врагами, потому что это создаст напряженность между всеми тремя друзьями. Ни один из друзей не сможет выразить свою неприязнь к другому, поскольку те дружат между собой! Учитывая эту напряженность, существует единственный выход – вся группа решит свои разногласия и станет хорошими друзьями. Это сменит несбалансированное триадическое замыкание (две позитивные взаимосвязи и одна негативная) на сбалансированное замыкание, потому что все взаимосвязи станут позитивными, подобно схеме на рис. 7.18, где Алиса, Боб и Чарли – коллеги.

Так же вероятен (хотя и менее приятен) результат, когда другу придется принять одну из сторон в споре между нашими «друзьями» и будут созданы две негативные взаимосвязи, смыкающиеся на одном человеке. Теперь становится возможен обмен репликами о взаимной неприязни к бывшему другу, и замыкание снова становится сбалансированным. Это отражено в организационном примере, где Алиса, руководя Бобом и Чарли, становится ему, по сути, врагом, как показано на рис. 7.17.

Получение организационных данных из реального мира

Не следует создавать графы, основываясь на анализе организационных схем, поскольку они дают лишь статическое и часто непоследовательное представление о функционировании организации. Практичный и актуальный подход заключается в создании графа на основе анализа истории переписки по электронной почте между сотрудниками компании.

Эти взаимодействия легко сохранить в виде графа и затем извлекать из графовой базы данных, что даст возможность выполнить прогнозный анализ эволюции организационной структуры с помощью поиска воз-

можностей для создания сбалансированных замыканий. Такие структуры указывают как на положительные моменты, например на то, как сотрудники группируются для получения успешного результата, так и могут свидетельствовать о противоправной деятельности, когда некоторые сотрудники собираются по темным углам для совершения корпоративного мошенничества! В любом случае, прогнозные возможности графа позволят эффективно решать подобные вопросы.

Сбалансированные замыкания добавляют еще одно измерение к прогнозным способностям графов. Выполнив поиск возможностей для создания сбалансированных замыканий в графе, даже если он имеет большой размер, можно подготовить структуру графа для точного прогнозного анализа. Но можно пойти и дальше, и в следующем разделе будет введено понятие *локальных перемычек*, которые предоставляют ценную информацию о коммуникационном потоке в организации, и эти сведения обеспечат возможность адаптации к будущим вызовам.

Локальные перемычки

Организации, состоящие всего из трех людей, встречаются нечасто, и графы в этом разделе следует воспринимать как подграфы, представляющие фрагменты иерархии большой организации. Модели управления крупной организацией будет соответствовать гораздо более сложная графовая структура, но к ней можно будет применить и другие эвристические методы, помогающие разобраться в прикладной области. После объединения всех частей организации в граф можно проанализировать его глобальные свойства на основании локального принципа сильного триадического замыкания.

На рис. 7.19 изображена противоречивая схема, демонстрирующая две внутренние группы в организации, подчиненные, соответственно, Алисе (`Alice`) и Давине (`Davina`). Некоторую несогласованность структуре придает тот факт, что Алиса руководит не только командой, состоящей из Боба (`Bob`) и Чарли (`Charlie`), но также Давиной. Хотя такое вполне возможно (на Алису может быть возложена и такая обязанность), но организационная структура чисто интуитивно кажется несогласованной.

С точки зрения теории графов, такая ситуация также маловероятна. Алиса связана двумя сильными взаимосвязями `MANAGES` с Чарли (и Бобом) и Давиной, поэтому возникает естественное желание создать триадическое замыкание, добавив взаимосвязь `PEER_OF` между Давиной и Чарли (или Бобом). Но Алиса также участвует в *локальной перемычке* с Давиной, являющейся единственной связью между

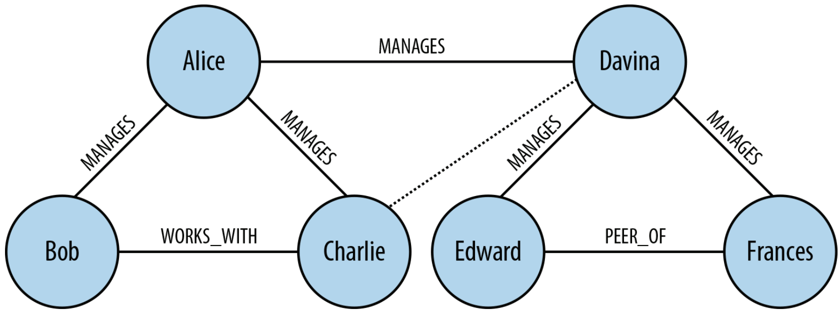


Рис. 7.19 ❖ Обязанности Алисы ведут к перекосу управления

группами внутри организации. Наличие взаимосвязи MANAGES между Алисой и Давиной указывает на необходимость такого замыкания. Эти две особенности – локальная перемычка и строгое триадическое замыкание – вступают в противоречие друг с другом.

Но если Алиса и Давина занимают равнозначные должности (слабая взаимосвязь), принцип строгого триадического замыкания не срабатывает, так как имеется только одна сильная взаимосвязь MANAGES с Бобом (или Чарли), и противоречие с локальной перемычкой пропадает, как показано на рис. 7.20.

Что интересно, локальная перемычка представляет связующий канал между группами внутри организации. Такие каналы имеют чрезвычайно большое значение для жизнедеятельности предприятия. В частности, для нормальной работы компании необходимо, чтобы локальные перемычки активно работали, или следует отслеживать локальные перемычки, чтобы не допустить злоупотреблений (хищений, мошенничества и т. д.).

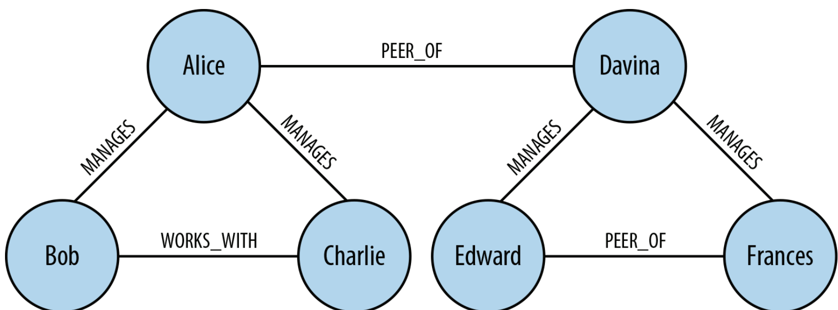


Рис. 7.20 ❖ Алиса и Давина соединены локальной перемычкой

Поиск новой работы

Понятие слабых связей особенно актуально в алгоритмах поиска работы (в социальной сети). Самым любопытным в поиске работы является возможность получить хорошие советы скорее от дальнего знакомого, чем от близкого друга.

Почему так? Наши близкие друзья разделяют наши взгляды (они находятся в том же *компоненте графа*), имеют аналогичный нашему доступ к данным по занятости и подобное нашему мнение об этих рабочих местах. Друг, отделенный локальной перемычкой, явно находится в иной социальной сети (в другом компоненте графа), соответственно, с иным доступом к рабочим местам и другим мнением о них. Так что, если вы собираетесь найти работу, взгляните на локальные перемычки, потому что там вы найдете людей, обладающих сведениями, которых нет у вас и у ваших близких друзей.

Особенность локальных перемычек как слабых звеньев (`PEER_OF` в нашем случае) является весьма распространенным явлением во всех социальных графах. Это значит, что существует возможность проводить прогностические анализы развития организации на основе эмпирических понятий локальных перемычек и сильного триадического замыкания. Итак, получив любой организационный граф, можно определить, как вероятнее всего будет развиваться структура прикладной области, и планировать необходимые действия.

Итоги

Графы являются действительно замечательными структурами. Они уже сотни лет служат объектами научного математического исследования. Но только в настоящее время их стали применять в личной, социальной и деловой сфере. После появления современных графовых баз данных технологии их использования стали открытыми и доступными каждому. Возможности же их бесконечны.

