ЧАСТЬ I

Антипаттерны логического проектирования баз данных

Прежде чем браться за написание кода, следует решить, какая информация должна храниться в базе данных, и выбрать лучший способ упорядочить и связать эти данные. В частности, на этом этапе проводится планирование таблиц баз данных, столбцов и отношений между ними.

Один инженер из Netscape, имя которого мы называть не будем, однажды передал указатель в JavaScript, сохранил его в строке, а потом передал обратно в C. Было много жертв.

➤ *Блейк Росс (Blake Ross)*

ГЛАВА 2

Кривая дорожка

Вы работаете над приложением для отслеживания ошибок и разрабатываете функциональность, которая позволяет назначить пользователя основным контактным лицом для продукта. Исходная архитектура позволяла выбрать только одного пользователя в качестве контакта. Впрочем, как это часто бывает, потом от вас потребовали, чтобы контактными лицами можно было назначить сразу нескольких пользователей.

В тот момент решение казалось простым: изменить базу данных, чтобы вместо одного идентификатора, как в предыдущей версии, в ней хранился список идентификаторов учетных записей пользователей, разделенных запятыми.

Но вскоре вас вызывает начальник и сообщает о возникшей проблеме. «Техотдел добавляет в проекты ассистентов. Они говорят, что могут добавить только пять человек. Когда пытаются добавить больше, происходит ошибка. В чем дело?»

«Да, в проект можно добавить ограниченное число контактов» — киваете вы, словно это совершенно обычное дело.

Похоже, шефу требуется более подробное объяснение. «Ну, от пяти до десяти, может, чуть больше. Зависит от того, когда были созданы их учетные записи». Шеф удивленно поднимает брови. Вы продолжаете: «Я храню идентификаторы учетных записей для проекта в списке, разделенном запятыми. Список идентификаторов должен помещаться в строке максимальной длины. Если идентификаторы учетных записей короткие, в списке их поместится больше. Учетным записям, созданным первыми, назначаются идентификаторы 99 и менее, и они занимают меньше места».

Шеф хмурится. Похоже, вам придется поработать по вечерам.

Разработчики часто используют списки, разделенные запятыми, чтобы не создавать таблицы пересечений для отношений «многие ко многим». Такой антипаттерн называется «Кривая дорожка» (jaywalking)[1, 2].

Цель: хранение многозначных атрибутов

Если столбец таблицы может содержать только одно значение, ее структура получается простой: достаточно выбрать тип данных SQL для представления одного экземпляра этого значения (например, целого числа, даты или строки). Не совсем понятно, как сохранить в столбце коллекцию связанных значений.

В примере с БД для отслеживания ошибок продукт связывается с контактом через целочисленный столбец таблицы `Products`. С каждой учетной записью может быть связано несколько продуктов, и каждый продукт хранит ссылку на один контакт, так что продукты связаны с учетными записями отношением «многие ко многим».

Jaywalking/obj/create.sql

```
CREATE TABLE Products (
  product_id SERIAL PRIMARY KEY,
  product_name VARCHAR(1000),
  account_id BIGINT UNSIGNED,
  -- . . .
  FOREIGN KEY (account_id) REFERENCES Accounts(account_id)
);

INSERT INTO Products (product_id, product_name, account_id)
VALUES (DEFAULT, 'Visual TurboBuilder', 12);
```

В процессе развития продукта вы понимаете, что контактов для него может быть несколько. Кроме отношения «многие к одному», необходимо поддерживать отношение «один ко многим» от продуктов к учетным записям. Одна строка данных таблицы `Products` должна быть способна хранить сразу несколько контактов.

Антипаттерн: форматирование списка, разделенного запятыми

Чтобы свести к минимуму изменения в структуре базы данных, вы решаете переопределить столбец `account_id` с типом `VARCHAR`, чтобы в нем можно

---

[1] Jaywalking — в частности, это переход улицы в неположенном месте. Дальше у автора игра слов: jaywalking — это избегание перекрестка (intersection), поэтому так назвали паттерн, родившийся из-за избегания пересечений (intersections). — *Примеч. пер.*

[2] Этот антипаттерн часто называют просто Multi-Valued Attribute. — *Примеч. науч. ред.*

было хранить несколько идентификаторов учетных записей, разделенных запятыми.

