Глава 3

Более подробно о моделях данных

В предыдущих главах описана основная особенность большинства баз данных NoSQL — использование агрегатов. Кроме того, показаны разные способы моделирования агрегатов разными агрегатно-ориентированными базами. Хотя агрегаты представляют собой ядро технологии NoSQL, существуют и другие концепции моделирования данных, к описанию которых мы переходим в этой главе.

3.1. Отношения

Агрегаты полезны тем, что они объединяют в одно целое данные, доступ к которым осуществляется одновременно. Однако существует много ситуаций, в которых доступ к связанным данным осуществляется по-другому. Рассмотрим связь между клиентом и всеми его заказами. Некоторые приложения требуют доступ к истории заказов каждый раз, когда они обращаются к данным о клиенте. В этом случае данные о клиенте и его истории заказов целесообразно объединить в один агрегат. Однако другие приложения желают обрабатывать заказы по отдельности и моделируют их как независимые агрегаты.

В этом случае агрегаты заказов и клиента целесообразно разъединить, сохранив между ними определенную связь, чтобы любая операция над заказом могла использовать данные о клиенте. Проще всего установить такую связь, включив идентификатор клиента в данные агрегата заказа. В этом случае, если вам потребуются данные из записи о клиенте, вы прочитаете заказ, узнаете идентификатор клиента и пошлете базе данных другой вызов, чтобы прочитать данные о клиенте. Это вполне работоспособное решение, которое прекрасно подходит для многих сценариев, но база данных не будет знать о связи между данными. Это может оказаться важным, поскольку иногда полезно, чтобы база данных знала о связях между данными.

В результате многие разработчики баз данных, даже хранилищ типа “ключ–значение”, предпринимают усилия, чтобы сделать такие отношения видимыми. Документные хранилища открывают содержимое агрегатов для баз данных, чтобы те формировали индексы и запросы. Например, Riak, хранилище типа “ключ–значение”, позволяет размещать информацию о связи в метаданных, поддерживая частичное извлечение и прослеживание связей.

Важным аспектом связей между агрегатами является их обработка модификаций. Агрегатно-ориентированные базы данных интерпретируют агрегат как единое целое при извлечении данных. Следовательно, атомарность поддерживается только в содержимом отдельного агрегата. Если вы обновляете несколько агрегатов одновременно, то должны самостоятельно реагировать на сбой при обращении к ним. Реляционные базы данных помогают пользователям тем, что позволяют модифицировать несколько записей одновременно в рамках одной транзакции, обеспечивая гарантии безопасности выполнения операций ACID при изменении нескольких строк.

Все это означает, что при обращении к нескольким агрегатам одновременно агрегатно-ориентированные базы данных становятся неудобными. Существует несколько способов смягчения этой проблемы, которые мы исследуем в следующих разделах, но фундаментальное неудобство устранить невозможно.

Следовательно, при работе с данными, между которыми установлено много связей, следует предпочесть реляционные базы данных, а не хранилище NoSQL. Несмотря на этот недостаток агрегатно-ориентированных баз данных, следует помнить, что не все реляционные базы данных великолепно справляются со сложными связями. И хотя в языке SQL вы можете формировать запросы, содержащие операции соединения, с ростом количества соединений ситуация быстро становится запутанной — как с точки зрения языка SQL, так и с точки зрения итоговой производительности.

А сейчас настал удобный момент представить вам другую категорию баз данных, которые часто относят к технологии NoSQL.

3.2. Графовые базы данных

Графовые базы данных — белые вороны в стае баз данных NoSQL. Причиной разработки большинства баз данных NoSQL стала необходимость работать на кластерах, которая привела к агрегатно-ориентированным моделям больших записей с простыми связями. Графовые базы данных появились как решение другой проблемы и поэтому имеют противоположную модель — маленькие записи со сложными связями, как показано на рис. 3.1.

В таком контексте этот граф — не диаграмма, а структура данных с узлами, соединенными ребрами.

На рис. 3.1 показана веб-информация с очень маленькими узлами и многочисленными связями между ними. Работая с этой структурой, мы можем задавать вопросы вроде “найти книгу в категории “Базы данных”, написанную кем-то, чей друг мне нравится”.