Эта книга наглядно показала, что для использования графовых алгоритмов и аналитических методов не требуется знание теории графов. Достаточно разобраться в том, как *применять* их для достижения своих целей. Эту книгу нужно рассматривать как призыв к действию: осваивайте графы и графовые базы данных. Вооружитесь знаниями о моделировании графами, об архитектуре графовых баз данных, о разработке и внедрении проектов, основанных на графовых базах данных, и о применении графовых алгоритмов для сложных прикладных задач, и сделайте вашу следующую информационную систему по-настоящему новаторской.

Приложение A

Обзор NOSQL-баз данных

В последние годы наблюдается стремительный взлет популярности семейства технологий хранения данных, известных как *NOSQL* (дерзкий акроним от *Not Only SQL* (Не только SQL), или акроним от еще более категоричного *No to SQL* (Нет SQL)). Но сам по себе термин NOSQL означает лишь, что такие хранилища данных не являются SQL-ориентированными реляционными базами данных, а интересным и полезным набором разнообразных технологий хранения, имеющих множество эксплуатационных, функциональных и архитектурных характеристик.

Что послужило причиной создания этих новых баз данных? Какие задачи они призваны решить? Здесь будут рассмотрены некоторые из проблем обработки данных, которые возникли в течение последнего десятилетия. Далее будут описаны четыре семейства NOSQL-баз данных, в том числе графовые.

Движение NOSQL

Исторически сложилось так, что большинство веб-приложений корпоративного уровня использует реляционные базы данных. Но в последнее десятилетие мы столкнулись с настолько большими объемами данных, которые так быстро меняются и так разнообразны по своей структуре, что с ними невозможно работать, используя традиционные реляционные СУБД. Движение NOSQL создано для решения этой проблемы.

Неудивительно, что количество хранимых данных резко возросло, и их *объем* стал основной движущей силой использования NOSQL-хранилищ. Объем можно определить просто как *размер наборов хранимых данных*.

Как известно, большие наборы данных очень сложно хранить в реляционных базах данных. В частности, время выполнения запросов увеличивается вместе с увеличением размеров таблиц и ростом числа соединений (так называемая *болезнь соединений*). И это не вина самих баз данных. Это один из аспектов, лежащий в основе модели данных, заключающийся в извлечении множества возможных результатов запроса с последующей их фильтрацией для получения лишь необходимых.

Чтобы избежать соединений и сопутствующих им болезней и тем самым улучшить обработку очень больших наборов данных, в NOSQL-мире предложено несколько альтернатив реляционной модели. Хотя они лучше справляются с обработкой очень больших наборов данных, эти альтернативные модели, как правило, менее наглядны, чем реляционная (за исключением графовой модели, которая является даже *более* наглядной).

Но объем – это не единственная проблема, с которой сталкиваются современные веб-системы. Помимо своего большого объема, современные данные обычно очень быстро изменяются. *Скорость* – это темп изменения данных с течением времени.

Скорость редко остается статичной. Внутренние и внешние изменения системы и контекста ее использования могут оказать значительное воздействие на скорость. В сочетании с большим объемом данных переменная скорость требует от хранилища не только справляться с устойчиво высоким уровнем объемов записи, но и оставаться работоспособным при пиковых нагрузках.

Существует еще один аспект скорости – скорость изменения структуры данных. Другими словами, вдобавок к изменению значений определенных свойств может меняться общая структура элементов, определяющих эти свойства. Это обычно происходит по двум причинам. Первой является динамизм прикладной области. При изменениях в прикладной области меняются и потребности в данных. Во-вторых, сбор данных часто является экспериментальным процессом. Некоторые свойства создаются «на всякий случай», другие добавляются позднее в связи с изменением требований. Те, что оказались нужными, остаются, прочие выбрасываются. Оба этих аспекта в реляционном мире вызывают проблемы, поскольку большой объем записи привносит высокие расходы обработки, а высокая изменчивость схем сопровождается высокими эксплуатационными затратами.

Хотя позднее на чашу весов были добавлены и прочие вполне обоснованные причины, но решающей причиной стало осознание, что

данные гораздо более разнообразны, чем данные, с которыми обычно имеет дело реляционный мир. Существенным аргументом является мысль об обилии пустых значений в таблицах и проверок на существование значений в коде. А последние сомнения прогнало *разнообразие*, т. е. степень регулярности или нерегулярности структуры данных, плотность или разреженность, связанность или разделенность.

ACID или BASE

Первое знакомство с базами данных NOSQL часто происходит в хорошо знакомом контексте реляционных баз данных. Понятно, что данные и модели запросов будут другими (в конце концов, отсутствует поддержка SQL), но модели согласования данных, используемые NOSQL-хранилищами, также весьма отличаются от тех, что используются реляционными базами данных. Разные NOSQL-базы данных используют разные модели согласования для поддержки разных объемов, скоростей изменения и разнообразия данных, упомянутых выше.

Давайте рассмотрим, какие функции согласованности помогают обеспечить безопасность хранимых данных и какие компромиссы допускаются при использовании (большинства) NOSQL-хранилищ[1].

В мире реляционных баз данных общеизвестны ACID-транзакции, некоторое время являвшиеся эталоном. Гарантии ACID обеспечивают безопасную среду для обработки данных:

- атомарность (Atomic) – все операции в транзакции либо успешны, либо для всех них выполняется откат;
- согласованность (Consistent) – по окончании транзакции база данных является структурно согласованной;
- изолированность (Isolated) – транзакции не мешают друг другу. Спорные ситуации разрешаются базой данных так, что транзакции выполняются последовательно;
- долговечность (Durable) – результаты применения транзакции не должны теряться, даже при сбоях.

Эти свойства означают, что сразу по завершении транзакции ее данные согласуются (так называемая *согласованность записи*) и записываются на диск (или на диски, или в разные области памяти). Это

---

[1] База данных RavenDB, основанная на .NET, уже нарушила традицию агрегированных хранилищ относительно поддержки ACID-транзакций. Как будет упомянуто в этой книге, свойства ACID поддерживаются и многими графовыми базами данных.

отличная абстракция для разработчиков приложений, но требует сложных блокировок, которые могут вызвать логическую недоступность, и, как правило, считается излишне тяжеловесным шаблоном в подавляющем большинстве прикладных областей.

Для многих прикладных областей ACID-транзакции являются излишне пессимистичными. В мире NOSQL ACID-транзакции вышли из моды, поскольку хранилища смягчили требования к немедленной согласованности, актуальности данных и их точности, чтобы получить другие преимущества, такие как масштабируемость и гибкость. Вместо ACID возник другой популярный подход *BASE*, описывающий принципы более оптимистичной стратегии хранения:

- *обычно доступно* (Basic availability) – хранилище доступно большую часть времени;
- *гибкое состояние* (Soft-state) – хранилища не обязаны соблюдать очередность записей, и разные реплики не должны постоянно согласовываться;
- *отложенная согласованность* (Eventual consistency) – хранилища достигают согласованности с некоторой задержкой по времени (например, позже, во время чтения).

Принципы BASE значительно слабее гарантий ACID, и между ними нет прямого соответствия. Значения в BASE-хранилище доступны (потому что это является основой масштабирования), но это не предлагает гарантированной согласованности реплик при записи. BASE-хранилища обеспечивают менее строгий контроль: данные будут согласованы позднее, вероятнее всего, во время чтения (как, например, в Riak), или всегда будут согласованы, но только для определенных фиксаций, обработанных последними (как, например, в Datomic).

Разработчики должны учитывать такую свободную поддержку согласованности и уделять больше внимания согласованности данных. Им следует познакомиться с методами BASE конкретного хранилища и научиться работать в рамках его ограничений. На практике в каждом конкретном случае делается выбор, приемлема ли возможная противоречивость данных или же нужно потребовать от базы данных обеспечить непротиворечивость при чтении, согласившись с возникающими при этом задержками. (Чтобы гарантировать непротиворечивое чтение, базе данных необходимо сравнить все реплики элемента данных и в случае их несогласованности выполнить корректирующую обработку.) При разработке это добавляет сложностей, по сравнению с применением транзакций, которые берут на себя

обязанности по достижению согласованности, но это не является неразрешимой проблемой, просто это требует дополнительных усилий.

Секторы NOSQL

Познакомившись с базовой моделью согласованности данных в NOSQL-хранилищах, можно перейти к рассмотрению использования многочисленных моделей данных. Для большей определенности нами разработана простая классификация, изображенная на рис. A.1. Эта классификация делит современную область NOSQL на четыре сектора. Хранилища в каждом секторе предназначены для разных типов функционального применения, хотя и нефункциональные требования также могут сильно повлиять на выбор базы данных.

В следующих разделах будут отдельно рассмотрены каждый из этих секторов, особое внимание будет уделено характеристикам их моделей данных, особенностям их эксплуатации и побудительным причинам их выбора.

Хранилища документов

Базы данных документов предоставляют достаточно знакомую разработчикам парадигму, используемую для работы с иерархически структурированными документами. Хранилище документов и способы извлечения документов из него напоминают электронный шкаф. Документы, как правило, включают в себя таблицы и перечни, составляющие естественную иерархию, для работы с которой обычно используются такие форматы, как JSON и XML.

В простейшем случае сохранение и чтение документов могут основываться на идентификаторах. При условии, что приложение обеспечивает нужные идентификаторы (например, имена пользователей), хранилище документов может работать подобно хранилищу пар ключ-значение (с которым мы познакомимся чуть позже). Но в общем случае хранилища документов ориентируются на индексы для облегчения доступа к документам по их атрибутам. Например, в области электронной коммерции можно использовать индексы различных видов предлагаемых товаров, выставленных на продажу, как показано на рис. A.2. В целом индексы применяются для получения наборов связанных с ними документов из хранилища и использования в приложении.

Подобно индексам в реляционных базах данных, индексы в хранилище документов увеличивают эффективность чтения за счет умень-

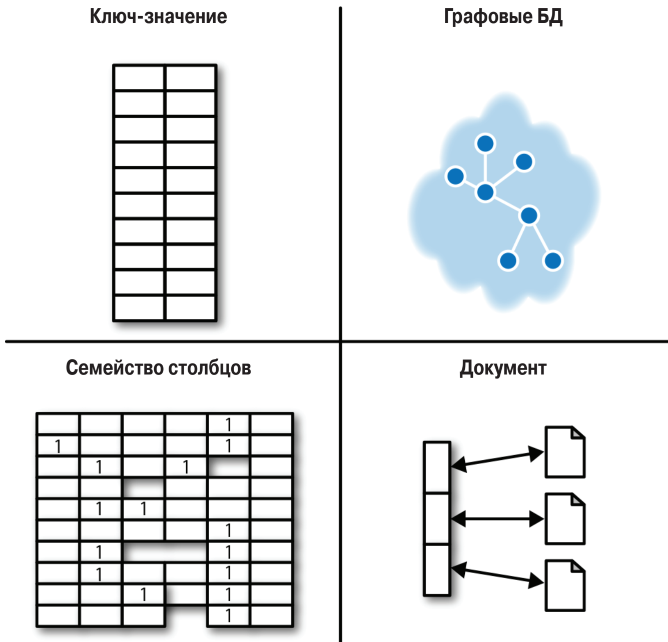


Рис. А.1 ❖ Секторы NOSQL-хранилищ

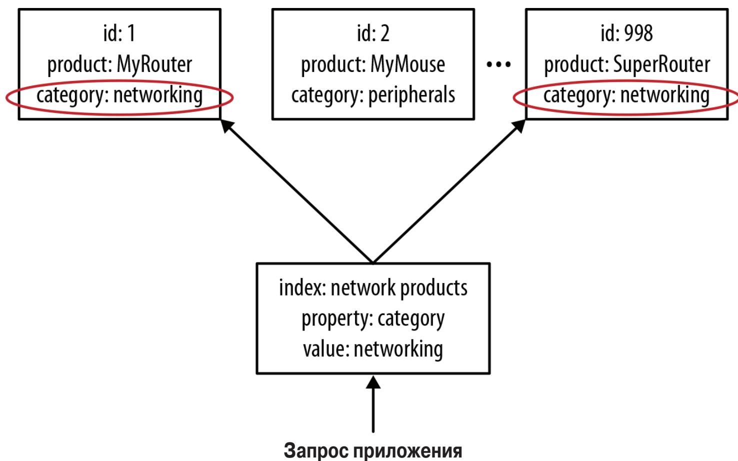


Рис. А.2 ❖ Индексация отображает наборы сущностей в хранилище документов

шения производительности записи. Процесс записи становится более затратным за счет поддержки индексов, но при чтении требуется просканировать меньше записей, чтобы найти нужные данные. Планируя выполнять большое число операций записи, следует иметь в виду, что поддержка индексов значительно снижает общую производительность.

Если нужные запросу данные не были проиндексированы, он выполняется значительно медленнее из-за необходимости сканирования всего набора данных. А это, естественно, затратный процесс, которого следует избегать везде, где это только возможно, и хотя это внутренний процесс, обычной практикой при использовании базы данных документов считается передача этого вида обработки фреймворкам параллельных вычислений.

Поскольку модель данных хранилищ документов является несвязной, они, как правило, обладают хорошими эксплуатационными характеристиками. Они *поддерживают* горизонтальное масштабирование, поскольку отсутствие связей между независимыми друг от друга записями не требует выполнения транзакций при их сохранении с помощью реплик.

Распределение

Большинство баз данных документов требует от пользователей планирования распределения данных между экземплярами для поддержки горизонтального масштабирования. Масштабирование, таким образом, становится одним из конкретных аспектов разработки и эксплуатации. Хранилища пар ключ-значение и семейств столбцов, наоборот, обычно не требуют такого планирования, так как рассматривают подготовку данных для реплик естественной частью своей внутренней реализации. Это иногда, как ни странно, приводится в качестве побудительной причины выбора хранилищ документов, скорее всего, потому, что вызвано (неуместным) умозаключением, что масштабирование с помощью распределения – это не то, чем надо просто с удовольствием пользоваться, а то, что требует долгого и старательного осваивания.

Транзакции баз данных документов уже исторически ограничиваются только уровнем индивидуальной записи. То есть база данных документов гарантирует, что при записи единичного документа будет обеспечена атомарность, причем предполагается, что администратор выбрал безопасный уровень сохранения при настройке базы данных. До поддержки атомарной обработки *набора* документов они еще не доросли. При отсутствии встроенной поддержки транзакций сразу нескольких операций эта обязанность перекладывается на разработчиков приложений, добавляющих компенсирующую эту особенность логику в код приложения.

Поскольку хранимые документы ничем не связаны (кроме индексов), существуют многочисленные оптимистические механизмы управления конкуренцией, которые можно использовать для согласования выполнения одновременных записей в один и тот же документ без необходимости прибегать к строгим блокировкам. Фактически некоторые хранилища документов (например, CouchDB) сделали ключевым пунктом своего предложения следующее: базы данных документов могут иметь несколько главных серверов, которые автоматически реплицируют одновременный доступ, согласовывая состояние с помощью экземпляров, без участия пользователей.

В других хранилищах система управления базами данных может распознавать и согласовывать запись в разные части одного документа или даже использовать моменты времени для согласования нескольких операций записи в единый, логически последовательный результат. Это разумный оптимистический компромисс, поскольку сглаживает отсутствие транзакций с помощью альтернативных механизмов, которые обеспечивают оптимистическое управление хранением с низкой латентностью и высокой пропускной способностью.

Хранилища пар ключ-значение

Хранилища пар ключ-значение являются родственными семейству хранилищ документов, но они ведут свою родословную от базы данных Dynamo Amazon (http://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf). Они функционируют как крупные распределенные структуры данных, хранящие хешированные непрозрачные данные и извлекающие значения по ключу.

Как показано на рис. A.3, объем ключей хешированных данных распространяется по сети посредством множества блоков. По соображениям отказоустойчивости, каждый блок реплицируется на несколько машин. Количество реплик рассчитывается по формуле $R = 2F + 1$, где F – число допустимых отказов. Алгоритм репликации стремится обеспечить согласованность, когда серверы не являются точными копиями друг друга. Он обеспечивает системе баланс загрузки при восстановлении сервера и его блоков. Также он помогает избегать всплесков, которые могут привести к случайным отказам в обслуживании.

С точки зрения клиента, хранилища пар ключ-значение просты в использовании. Клиент сохраняет элемент данных с помощью хеширования специфичного для прикладной области идентификатора (ключа). Хеш-функция разработана таким образом, чтобы обеспечивать равно-

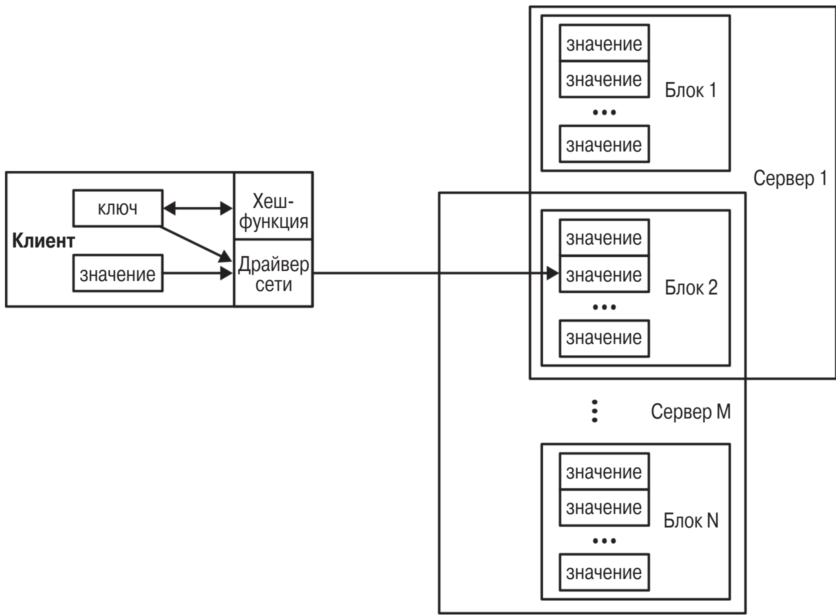


Рис. А.3 ❖ Хранилища пар ключ-значение функционируют подобно распределенным структурам хешированных данных

мерное распределение данных по доступным блокам, обеспечивая тем самым отсутствие перегрузок каждого из отдельных серверов. Учитывая хеширование ключей, клиент может использовать адрес для хранения значений в соответствующем блоке. Клиенты используют аналогичный процесс для извлечения сохраненных значений.

Согласование хеширования

Ошибкам подвержены все компьютерные системы. В надежных системах ошибки маскируются с помощью переключения с неисправных компонентов на резервные. В хранилище пар ключ-значение, так же как в любых других распределенных базах данных, отдельные серверы становятся недоступными при аппаратных или сетевых сбоях. Такие события считаются нормальными, если побочные эффекты сбоев не вызывают проблем после восстановления. Например, если происходит сбой сервера, поддерживающего определенный диапазон хешей, это не должно приводить к отказу сохранения значений в этом диапазоне или вызывать недоступность при выполнении внутренней перестройки.

Это достигается с помощью *согласования хеширования* (http://en.wikipedia.org/wiki/Consistent_hashing). Данная технология позволяет

переназначить диапазон хешей отказавшего сервера следующему доступному северу, не нарушая целостности хранимого набора данных. Фактически во многих случаях переназначается только часть ключей переназначаемого диапазона, а не весь набор. Если отказавший сервер нельзя восстановить или заменить, согласование хеширования позволяет перераспределить только часть общего пространства ключей.

Такая модель позволяет приложениям, сохраняющим и извлекающим данные из хранилища пар ключ-значение, знать (или вычислять) только соответствующий ключ. Хотя набор ключей включает в себя огромное количество всевозможных ключей, на практике ключи определяются областью применения. Имена пользователей и адреса электронной почты, декартовы координаты интересующих областей, номера социального страхования и почтовые индексы – все это является естественными ключами для различных областей. В разумно спроектированной системе вероятность потери данных из-за утери ключа является крайне низкой.

Модель данных хранилищ пар ключ-значение подобна модели данных хранилищ документов. Их отличием является лишь уровень *погружения в суть* данных.

Теоретически хранилища пар ключ-значение не обращают внимания на информацию в значениях. Строго говоря, хранилища пар ключ-значение просто заботятся об обеспечении эффективного хранения и выборки непрозрачных для них данных, перекладывая заботы об их природе и использовании на плечи приложений.

Непрозрачность и доступ к внутренним элементам структур данных

Непрозрачность имеет оборотную сторону. При извлечении элементов данных из хранимых значений клиентам часто приходится извлекать все значение целиком, а затем отфильтровать нежелательные элементы от родительских или параллельных данных. По сравнению с хранилищами документов, которые выполняют такие операции на сервере, данный подход несколько неэффективен.

Но при практической реализации такие различия не всегда четко выражены. Некоторые популярные хранилища пар ключ-значение, например Riak, предполагают определенную прозрачность некоторых типов структурированных данных, хранящихся в форматах XML и JSON. Они также поддерживают некоторые типы данных (называемых CRDT – Core Data Types), которые можно объединять даже в одновременных операциях записи. На уровне готовых продук-

тов различия между хранилищами документом и пар ключ-значение в значительной мере сглажены.

Несмотря на свою простоту, модель данных хранилищ пар ключ-значение, подобно моделям хранилищ документов, обеспечивает некоторую прозрачность данных для разработчиков приложений. Для извлечения наборов полезных сведений из нескольких отдельных записей обычно применяются внешние инфраструктуры, такие как модель распределённых вычислений MapReduce. Эти инфраструктуры отличает высокая латентность, по сравнению с выполнением запросов к хранилищам данных.

Хранилища пар ключ-значение предоставляют определенные преимущества при эксплуатации и масштабировании. Например, база данных Dynamo Amazon, предназначенная для непрерывного обслуживания корзины покупателя, оптимизирована для обеспечения высокой доступности и масштабирования. Команда Amazon позиционирует ее как продолжающую работать, даже «если диски сбоят, сетевые маршруты разорваны, а центры обработки данных разрушены торнадо».

Семейства столбцов

Примером хранилищ семейств столбцов является BigTable Google (http://research.google.com/archive/bigtable.html). Модель данных таких хранилищ основана на разреженных таблицах, чьи строки содержат произвольные столбцы, ключи которых обеспечивают естественную индексацию.

В нашем описании мы будем использовать терминологию распределенной системы управления базами данных Apache Cassandra. Терминология Cassandra не всегда верно интерпретирует понятия BigTable, но она широко распространена, и ее определения достаточно удобны для восприятия.

На рис. А.4 изображены четыре вида строительных блоков, обычно используемых в базах данных семейств столбцов. Простейшей единицей хранения является собственно *столбец* с парой имя-значение. Любое количество столбцов может быть объединено в *суперстолбец*, который придает имя отсортированному множеству столбцов. Столбцы хранятся в строках, и если строка содержит только столбцы, она называется *семейством столбцов*. Если строка содержит суперстолбцы, ее называют *суперсемейством столбцов*.

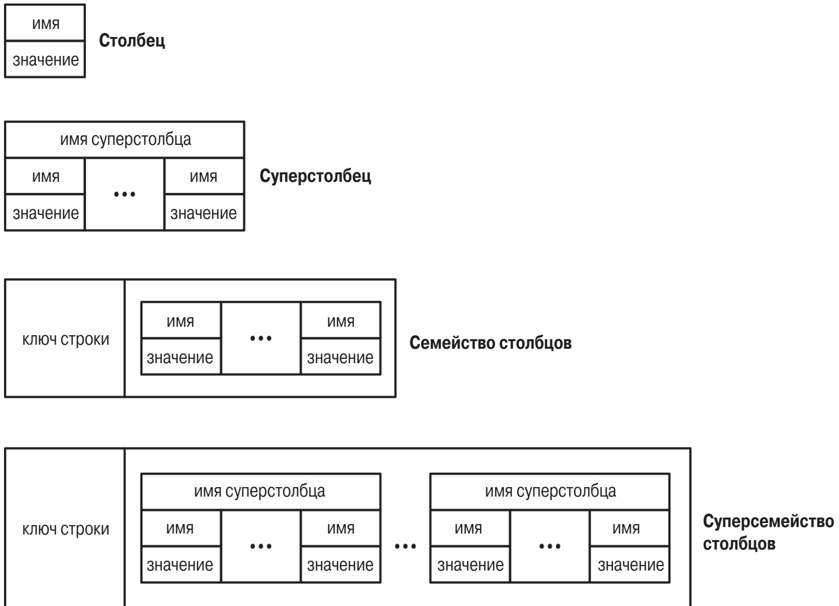


Рис. А.4 ❖ Четыре вида строительных блоков хранилища семейств столбцов

Может показаться странным, что в модели данных столбцов такое внимание уделяется строкам, но строки здесь также важны, потому что обеспечивают структуру хеш-таблиц для нормализации данных. На рис. А.5 изображена модель артиста и его альбомов на основе структуры суперсемейства столбцов. По своей сути, это не что иное, как таблица таблиц.

В базе данных семейств столбцов каждая строка таблицы – это особый самодостаточный объект (например, все об артисте). Семейства столбцов являются контейнерами для связанных элементов данных, таких как имя артиста и сведения о выпущенных им дисках. Внутрь семейства столбцов помещены пары ключ-значение, например дата выхода альбома и дата рождения артиста.

Наиболее полезным здесь является то обстоятельство, что ориентированное на строки представление можно повернуть на 90 градусов и получить представление, ориентированное на столбцы. При том, что каждая строка содержит полную информацию о сущности, а каждый столбец естественным образом определяет конкретный аспект всех

данных. Например, как показано на рис. A.6, «выстраивание в линию» ключей позволяет отобрать все строки, где артист является англичанином. Благодаря этому упрощается извлечение полных данных об артистах из всех строк. Это не взаимосвязи данных в графе, но, по крайней мере, обеспечивает некое моделирование набора связанных объектов.

Суперсемейство столбцов

Билли Брэгг

личные данные	
дата рождения	национальность
19571220	Англичанин

Life's Riot with Spy vs. Spy	
год	награда
1983	золотой диск

Worker's Playtime
год
1988

Ключ строки

Суперстолбец

Столбец

Рис. A.5 ❖ Хранение линейных данных в суперсемействе столбцов

Джо Страммер — личные данные — национальность: англичанин …

Билли Брэгг — личные данные — национальность: англичанин …

Вуди Гатри …

Фил Оукс — личные данные — национальность: американец …

Бен Дрю — личные данные — национальность: англичанин …

Рис. A.6 ❖ Ключи образуют естественный межстрочный индекс в базе данных семейств столбцов

Базы данных семейств столбцов отличаются от хранилищ документов и пар ключ-значение не только более наглядной моделью данных, но и эксплуатационными характеристиками. Например, база данных Apache Cassandra, базирующаяся на инфраструктуре, подобной базе данных Dynamo, является распределенной, масштабируемой и отказоустойчивой архитектурой. Внутри нее функционируют несколько движков, которые хорошо справляются с высокими нагрузками записи, подобными пиковым нагрузкам записи, создаваемыми популярными интерактивными телешоу.

В целом базы данных семейств столбцов являются достаточно наглядными и обладают хорошими эксплуатационными качествами. Но все-таки базам данных *агрегированных хранилищ*, так же как базам данных документов и базам данных хранения пар ключ-значение, не хватает взаимосвязей. Выполнение в них запросов для масштабного охвата данных требует применения в приложении дополнительных внешних инфраструктур.

Запросы или обработка в агрегированных хранилищах

В предыдущих разделах была приведена оценка сходств и различий между моделями данных документов, пар ключ-значение и семейств столбцов. В целом сходств больше, чем различий. Фактически сходство настолько велико, что все три типа вместе иногда называют *агрегированными хранилищами* (http://martinfowler.com/bliki/NosqlDistilled.html). Агрегированные хранилища хранят отдельные комплексные записи, отражающие определяемое прикладной областью понятие агрегата (http://domaindrivendesign.org/).

Хотя каждое из агрегатных хранилищ имеет разную стратегию хранения, в их методах извлечения данных много общего. Для простых запросов они предоставляют индексирование, примитивное связывание и язык запросов. Для более сложных запросов, позволяющих приложениям найти и извлечь множество данных из хранилища, применяется передача их через какую-то внешнюю обработку, такую как фреймворк MapReduce. Это делается, когда требуемого уровня осмысления данных нельзя достичь с помощью только проверки отдельных агрегатов.

Фрейворк MapReduce (http://research.google.com/archive/mapreduce.html), так же как BigTable, является еще одной технологией, предоставляемой Google. Наиболее популярной является реализа-

ция MapReduce с открытым исходным кодом Apache Hadoop и сопутствующие ей экосистемы.

MapReduce представляет собой модель параллельного программирования, которая разбивает данные и параллельно обрабатывает их, перед тем как снова собрать воедино и агрегировать для выдачи конкретных сведений. Например, если использовать MapReduce для подсчета записей в базе данных об американских исполнителях, будут извлечены все записи об исполнителях, на табличном этапе отброшены неамериканские исполнители, а затем на этапе сжатия подсчитаны оставшиеся записи.

Даже при большом количестве серверов и быстрой сетевой инфраструктуре MapReduce может работать довольно медленно. Обычно следует использовать особенности хранилища данных для обеспечения целенаправленной выборки данных, например с помощью индексов или специфичных запросов, а затем передать MapReduce меньший набор данных для получения нужных ответов.

Агрегированные хранилища не предназначены для работы с тесно связанными данными. Можно попытаться использовать их для этой цели, но тогда нужно будет самостоятельно дописать код для заполнения брешей в модели данных. Опыт такой разработки свидетельствует о далеких от совершенства результатах, имеющих плохие эксплуатационные характеристики, особенно при возрастании числа переходов (или «степени» запросов). Агрегированные хранилища хорошо справляются с хранением данных, но они не сильны в решении проблем, требующих понимания взаимосвязанности сущностей.

Графовые базы данных

Графовые базы данных являются веб-системами управления данными с поддержкой методов создания, чтения, изменения и удаления (CRUD) данных, отражающих графовую модель. Графовые базы данных обычно поддерживают систему транзакций (OLTP). Соответственно, они оптимизированы для обработки транзакций и спроектированы с учетом транзакционной целостности и эксплуатационной доступности.

При рассмотрении свойств графовой базы данных полезно учитывать две ее основные особенности:

- технология хранения – некоторые графовые базы данных используют оптимизированные хранилища, предназначенные именно для хранения графов и управления ими. Но не все графовые базы данных используют специализированные хра-

нилища. Некоторые из них сериализируют графовые данные в реляционную базу данных, объектно-ориентированную базу данных или другой тип NOSQL-хранилища;
- технология обработки – некоторые определения графовых баз данных требуют от них поддержки *смежности без индексов*, что означает соединение узлов посредством физических «указателей» в базе данных[1]. В этой книге используется более широкий подход. Любая база данных, которая с точки зрения пользователей ведет себя как графовая база данных (т. е. предоставляет графовую модель данных, обеспечивая операции CRUD), квалифицируется как графовая база данных. Но следует признать и значительное повышение производительности, обеспечиваемое использованием смежности без индексов, и, следовательно, применить термин *нативной обработки графов* для ссылки на графовые базы данных, которые используют смежность без индексов.

Графовые базы данных, в частности нативные, не зависят от индексов, потому что граф сам по себе обеспечивает естественный индекс для смежностей. В нативной графовой базе данных взаимосвязи, прикрепленные к узлу, обеспечивают прямое соединение с другими узлами. Графовые запросы используют эту особенность для обхода графа, следуя указателям. Такие операции выполняются особо эффективно, проходя миллионы узлов в секунду, в отличие от объединения данных посредством глобального индекса, которое работает на несколько порядков медленнее.

Кроме того, обладая собственным подходом к хранению и обработке данных, графовые базы данных также имеют свои модели данных. Популярными являются несколько разных моделей графовых данных, в том числе графы со свойствами, гиперграфы и триплеты. Ниже мы рассмотрим все эти модели.

Графы со свойствами

Графы со свойствами обладают следующими особенностями:
- они хранят узлы и взаимосвязи;
- узлы имеют свойства (пары ключ-значение);
- узлы могут быть помечены одной или несколькими метками;

---

[1] Родригес Марко А. (Rodriguez A. Marko) и Питер Неубауэр (Peter Neubauer). The Graph Traversal Pattern. 2011 (http://arxiv.org/abs/1004.1001) // Graph Data Management: Techniques and Applications. Sherif Sakr and Eric Pardede, 29–46. Hershey, PA: IGI Global.