Jaywalking/anti/create.sql

```
  CREATE TABLE Products (
  product_id SERIAL PRIMARY KEY,
  product_name VARCHAR(1000),
  account_id VARCHAR(100), -- список, разделенный запятыми
-- . . .
);

INSERT INTO Products (product_id, product_name, account_id)
VALUES (DEFAULT, 'Visual TurboBuilder', '12,34');
```

Идея кажется удачной, потому что вам не пришлось создавать дополнительные таблицы или столбцы; вы изменили тип данных всего одного столбца. Но давайте присмотримся к проблемам производительности и целостности данных, свойственным такой структуре таблицы.

Запрос продуктов для конкретной учетной записи

Объединение всех внешних ключей в одном поле усложняет запросы. Теперь невозможно использовать равенство; вместо него приходится применять поиск по шаблону. Например, запрос для получения всех продуктов для учетной записи 12 в MySQL будет выглядеть примерно так:

Jaywalking/anti/regexp.sql

```
SELECT * FROM Products WHERE account_id REGEXP '\\b12\\b';
```

Выражения с поиском по шаблону могут возвращать ложные совпадения. Производительность ухудшается, потому что при поиске совпадения невозможно извлечь пользу из индексов. Так как в разных базах данных используются разные варианты синтаксиса поиска по шаблону, код SQL будет привязан к конкретному продукту.

Запрос учетных записей для конкретного продукта

Кроме того, соединение списка, разделенного запятыми, с соответствующими строками таблицы выполняется слишком медленно и неудобно.

Jaywalking/anti/regexp.sql

```
SELECT * FROM Products AS p JOIN Accounts AS a
    ON p.account_id REGEXP '\\b' || a.account_id || '\\b'
WHERE p.product_id = 123;
```

Соединение двух таблиц в таких выражениях делает использование индексов полностью невозможным, так что производительность снова страдает. Запрос должен просканировать обе таблицы, сгенерировать перекрестное произведение и применить регулярное выражение к каждой комбинации строк данных.

Создание агрегатных запросов

Агрегатные запросы используют такие функции, как `COUNT()`, `SUM()` и `AVG()`. Однако эти функции проектировались для использования с группами строк, а не со списками, разделенными запятыми. Приходится идти на разные ухищрения, например, вычислять длину строки со значениями, разделенными запятыми, за вычетом длины строки с удаленными запятыми. Результат может использоваться для подсчета элементов в списке.

Jaywalking/anti/count.sql

```
SELECT product_id,
    LENGTH(account_id) - LENGTH(REPLACE(account_id, ',', '')) + 1
      AS contacts_per_product
FROM Products;
```

Такие трюки впечатляют, но они неочевидны. На разработку таких решений уходит много времени, а отлаживать их трудно. Некоторые агрегатные запросы вообще невозможно реализовать, как ни старайся.

Обновление учетных записей для конкретного продукта

Новый идентификатор можно добавить в конец списка с помощью конкатенации строк, но это может привести к нарушению сортировки списка.

Jaywalking/anti/update.sql

```
UPDATE Products
SET account_id = account_id || ',' || 56
WHERE product_id = 123;
```

Чтобы удалить элемент из списка, необходимо выполнить два запроса SQL: на получение старого списка и на сохранение нового.

Jaywalking/anti/remove.py

```
import mysql.connector

cnx = mysql.connector.connect(user='scott', database='test')
cursor = cnx.cursor()

product_id_to_search = 2
value_to_remove = '34'
```

```
query = "SELECT product_id, account_id FROM Products WHERE product_id = %s"
cursor.execute(query, (product_id_to_search,))
for (row) in cursor:
    (product_id, account_ids) = row
    account_id_list = account_ids.split(",")
    account_id_list.remove(value_to_remove)
    account_ids = ",".join(account_id_list)
    query = "UPDATE Products SET account_id = %s WHERE product_id = %s"
    cursor.execute(query, (account_ids, product_id,))

cnx.commit()
```

Слишком много кода, чтобы просто удалить элемент из списка.

Проверка идентификаторов продуктов

Что мешает пользователю ввести недопустимый идентификатор, например `banana`?

Jaywalking/anti/banana.sql

```
INSERT INTO Products (product_id, product_name, account_id)
VALUES (DEFAULT, 'Visual TurboBuilder', '12,34,banana');
```

Пользователь рано или поздно введет что-нибудь не то, и ваша база данных превратится в тыкву. Возможно, обойдется без ошибок, но полученная информация не будет иметь никакого смысла.

Даже если значения являются целыми числами, нельзя быть уверенными, что эти целые числа присутствуют в таблице `Accounts`. Стандартный способ проверки основан на использовании ограничения внешнего ключа, но внешние ключи могут проверять только целые столбцы, а не отдельные элементы списка.