Графовые базы данных специально предназначены для хранения такой информации — но в более крупном масштабе, чем можно показать на диаграмме. Они идеально подходят для хранения любых данных, связанных со сложными отношениями, например, социальных сетей, товарных предпочтений или правил приема на работу.

Фундаментальная модель данных графовых баз очень простая: узлы, соединенные ребрами (которые называют также дугами). Помимо этой существенной характеристики, существует много вариаций в моделях данных — в частности, в том, какие

механизмы используются для хранения узлов и ребер. Например, база FlockDB, представляет собой простую совокупность узлов и ребер без какого-либо механизма для дополнительных атрибутов, Neo4J позволяет присоединять Java-объекты в качестве свойств узлов и ребер в неструктурированном виде (см. раздел 11.2, "Функциональные возможности"); а Infinite Graph хранит Java-объекты, являющиеся экземплярами подклассов таких встроенных типов, как узлы и ребра.

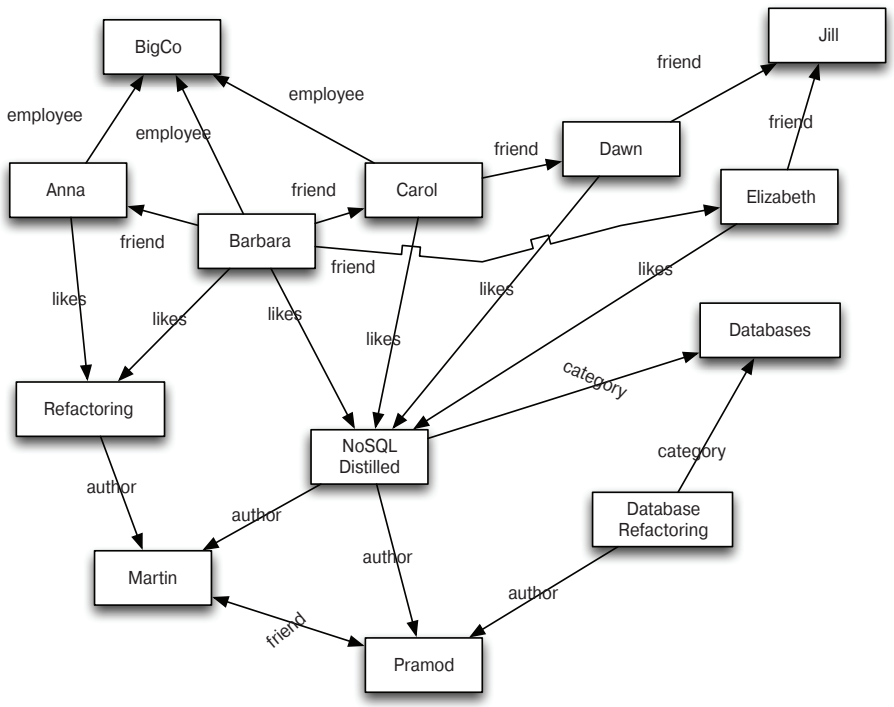

Рис. 3.1. Пример графовой структуры

Как только вы построите граф узлов и ребер, графовая база данных позволит вам послать запрос к сети, способной выполнять операции над запросами, относящимися к таким графам. В этом проявляется важное различие между графовыми и реляционными базами данных. Несмотря на то что реляционные базы данных могут реализовывать связи с помощью внешних ключей, операции соединения требуют навигации, которая может оказаться затратной. Следовательно, в моделях данных с большим количеством связей производительность упадет.

В графических базах данных обход узлов требует очень небольших затрат. В основном это объясняется тем, что графовые базы данных переносят б*о*льшую часть работы, связанной с навигацией по связям, с момента запроса на момент вставки. Это естественно оправдывает себя в ситуациях, когда производительность запроса важнее скорости вставки.

Б*о*льшую часть времени вы ищете данные, перемещаясь по ребрам сети с запросами вроде "назовите мне всех, кто любит Анну и Барбару". Однако вам необходима

отправная точка, поэтому некоторые узлы могут быть индексированы атрибутом, например идентификатором. Таким образом, вы можете начать с поиска идентификатора (например, найти людей с именами Анна и Барбара), а затем начать перемещение по ребрам. Как видим, графовые базы предназначены для ситуаций, в которых б*о*льшую часть времени вы перемещаетесь по связям.

Акцент на связях резко отличает графовые базы данных от агрегатно-ориентированных. Это отличие имеет несколько последствий; такие базы данных чаще работают на одном сервере, а не распределены по кластерам. Транзакции ACID должны охватывать несколько узлов и ребер, чтобы обеспечивать согласованность данных. Единственное, что связывает их с агрегатно-ориентированными базами данных, — отрицание реляционной модели и повышенное внимание специалистов, объясняемое интересом к технологии NoSQL.

3.3. Неструктурированные базы данных

Все базы данных NoSQL являются неструктурированными. Когда вы хотите хранить данные в реляционной базе, сначала определяете схему базы данных, т.е. указываете, какие существуют таблицы и столбцы, и задаете типы данных, которые могут содержаться в этих столбцах. Прежде чем сохранить данные, вы должны иметь схему для этого.

В базах данных NoSQL хранение данных происходит много проще. Хранилище типа "ключ–значение" позволяет хранить данные по ключу. Документная база данных по существу делает то же самое, поскольку она не накладывает ограничений на структуру хранящихся документов. Семейство столбцов позволяет хранить любые данные в любом столбце. Графовые базы данных позволяют свободно добавлять новые ребра, а также новые свойства в узлы и ребра.

Сторонники неструктурированных баз данных подчеркивают их свободу и гибкость. Имея схему, вы должны заранее угадать, что вам потребуется хранить, а сделать это бывает трудно. Используя неструктурированные базы данных, вы можете легко изменять хранилище данных, по мере увеличения знаний о своем проекте. Обнаруживая новые сущности, вы можете легко их добавлять. Более того, если выяснится, что некую сущность вам больше не надо хранить, вы можете просто перестать это делать, не беспокоясь о потере старых данных. Работая с реляционной схемой, вы должны были бы обеспечить сохранность старых данных при удалении столбцов.

Помимо обеспечения удобных изменений, неструктурированные базы данных облегчают обработку *неоднородных данных*, т.е. данных, в которых все записи имеют разные наборы полей. Схема помещает все строки таблицы в "смирительную рубашку". Это может оказаться неудобным, если в разных строках хранятся разные данные. Вы либо останетесь с множеством столбцов, заполненных нулями (т.е. с разреженной матрицей), либо с бессмысленными столбцами вроде `custom column 4`. Неструктурированные базы данных позволяют избежать этого, позволяя каждой записи хранить все, что требуется, — ни больше ни меньше.

Отсутствие структуры выглядит привлекательно. Это позволяет избежать многих проблем, существующих в базах данных с фиксированной схемой, но порождает новые. Если вы только храните данные и выводите их на экран в виде простого списка, состоящего из строк `fieldName: value`, то неструктурированные базы данных — это то, что нужно. Но обычно мы делаем с базами не только это. Как правило, мы пишем программы для обработки данных, которые должны знать, что адрес заказчика, например, называется `billingAddress`, а не `addressForBilling`, а поле `quantify` будет хранить целое число `5`, а не слово `five`.

Крайне важный, хотя и неудобный факт заключается в том, что когда мы пишем программу, получающую доступ к данным, она практически всегда подразумевает какую-то форму неявной схемы. За исключением случаев, когда мы пишем простой код наподобие

```
//псевдокод
foreach (Record r in records) {
  foreach (Field f in r.fields) {
    print (f.name, f.value)
  }
}
```

мы должны подразумевать определенные имена полей и некоторые осмысленные данные, а также делать предположения о типах данных, хранящихся в этих полях. Программа — не человек, она не может прочитать слово “qty” и сделать вывод, что оно означает “quantity”, — по крайней мере, если вы не запрограммировали ее на это. Итак, даже в неструктурированных базах данных обычно существует неявная схема — набор предположений о структуре данных в коде, манипулирующем этими данными.

Наличие неявной схемы в коде приложения порождает несколько проблем. Неявная схема означает, что, для того чтобы понять, какие данные хранятся в базе, вы должны заглянуть в код приложения. Если этот код хорошо структурирован, вы сразу найдете место, по которому можно определить схему. Но это ничем не гарантируется; все зависит от того, насколько ясным является код. Более того, база данных никак не отражает наличие схемы — она просто не может использовать схему, чтобы помочь вам выбрать способ хранения данных и эффективно извлекать их. Она не может предотвратить несогласованное манипулирование данными в разных приложениях.

Существует несколько причин, по которым реляционные базы данных имеют, а большинство баз данных в прошлом имели фиксированную схему. Схема имеет большое значение, а отрицание схем в базах данных NoSQL выглядит довольно пугающим.

По существу, неструктурированные базы данных переносят схему в код приложения, который к ней обращается. Это становится проблематичным, если к базе данных обращаются несколько приложений, разработанных разными людьми. Эти проблемы можно смягчить несколькими способами. Один из них — инкапсулировать все взаимодействия с базой данных в отдельном приложении и интегрировать его с другими приложениями через веб-сервисы.

Это соответствует современным предпочтениям многих людей, желающих обеспечить интеграцию с помощью веб-сервисов. Другой подход основан на четком разграничении разных областей агрегата, открытых для доступа разных приложений. Это могут быть разные разделы в документной базе данных или разные семейства столбцов в базе данных типа "семейство столбцов".

Несмотря на то что сторонники технологии NoSQL часто критикуют реляционные схемы за негибкость, эта критика не совсем справедлива. Реляционные схемы можно изменить в любой момент с помощью стандартных SQL-команд. По мере необходимости можно создать новые столбцы специальным образом, чтобы хранить в них неоднородные данные. Нам редко приходилось это делать, но это вполне возможно. Тем не менее неоднородность данных — хороший аргумент в пользу неструктурированных баз.

Неструктурированность оказывает большое влияние на изменения, происходящие со временем в структуре баз данных, особенно хранящих однородные данные. Несмотря на то что управление изменениями схемы реляционных баз данных практикуется реже, чем требовалось, это вполне возможно. Аналогично необходимо управлять изменениями, происходящими в способах хранения неструктурированных баз данных, чтобы пользователь мог легко извлечь как старые, так и новые данные. Более того, гибкость неструктурированных баз данных проявляется только внутри агрегата. Если вам требуется изменить границы агрегата, то перенос каждого бита окажется таким же сложным, как и в реляционных базах. Вопросы миграции баз данных мы обсудим позднее (глава 12, "Миграция схем").

3.4. Материализованные представления

Когда мы говорили об агрегатно-ориентированных моделях, мы подчеркивали их преимущества. Если вы хотите получить доступ к заказам, полезно собрать все данные для заказа в одном агрегате, который может храниться и предоставлять доступ как отдельная единица. Однако агрегатная ориентация имеет и недостаток: что произойдет, если товаровед захочет узнать, как часто конкретный товар заказывался на протяжении последних недель? В этом случае агрегатная ориентация мешает, вынуждая вас прочитывать каждый заказ в базе данных, чтобы получить ответ на вопрос. Вы можете облегчить задачу, построив индекс товаров, но все равно агрегатная структура останется неудобной.

Преимущество реляционных баз данных заключается в том, что отсутствие у них агрегатной структуры обеспечивает разнообразный доступ к данным. Более того, они обеспечивают удобный механизм, позволяющий искать данные не так, как они хранятся. Этот механизм называется *представлением*. Представление напоминает реляционную таблицу (оно является отношением), но определяется путем вычислений по базовым таблицам. Когда вы обращаетесь к представлению, база данных вычисляет данные в этом представлении — это удобная форма инкапсуляции.

Представления обеспечивают механизм сокрытия от клиента источника данных. Клиент не знает, были ли данные вычислены или просто взяты из базы данных. Однако вычисление некоторых представлений является затратным. Для решения этой проблемы были изобретены *материализованные представления* (materialized views), т.е. представления, вычисленные заранее и записанные в кеш-память диска. Материализованные представления эффективны при работе с часто считываемыми данными, но могут оказаться устаревшими

Несмотря на то что в базах данных NoSQL нет представлений, они могут иметь заранее вычисленные и кешированные запросы, к которым также применяется термин "материализованное представление". Этот аспект имеет гораздо большее значение для агрегатно-ориентированных баз данных, чем для реляционных баз, поскольку большинство приложений работают с запросами, которые плохо согласуются с агрегатной структурой. (Часто базы данных NoSQL создают материализованные представления с помощью подхода "отображение–свертка" (map-reduce), о котором мы поговорим в главе 7.)

Существуют две основные стратегии создания материализованного представления. Первая стратегия называется интенсивным обновлением, в рамках которой материализованное представление обновляется одновременно с обновлением соответствующих ему данных. В этом случае добавление заказа будет сопровождаться обновлением агрегатов истории покупок для каждого товара. Этот подход хорош, когда материализованное представление читается чаще, чем записывается, и должно быть актуальным. Здесь удобно применить подход, основанный на использовании баз данных приложения, поскольку он облегчает синхронную модификацию данных в базе и материализованном представлении.

Если вы не хотите мириться с дополнительными затратами при каждом обновлении базы данных, то можете через регулярные промежутки времени выполнять пакет заданий, связанных с обновлением материализованных представлений. Вы должны правильно задать допустимый уровень устаревания материализованных представлений, исходя из конкретных условий.

Материализованные представления можно создать за пределами базы данных, считывая данные, вычисляя представление и возвращая его в базу данных. Обычно базы данных самостоятельно создают материализованные представления. В этом случае вы должны задать необходимые вычисления, а база данных выполнит их, когда потребуется, в соответствии с заданными параметрами конфигурации. Это особенно удобно при интенсивном обновлении представлений в рамках постепенного отображения–свертки (подробнее об этом — в разделе 7.3.2, "Постепенное отображение–свертка").

Материализованные представления можно использовать в одном и том же агрегате. Документ заказа может содержать элемент, представляющий собой резюме заказа. Запрос к этому резюме не требует обращений ко всему документу заказа. В базах данных типа "семейство столбцов" для создания материализованных представлений используются разные семейства столбцов. Это позволяет обновлять материализованное представление в ходе одной и той же атомарной операции.

3.5. Моделирование доступа к данным

Как указывалось ранее, при моделировании агрегатов данных необходимо анализировать, как будут считываться данные, а также учитывать побочные эффекты, связанные с использованием агрегатов.

Начнем с модели, в которой все данные о клиенте хранятся в хранилище типа "ключ–значение" (рис. 3.2).

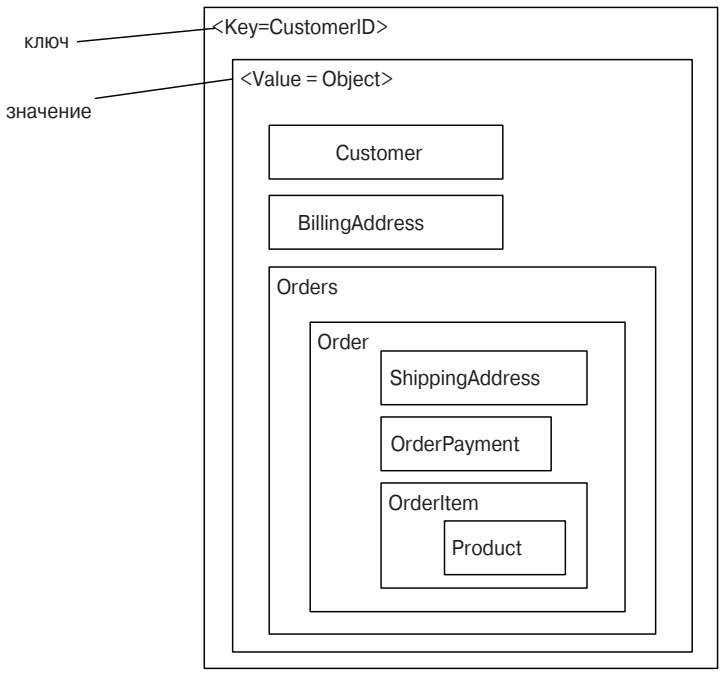

Рис. 3.2. Вложение всех объектов, относящихся к клиенту и его заказам

В этом сценарии приложение может считывать информацию о клиенте и все связанные с ним данные по ключу. Если необходимо считать заказы или товары, проданные по каждому заказу, считывается весь объект, а затем в результате анализа его клиентской части формируются результаты. Если нужны ссылки, можно перейти на документные хранилища, а затем направить запрос в документы или даже изменить данные в хранилище "ключ–значение" и разделить значение между объектами `Customer` и `Order`, а затем обеспечить взаимные ссылки этих объектов.

При работе со ссылками (рис. 3.3) заказы можно искать независимо от объекта `Customer`, а по ссылке `orderId` в объекте `Customer` можно найти все заказы клиента. Использование агрегатов позволяет оптимизировать чтение, но для этого необходимо поместить ссылку `orderId` в объект `Customer` для каждого нового объекта `Order`.

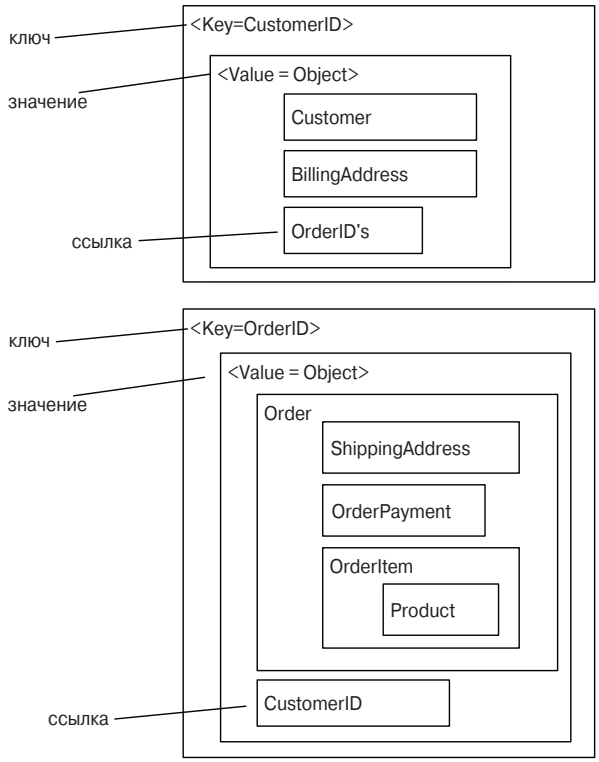

Рис. 3.3. Объект Customer хранится отдельно от объекта Order

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

# Order object
{
"customerId": 1,
"orderId": 99,
"order":{
  "orderDate":"Nov-20-2011",
  "orderItems":[{"productId":27, "price": 32.45}],
  "orderPayment":[{"ccinfo":"1000-1000-1000-1000",
     "txnId":"abelif879rft"}],
   "shippingAddress":{"city":"Chicago"}
  }
}
```

Агрегаты можно также использовать для получения аналитических данных; например, одновременно с обновлением агрегата можно заполнять информацию о том, какие объекты `Order` содержат заданный объект `Product`. Такая денормализация данных обеспечивает быстрый доступ к данным и лежит в основе процессов **Real Time BI** или **Real Time Analytics**, в которых предприятия не обязаны в конце каждого дня выполнять процедуры заполнения таблиц в хранилищах данных и генерировать аналитические отчеты; теперь они могут заполнять отчеты в момент размещения заказа клиентом.

```
{
"itemid":27,
"orders":{99,545,897,678}
}
{
"itemid":29,
"orders":{199,545,704,819}
}
```

В документных хранилищах запросы можно адресовать в документах, поэтому можно удалить ссылки на объекты `Order` из объекта `Customer`. Это позволяет не обновлять объект `Customer`, когда клиенты размещают новые заказы.

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

# Объект Order
{
"orderId": 99,
"customerId": 1,
"orderDate":"Nov-20-2011",
"orderItems":[{"productId":27, "price": 32.45}],
"orderPayment":[{"ccinfo":"1000-1000-1000-1000",
   "txnId":"abelif879rft"}],
"shippingAddress":{"city":"Chicago"}
}
```

Поскольку документные хранилища данных позволяют посылать запросы к атрибутам в документе, становятся возможными запросы наподобие “найти все заказы, содержащие товар *Рефакторинг баз данных*”, но решение о создании агрегата товаров и заказов следует принимать не из-за возможности таких запросов, а из-за возможности оптимизации чтения базы данных приложением.

При моделировании хранилищ типа "ключ–значение" мы получаем выгоду от упорядоченности столбцов. Это позволяет называть столбцы так, чтобы часто используемые столбцы считывались первыми. При использовании семейств столбцов для моделирования данных важно помнить, что это необходимо для оптимизации запросов, а не записи; общее правило таково: следует облегчить процедуру запроса и денормализовать данные при записи.

Существует много способов моделирования данных; можно хранить объекты `Customer` и `Order` в разных семействах, состоящих из *семейств столбцов* (рис. 3.4). Обратите внимание на то, что ссылка на все заказы, размещенные клиентом, хранится в семействе столбцов `Customer`. Аналогичные процедуры денормализации позволяют повысить эффективность выполнения запроса (чтения).

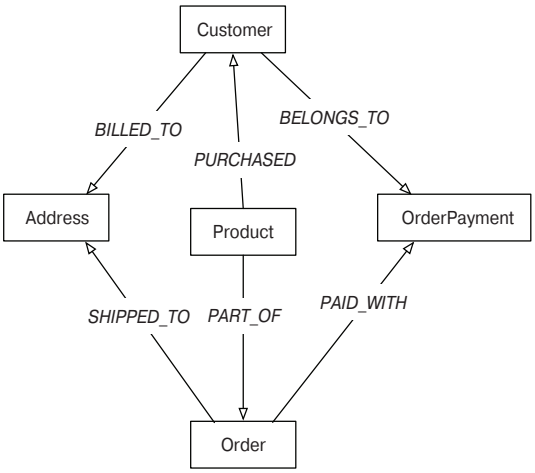

Рис. 3.4. Концептуальное представление о столбцовом хранении данных

При использовании графовой модели для моделирования тех же самых данных мы моделируем все объекты как узлы, а отношения между ними — как связи; эти связи имеют тип на направленность.

Каждый узел имеет независимые связи с другими узлами. Данные связи называются *`PURCHASED`*, *`PAID_WITH`* или *`BELONGS_TO`* (рис. 3.5); эти имена связей позволяют обойти граф. Допустим, вы хотите найти всех клиентов (`Customer`), купивших (*`PURCHASED`*) товар *Refactoring Database*. Для этого достаточно направить запрос к узлу товаров `Refactoring Databases` и найти все объекты `Customer` с входящей связью *`PURCHASED`*.

В графовых базовых данных этот тип обхода связей очень легко выполнить. Это особенно удобно, если данные нужны для рекомендации товаров пользователям или определения их покупательских предпочтений.

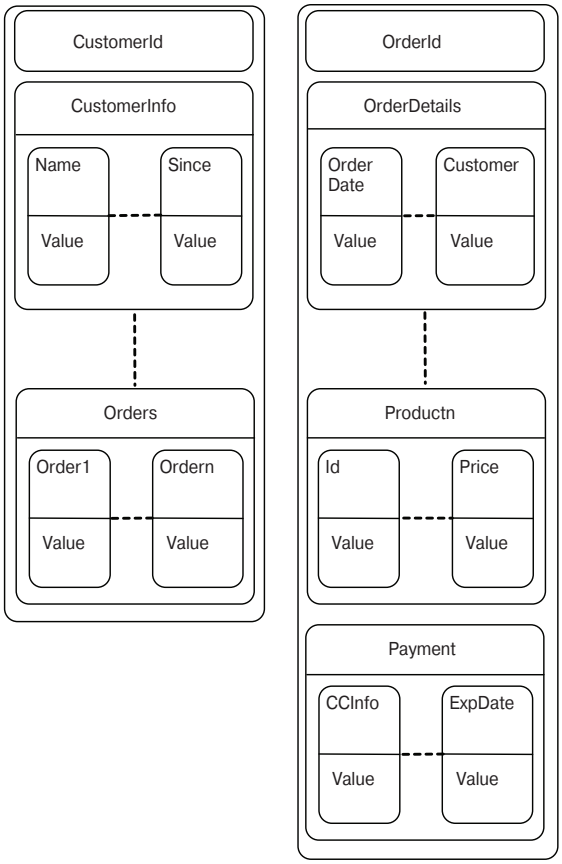

Рис. 3.5. Графовая модель данных для электронной торговли

3.6. Резюме

- Агрегатно-ориентированные базы данных создают межагрегатные связи, которые труднее обрабатывать, чем внутриагрегатные.
- Графовые базы данных организуют данные в виде графа, состоящего из узлов и ребер; они лучше всего работают с данными, имеющими сложную структуру связей.
- Неструктурированные базы данных позволяют легко добавлять поля в записи, но обычно существует неявная схема, подразумеваемая пользователями этих данных.
- Агрегатно-ориентированные базы данных часто вычисляют материализованные представления, чтобы представить пользователям данные, организованные не так, как в их исходных агрегатах. Для этого часто используются вычисления “отображения–свертка”.