- взаимосвязи именованы и направлены и всегда имеют начальный и конечный узел;
- взаимосвязи также могут иметь свойства.

Гиперграфы

Гиперграфы – это обобщенная графовая модель, в которой взаимосвязи (так называемые гиперребра) могут связывать любое количество узлов. В то время как модель графа со свойствами позволяет взаимосвязям иметь только один начальный узел и только один конечный узел, гиперграфовая модель допускает наличие любого числа узлов на каждом конце взаимосвязи. Гиперграфы полезны для прикладной области, содержащей преимущественно взаимосвязи многие ко многим. Например, на рис. А.7 показано, что Алиса (`Alice`) и Боб (`Bob`) являются совладельцами трех автомобилей. Это отображается одним гиперребром, в то время как в графе со свойствами для этого понадобится шесть взаимосвязей.

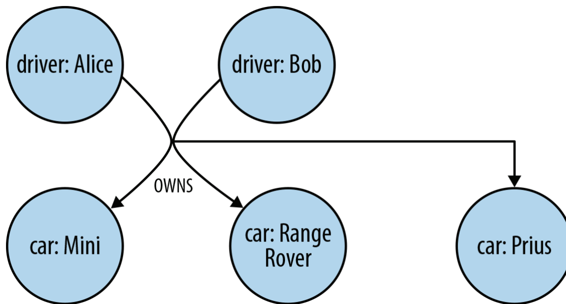


Рис. А.7 ❖ Простой (направленный) гиперграф

Как уже было отмечено в главе 3, графы позволяют моделировать прикладную область в наглядном и удобном для осознания виде, позволяющем с высокой точностью охватить множество нюансов. Хотя теоретически гиперграфы обеспечивают получение точных и богатых информационно моделей, на практике очень легко упустить детали при моделировании. Для иллюстрации этого утверждения рассмотрим граф со свойствами, изображенный на рис. А.8, эквивалентный гиперграфу на рис. А.7.

Особенности приведенного здесь графа потребуют использования нескольких взаимосвязей `OWNS` (владеет) для отображения того, на что в гиперграфе потребовалась только одна взаимосвязь. Но при-

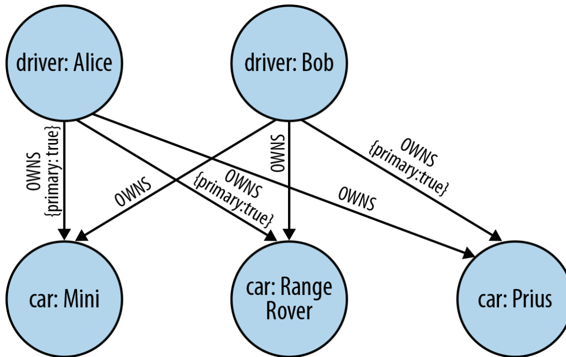


Рис. А.8 ❖ Граф со свойствами с добавленной семантикой

менение нескольких взаимосвязей не только позволяет использовать популярный и очень наглядный метод моделирования, но и произвести точную настройку модели. Например, определить «главного» водителя для каждого транспортного средства (в целях страхования), добавив свойство к соответствующим взаимосвязям, чего нельзя достичь при использовании одного гиперребра.

Так как гиперребра являются многомерными, гиперграфы способны отображать более общие модели, чем графы со свойствами. Тем не менее эти два метода моделирования изоморфны. Это значит, что всегда можно представить информацию, смоделированную в гиперграфе, в виде графа со свойствами (хотя бы и с большим количеством взаимосвязей и промежуточных узлов). Выбор между гиперграфом и графом со свойствами зависит от подхода к моделированию и особенностей проектируемого приложения. Интересно, что для большинства задач выбираются графы со свойствами из-за наилучшего баланса практичности и эффективности моделирования, этим объясняется их огромная популярность в области графовых баз данных. Но в ситуациях, когда нужно охватить метаотношения, эффективно отделяя одни взаимосвязи от других (например, мне нравится, что вам понравился тот автомобиль), гиперграфам требуется гораздо меньше примитивов, чем графам со свойствами.

Триплеты

Триплетные хранилища берут начало в инициативе Semantic Web (Семантическая паутина), занимающейся масштабным представлением знаний путем добавления семантической разметки в ссылки, связывающие веб-ресурсы (http://www.w3.org/standards/semanticweb/).

На сегодняшний день в Интернете такая разметка практически отсутствует, и поэтому выполнение запросов по семантическим слоям встречается редко. Сейчас большинство усилий Semantic Web направлено на сбор полезных сведений и связей между ними в Интернете (или в других более приземленных источниках данных, таких как приложения) и хранение его в *триплетных хранилищах* (*triple stores*) для последующего извлечения посредством запросов.

Триплет представляет собой структуру данных *субъект-предикат-объект* (*subject-predicate-object*). С помощью триплетов можно зафиксировать такие факты, как «Джинджер танцует с Фредом» или «Фред любит мороженое». Сами по себе отдельные триплеты не способны удержать сложной семантики, но при массовом их применении обеспечивают семантически богатый набор данных, содержащих набор знаний и *выводы* связей между ними. Триплетные хранилища обычно применяются для описания ресурсов SPARQL (http://www.w3.org/TR/rdf-sparql-query/) и хранения данных формата RDF (http://www.w3.org/RDF/).

Формат RDF, лингва франка триплетных хранилищ и Semantic Web, может быть сериализован несколькими способами. Следующий фрагмент демонстрирует соединение триплетов для формирования связанных данных формата RDF/XML:

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns="http://www.example.org/terms/">
        <rdf:Description rdf:about="http://www.example.org/ginger">
                <name>Ginger Rogers</name>
                <occupation>dancer</occupation>
                <partner rdf:resource="http://www.example.org/fred"/>
        </rdf:Description>
        <rdf:Description rdf:about="http://www.example.org/fred">
                <name>Fred Astaire</name>
                <occupation>dancer</occupation>
                <likes rdf:resource="http://www.example.org/ice-cream"/>
        </rdf:Description>
</rdf:RDF>
```

Поддержка W3C

Триплетные хранилища различаются по реализации. Внутренняя структура хранилища не обязательно должна быть основана на триплетах, чтобы поддерживать триплеты на логическом уровне. Большинство триплетных хранилищ поддерживает технологии Semantic Web, такие как RDF и SPARQL.

Хотя формат RDF не предоставляет особых преимуществ при сериализации связанных данных, он одобрен W3C и поэтому широко известен и документирован. Языку запросов SPARQL также покровительствует W3C.

В области графовых баз данных существует множество недавно появившихся форматов сериализации графов (например, GEOFF, http://nigelsmall.com/geoff) и несколько языков запросов (например, язык запросов Cypher, http://docs.neo4j.org/chunked/milestone/cypher-query-lang.html, который используется в этой книге). Их ключевой особенностью является отсутствие покровительства уважаемых структур, подобных W3C, хотя они и весьма популярны в рамках сообществ пользователей и поддерживаются производителями.

Триплетные хранилища включены в категорию графовых баз данных, потому что имеют дело с логически связанными данными, во всяком случае после их обработки. Но они не являются «нативными» графовыми базами данных, потому что не поддерживают смежности без индексов и хранение в них не оптимизировано для хранения графов со свойствами. Триплетные хранилища хранят триплеты как независимые артефакты, что позволяет использовать горизонтальное масштабирование при сохранении, но при этом поддерживать быстрый обход взаимосвязей. При выполнении графовых запросов триплетные хранилища создают связанные структуры независимых фактов, что снижает быстродействие запросов. По этим причинам лучшим применением для триплетных хранилищ является анализ, где быстродействие вторично, а не OLTP-системы (интерактивные, онлайн-системы обработки транзакций).

Хотя графовые базы данных предназначены преимущественно для обхода и обработки графов, их можно использовать и в качестве основного хранилища данных RDF/SPARQL. Например, программный интерфейс Blueprints SAIL API обеспечивает RDF-интерфейс нескольких баз данных, в том числе и Neo4j. На практике это означает существование определенного уровня функционального изоморфизма между графовыми базами данных и триплетными хранилищами. Тем не менее каждый тип хранилищ предназначен для различных видов нагрузки, и графовые базы данных оптимизированы для работы с графами и быстрого их обхода.

Предметный указатель

ACID-транзакции, 135, 231
Amazon, 236
Amazon Web Services (AWS), 107
Apache Cassandra, 239
Apache Hadoop, 243

BASE-транзакции, 231
BigTable, 239
Blueprints SAIL API, 248

Cassovary, 23
CEP (Complex Event Processing), 59
Connected, 136
CouchDB, 236
CRDT, 238
CREATE CONSTRAINT, команда, 68
CREATE INDEX, команда, 68, 132
CREATE UNIQUE, фраза, 48
CREATE, фраза, 48
Cypher
  введение в запросы, 66
  достоинства и недостатки, 196
  задание паттернов для поиска, 68
  индексы и ограничения, 67
  инструмент PERIODIC COMMIT, 132
  команда CREATE CONSTRAINT, 68
  команда CREATE INDEX, 68
  обработка результатов, 71
  ограничение сопоставлений, 70
  фраза CREATE, 48
  фраза CREATE UNIQUE, 48
  фраза DELETE, 48
  фраза FOREACH, 48
  фраза MATCH, 47
  фраза MERGE, 48
  фраза RETURN, 48
  фраза SET, 48
  фраза START, 49
  фраза UNION, 48
  фраза WHERE, 48
  фраза WITH, 49
  фразы, 46
  цепочки в запросах, 72

DELETE, фраза, 48

Facebook, 11, 136
FOREACH, фраза, 48

Gartner, 17
Gatling, 121
Giraph, 23
Global Post, 165
Google, 11, 23, 242
Gremlin, 45
Grinder, 121

Hadoop, 32, 243

Introduction To Graph Theory, Trudeau, 17
Introductory Graph Theory, Chartrand, 17

JAX-RS, 103
JMeter, 121

LinkedIn, 136, 143
LRU-K, страничное кэширование, 192

MapReduce, 21, 239, 242
MATCH, фраза, 47
MERGE, фраза, 48
MongoDB, 206

Neo4j
  восстанавливаемость, 199
  встроенная поддержка, 100
  доступность, 200
  другие варианты репликаций, 201
  задержки, 203
  кластеризация, 106
  масштабирование, 202
  мощность, 203
  нефункциональные
  характеристики, 196
  производительность, 204
  серверный режим, 102
  транзакции, 197
  фреймворк Traversal, 37
Neo4j in Action (Партнер и Вукотич), 36
Networks, Crowds, and Markets (Исли
и Клейнберг), 17
NOSQL-хранилища
  ACID или BASE, 231
  гиперграфы, 245
  графовые базы данных, 243
  графы со свойствами, 244
  документов, 30, 233

запросы или обработки
в агрегированных хранилищах, 242
недостатки, 30
обзор, 229
пар ключ-значение, 30, 236
секторы, 233
семейств столбцов, 30, 239
триплеты, 246

OLAP, 20
OLTP, 20
O-алгоритмы, 33
O-нотация, 33

Rails, 55
RDF, 247
RETURN, фраза, 48
Riak, 32
R-Tree, 40

SaaS, 142
SET, фраза, 48
SOR, базы данных, 22
START, фраза, 49

Twitter, 18

UNION, фраза, 48

W3C, 247
WHERE, фраза, 48
WITH, фраза, 49
Write Ahead Log, 198

Авторизация и контроль доступа, 141, 155
модель данных компании
TeleGraph, 156
определение доступности ресурса
администратору, 160
поиск администраторов по учетной
записи, 163
поиск доступных администратору
ресурсов, 159
Агрегированные хранилища, 38, 242
Администратор
доступность ресурса, 160
поиск доступных ресурсов, 159
Алгоритм A*, 217
Анализ графов, 20
Антипаттерны, избегание, 85
Архитектура приложений, 100
балансировка загрузки, 107
встроенная поддержка или сервер, 100
тестирование, 111
Атомарность транзакций, 231

База данных Dynamo, 239
Базовый интерфейс ядра, 193
Балансировка загрузки, 107
Баланс триадических замыканий, 221
Буфера, запись, 107

Вершины, 17
Взаимосвязанные данные
недостатки NOSQL-баз данных, 30
недостатки реляционных баз
данных, 26
преимущества графовых баз
данных, 35
хранение, 35
Взаимосвязи, 17
в NOSQL-хранилищах, 30
в графовых базах данных, 35
в моделях, 90
в реляционнх базах данных, 26
добавление новых, 36
идентификация, 85
или узлы, 90
и узлы, 44
между агрегатами, 31
метки, 64
подробные имена против свойств, 91
свойства, 158
сильные и слабые, 220
Виртуальная Java-машина (JVM), 101,
106, 122
Внешние ключи, 30
Восстанавливаемость, 199
Встроенный режим
JVM, 101
жизненный цикл базы данных, 101
малое время отклика, 101
сборка мусора, 101
управление транзакциями, 101
широкий выбор программных
интерфейсов, 101

Генерация нагрузки, 121
Гибкое состояние, 232
Гиперграфы, 245
Готовность к промышленной
эксплуатации, 135
Графовое моделирование системы
управления, 56
Графовые базы данных (графовые СУБД)
в NOSQL, 243

взаимосвязи, 35
внутреннее усройство, 182
гиперграфы, 245
графы со свойствами, 244
нефункциональные характеристики, 196
определение, 243
преимущества, 23
причины выбора, 134
производительность, 124
разработка приложений, 88
реализация, 206
свойства, 20
триплеты, 246
Графы
введение, 17
запросы на Cypher, 44
метки, 36
моделирование данных, 42
модель с метками и свойствами, 18
области применения, 17
управление версиями, 98
Графы в реальном мире, 134
авторизация и контроль доступа, 141
геоинформационные системы, 138
причины выбора, 134
сети и управление центром обработки данных, 139
социальные рекомендации, 137
социальные сети, 136
типичные примеры использования, 136
управление справочными данными, 139
Группировка узлов, 36

Данные
импорт и массовая загрузка, 128
пакетный импорт, 130
первоначальный импорт, 128
управление центром обработки, 139
Двунаправленные списки, 188
Дейкстры, алгоритм, 209
прокладывание маршрута, 210
эффективность, 209
Денормализация, 53
Джеймс Фаулер, 136

Жизненный цикл базы данных, 101

Задания на извлечение, преобразование и загрузку данных (Extract, Transform, Load, сокращенно ETL), 22
Задача о семи мостах Кенигсберга, 138
Задержки, 203
Запросы
Cypher, 45, 66, 72
в агрегированных хранилищах, 242
выбор подхода, 196
идиоматические, 201
неидиоматические, 201
обратные, 28
по нескольким прикладным областям, 64
разные языки, 45
Значительное ускорение разработки, 135

Идентификаторы, 45
Идиоматические запросы, преимущества, 201
Инкапсуляция, 105
Исли Дэвид, 17

Кластеризация, 106
Клейберг Джон, 219
Клейнберг Джон, 17
Количество одновременных обращений, 127
Коммуникации, авторизация и контроль доступа, 141
Конечный узел, 44
Кросс-модели нескольких прикладных областей, 60
введение в запросы, 66
задание паттернов для поиска, 68
обработка результатов, 71
ограничение сопоставлений, 70
создание графа творчества Шекспира, 64
цепочки в запросах, 72

Масштабирование, 202
задержки, 203
мощность, 203
производительность, 204
Метки
в графе, 36
взаимосвязей, 64
узлов, 44
Модели данных
Global Post, 166
Talent.net, 142
TeleGraph, 156
графовая модель со свойствами и метками, 43
для приложений, 88
идентификация узлов и взаимосвязей, 85

итеративная и поэтапная разработка, 98
модели и задачи, 42
описание с точки зрения потребностей приложения, 89
подробные имена против свойств взаимосвязей, 91
представление типовых комплексных значений, 94
тестирование моделей прикладных областей, 58
узлы или взаимосвязи, 90
эволюция прикладной области, 79
Моделирование данных, избегание антипаттернов, 85
Мощность, 203

Нативная обработка, 21, 183
Начальный узел, 44
Непрозрачные данные, 236
Нефункциональные характеристики
восстанавливаемость, 199
доступность, 200
масштабирование, 202
транзакции, 197

Обычно доступно, 232
Обязательные узлы, 66
Оперативность графовых баз данных, 25
Описание модели с точки зрения потребностей приложения, 89
Оптимизация
для нагрузки, 127
загрузка, 124
избыточность, 124
критерии, 124
производительность, 124
стоимость, 124
экономная организация имен свойств, 191
Ориентированные на столбцы NOSQL-базы данных, 30
Отделение трафика чтения от трафика записи, 107
Отложенная согласованность, 232
Очереди, использование для записи буфера, 107

Пакетный импорт данных, 130
Планирование производственных мощностей, 123
варианты оптимизации, 125
избыточность, 127
критерии оптимизации, 124
нагрузка, 127
производительность, 124
Подробные имена против свойств взаимосвязей, 91
Поиск в глубину, 207
Поиск в ширину, 207
Поиск маршрутов с помощью алгоритма Дейкстры, 209
Поиск новой работы, 228
Политика частоты использования (LFU), 192
Полный перебор, 33
Преголь, 23
Приложения
архитектура, 100
импорт больших объемов данных, 132
итеративная и поэтапная разработка, 98
кластеризация, 106
моделирование данных, 88
моделирование фактов в виде узлов, 92
планирование мощностей, 123
подробные имена против свойств взаимосвязей, 91
представление комплексных значений в виде узлов, 94
разработка, 88
тестирование, 111
узлы или взаимосвязи, 90
хронологическое древо, 95
Проблемы анализа источников электронных писем, 73
Программные интерфейсы (API), 25, 101, 102, 192
Производительность, 134, 204

Разработка, основанная на тестировании, 111
Распределительный кэш, 109
Ребра, 17
Реляционное моделирование, 49
Реляционные базы данных, 26, 29, 49
Репликация
Neo4j, 201
кластеризация, 107
Репрезентативные данные для тестирования, 121
Рефакторинг баз данных, 55

Сборка мусора, 106
Секторы NOSQL, 233
Семейства столбцов, 239
Сильные взаимосвязи, 220

Слабые взамосвязи, 220
Сложные транзакции, 105
Согласование хеширования, 237
Среднее время обращения, 127
Страничное кэширование LRU-K, 192
Суперсемейство столбцов, 239
Суперстолбец, 239

Теория графов, 13
  локальные перемычки, 227
  прогнозное моделирование, 218
  свойства в графах, 219
  структурный баланс, 221
  триадические замыкания, 219
Тесно связанные данные, 13
Тестирование
  приложений, 111
  производительности, 118
  производительности приложений, 120
  серверных расширений, 116
  с помощью репрезентативных данных, 121
Тестирование с помощью репрезентативных данных, 121
Транзакции, 197
Триадические замыкания, 219
Триадическое замыкание, 221
Триплеты, 246
Триплеты RDF, 247
Труде Ричард Дж., 17

Удивительная интерактивность, 135
Узлы, 17
  группировка, 36
  добавление новых, 36
  и взаимосвязи, 44
  идентификация, 85
  или взаимосвязи, 90
  использование в моделях, 90
  метки, 44
  моделирование фактов, 92
  представление комплексных значений, 94
Управление доступом, 141

Фаулер Джеймс, 136
Форматы ответов, 105
Фразы Cypher, 47

Хеширование, 237
Хранение взаимосвязей, 188
Хранилища документов, 233
Хранилища семейств столбцов, 239
Хранилище пар ключ-значение, 236

Цепочки взаимосвязей, 187
Цепочки в запросах, 72

Чартранд Гарри, 17

Шекспир, граф творчества, 64
Широкий выбор API, 105

Эйлер Леонард, 138

Об авторах

Ян Робинсон (Ian Robinson) – соавтор книги «*REST in Practice*» (http://shop.oreilly.com/product/9780596805838.do) (O'Reilly, 2010). Ян работает инженером в компании Neo Technology и занимается распределенной версией базы данных Neo4j. До прихода в команду инженеров занимал должность директора по работе с пользователями Neo Technology, руководя обучением, оказанием профессиональных услуг и обеспечением поддержки пользователей по продуктам Neo, помогая клиентам проектировать и разрабатывать решения на основе графовых баз данных. Ян пришел в Neo Technology из компании ThoughtWorks, где был руководителем внедрения SOA и членом технического консультативного совета технических директоров. Ян часто выступает на конференциях по всему миру по темам, связанным с внедрением технологий графовых баз данных и REST-архитектуры корпоративной интеграции.

Доктор Джим Уэббер (Dr. Jim Webber) – главный научный советник Neo Technology, занимается исследованием новых графовых баз данных и разработкой программного обеспечения с открытым исходным кодом. Ранее Джим посвящал все свое время работе с большими графами, такими как Web, для проектирования распределенных систем, что сделало его соавтором книги «*REST in Practic*», а прежде он написал «*Developing Enterprise Web Services: An Architect's Guide*» (Prentice Hall, 2003). Джим – активный член сообщества разработчиков, регулярно представляющий презентации. Его блог можно найти на http://jimwebber.org, его учетная запись в Twitter @jimwebber.

Эмиль Эифрем (Emil Eifrem) – генеральный директор Neo Technology и соучредитель проекта Neo4j. До основания Neo был техническим директором в Windh AB, где возглавлял разработку весьма сложных информационных архитектур для систем управления корпоративным контентом. Последовательный сторонник открытости исходных кодов, он ведет Neo по пути балансирования между бесплатностью и коммерческой надежностью. Эмиль часто выступает на конференциях и является автором работ, посвященных NOSQL-базам данных.

Заключение

Животное на обложке книги «Графовые базы данных» – европейский осьминог (*eledone cirrhosa*), также известный как малый осьминог, или рогатый осьминог. Родиной европейских осьминогов являются скалистые побережья Ирландии и Англии, но они встречаются также в Атлантическом океане, Северном и Средиземном морях. В основном живут на глубине от 10 до 15 метров, но были замечены и на больших глубинах, около 800 метров. Их отличительными особенностями являются красновато-оранжевая окраска, переходящая внутри в белую, гранулы на кожном покрове и овальная форма мантии.

Европейский осьминог в основном питается крабами и другими ракообразными. Часто попадает в сети рыболовных судов в Средиземном и Северном морях. Количество особей не контролируется, и квот на их вылов нет. Тем не менее их популяции в этих местах в последние годы выросли, в том числе и из-за уменьшения количества крупных хищных рыб.

Европейский осьминог примерно за год достигает размеров от 12 до 40 сантиметров в длину. Продолжительность жизни относительно невелика, менее пяти лет.

По сравнению с самками *осьминогов обыкновенных* (*octopus vulgaris*), самки европейских осьминогов откладывают гораздо меньшее количество яиц, в среднем от 1000 до 5000.

Многие животные, изображенные на обложках книг издательства O'Reilly, находятся на грани исчезновения. Все они важны для нашего мира. Чтобы узнать, как помочь их сохранению, посетите страницу animals.oreilly.com.

Изображение на обложке взято из книги «*Dover Pictorial Archive*». При подготовке обложки использованы шрифты URW Typewriter и Guardian Sans. В качестве шрифта текста использован шрифт Adobe Minion Pro, шрифта заголовка – шрифт Adobe Myriad Condensed и шрифта кодов – шрифт Дальтона Маага (Dalton Maag) Ubuntu Mono.

Книги издательства «ДМК Пресс» можно заказать
в торгово-издательском холдинге «Планета Альянс» наложенным платежом,
выслав открытку или письмо по почтовому адресу:
115487, г. Москва, 2-й Нагатинский пр-д, д. 6А.
При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.
Эти книги вы можете заказать и в интернет-магазине: **www.alians-kniga.ru**.
Оптовые закупки: тел. **(499) 782-38-89**.
Электронный адрес: **books@alians-kniga.ru**.

Ян Робинсон, Джим Вебер, Эмиль Эифрем

Графовые базы данных

Новые возможности для работы со связанными данными

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Научный редактор *Киселев А. Н.*
Переводчик *Рагимов Р. Н.*
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Формат 60×90 1/16.
Гарнитура «Петербург». Печать офсетная.
Усл. печ. л. 16. Тираж 200 экз.

Веб-сайт издательства: www.дмк.рф