Выбор символа-разделителя

Если вы храните список строковых значений вместо целых чисел, некоторые элементы списка могут содержать символ-разделитель. Использование запятой в качестве разделителя между элементами может быть нежелательно. Можно выбрать другой символ, но нельзя гарантировать, что он больше нигде не встретится.

Ограничения длины списка

Сколько элементов списка можно сохранить в столбце `VARCHAR(30)`? Зависит от длины каждого из них. Если все элементы имеют длину в два символа, получится сохранить десять элементов (с учетом запятых). Если каждый элемент имеет длину в шесть символов, поместятся только четыре элемента:

Jaywalking/anti/length.sql

```
UPDATE Products SET account_id = '10,14,18,22,26,30,34,38,42,46'
WHERE product_id = 123;

UPDATE Products SET account_id = '101418,222630,343842,467790'
WHERE product_id = 123;
```

Как гарантировать, что `VARCHAR(30)` поддерживает самый длинный список, который вам может понадобиться в будущем? Какой длины будет достаточно? Попробуйте-ка объяснить причины ограничений длины шефу или заказчику.

Как распознать антипаттерн

Если вы слышите из уст коллег подобные фразы, это может указывать на применение антипаттерна «Кривая дорожка»/Multi-valued Attribute:

- «Сколько элементов максимум должно быть в этом списке?»

  Этот вопрос часто возникает при выборе максимальной длины столбца `VARCHAR`.

- «А ты знаешь, как найти границу слова в SQL?»

  Если вы используете регулярные выражения для выделения частей строки, возможно, эти части следует хранить по отдельности.

- «Какого символа нет ни в одном элементе списка?»

  Вы ищете символ-разделитель, который не создаст неоднозначности, но любой символ когда-нибудь может встретиться в элементе списка.

Допустимые применения антипаттерна

Можно повысить производительность некоторых запросов, применяя *денормализацию* к структуре базы данных. Хранение списков в виде строк, разделенных запятыми, является примером денормализации.

Приложению могут требоваться данные, разделенные запятыми, но при этом ему не нужно обращаться к отдельным элементам списка. Аналогичным образом, если приложение получает данные, разделенные запятыми, из другого источника и вы просто хотите сохранить весь список в базе данных и позднее прочитать его точно в таком же виде, разделять значения не обязательно.

Если вы решите применить денормализацию, действуйте консервативно. Начните с использования нормализованной структуры базы данных, потому что с ней код приложения может быть более гибким и она помогает сохранить целостность данных в базе.

Некоторые продукты баз данных SQL расширяют типы данных SQL разновидностями типа массива. Поддержка массивов предусмотрена в PostgreSQL, Oracle,

IBM DB2 и Informix. В зависимости от реализации они помогают избежать некоторых проблем, описанных ранее в этой главе. Например, можно задать скалярный тип данных для элементов массива. Тем не менее они не решат всех проблем. Их сложно использовать, и вам придется учиться работать с каждым диалектом SQL по отдельности.

Решение: создание таблицы пересечений

Вместо того чтобы хранить `account_id` в таблице `Products`, храните значение в отдельной таблице, чтобы каждое значение атрибута занимало отдельную строку. Новая таблица `Contacts` реализует отношение «многие ко многим» между `Products` и `Accounts`:

Jaywalking/soln/create.sql

```
CREATE TABLE Contacts (
  product_id BIGINT UNSIGNED NOT NULL,
  account_id BIGINT UNSIGNED NOT NULL,
  PRIMARY KEY (product_id, account_id),
  FOREIGN KEY (product_id) REFERENCES Products(product_id),
  FOREIGN KEY (account_id) REFERENCES Accounts(account_id)
);

INSERT INTO Contacts (product_id, account_id)
VALUES (123, 12), (123, 34), (345, 23), (567, 12), (567, 34);
```

Таблица с внешними ключами, относящимися к двум таблицам, называется *таблицей пересечений* (intersection table). Иногда эта таблица называется «таблицей соединения», «таблицей “многие к многим”», «таблицей отображения» или как-то еще. Название роли не играет; принцип остается тем же. Эта таблица реализует отношение «многие ко многим» между двумя связываемыми таблицами. Это означает, что каждый продукт может быть связан через таблицу пересечений с несколькими учетными записями, а каждая учетная запись может быть связана с несколькими продуктами. Диаграмма «объект — отношение» выглядит так: