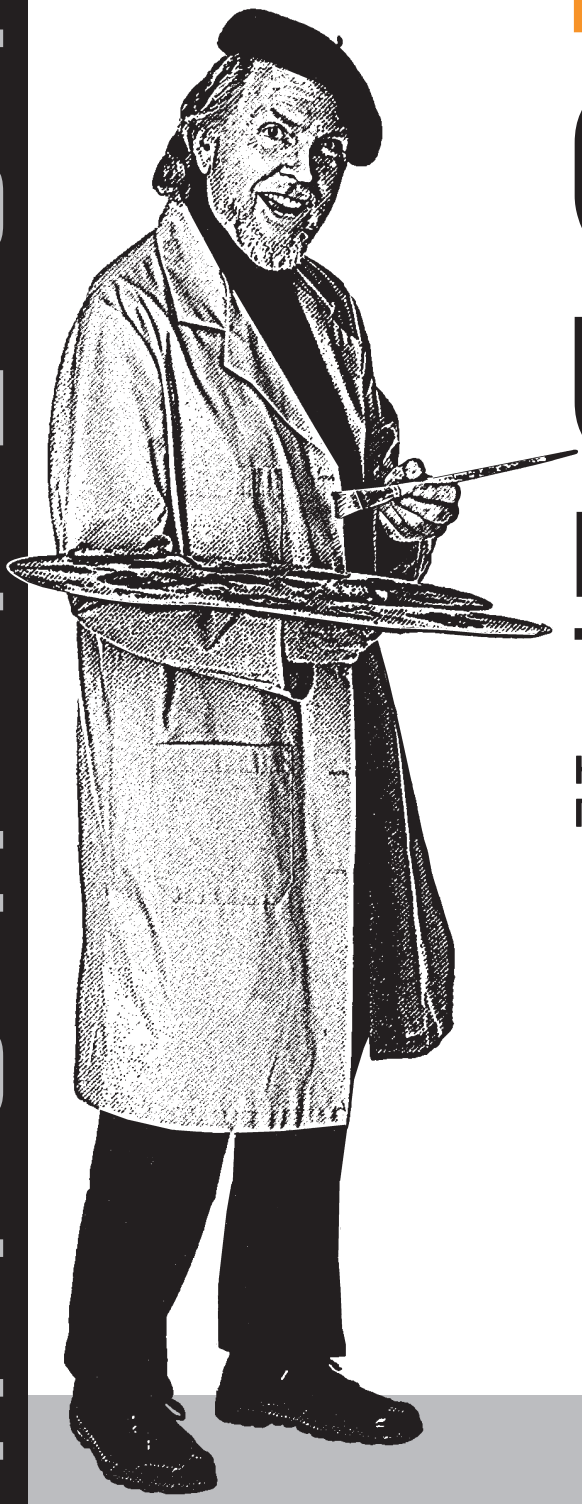
HIGH TECH

К. ДЖ. ДЕЙТ



SQL и реляционная теория

КАК ГРАМОТНО ПИСАТЬ КОД НА SQL

СИМВОЛ®
O'REILLY®

SQL and Relational Theory

How to Write Accurate SQL Code

C. J. Date

O'REILLY®

SQL и реляционная теория

Как грамотно писать код на SQL

К. Дж. Дейт

Санкт-Петербург — Москва
2010

К. Дж. Дейт

SQL и реляционная теория
Как грамотно писать код на SQL

Перевод А. Слинкина

Главный редактор	*А. Галунов*
Зав. редакцией	*Н. Макарова*
Выпускающий редактор	*П. Щеголев*
Редактор	*Ю. Бочина*
Корректор	*С. Минин*
Верстка	*К. Чубаров*

К. Дж. Дейт

SQL и реляционная теория. Как грамотно писать код на SQL. – Пер. с англ. – СПб.: Символ-Плюс, 2010. – 480 с., ил.

ISBN 978-5-93286-173-8

Язык SQL распространен повсеместно. Но работать с ним непросто: он сложен, запутан, при написании SQL-команд легко допустить ошибку. Понимание теории, лежащей в основе SQL, – лучший способ гарантировать, что ваш код будет написан правильно, а сама база данных надежна и легко сопровождаема.

В предлагаемой книге К. Дж. Дейт – признанный эксперт, начавший заниматься этими вопросами еще в 1970 году, – демонстрирует, как применить реляционную теорию к повседневной практике работы с SQL. Автор подробно объясняет различные аспекты этой модели, рассуждает и доказывает, приводит многочисленные примеры использования этого языка в соответствии с реляционной теорией.

Не будучи привязанным ни к какому конкретному продукту, издание опирается на многолетний опыт исследований и представляет наиболее актуальное на сегодняшний день изложение материала. Всякий, имеющий хоть какой-то опыт использования SQL – от скромного до весьма обширного, – получит от прочтения этой книги немалую пользу и удовольствие.

ISBN 978-5-93286-173-8
ISBN 978-0-596-52306-0 (англ)

Издательство «Символ-Плюс». 199034, Санкт-Петербург, 16 линия, 7,
тел. (812) 324-5353, www.symbol.ru. Лицензия ЛП N 000054 от 25.12.98.
Налоговая льгота – общероссийский классификатор продукции
ОК 005-93, том 2; 953000 – книги и брошюры.

Подписано в печать 17.02.2010. Формат 70×100 $^{1}/_{16}$. Печать офсетная.
Объем 30 печ. л. Тираж 1200 экз. Заказ №
Отпечатано с готовых диапозитивов в ГУП «Типография «Наука»
199034, Санкт-Петербург, 9 линия, 12.

Всем считающим,
что подобное упражнение
стоит затраченного времени,
а в особенности Лексу де Хаану,
которого так недостает.

Всякий, кто полагается на практику,
не зная теории, подобен кормчему,
вступающему на судно без руля и компаса, –
он не знает, куда плывет.
Практика всегда должна опираться
на твердые теоретические основания.

Леонардо да Винчи (1452–1519)

Не в том беда, что мы чего-то не знаем,
а в том, что наши знания кажущиеся.

Джош Биллингс (1818–1885)

Языки умирают…
Математические идеи – нет.

Дж. Х. Харди

К сожалению, разрыв между теорией
и практикой в теории не настолько широк,
как на практике.

Аноним

Оглавление

Предисловие

Язык SQL распространен повсеместно. Но работать с SQL трудно: он сложен, запутан, при написании SQL-команд легко допустить ошибку – рискну предположить, что куда легче, чем можно судить по уверениям апологетов этого языка. Поэтому, чтобы уверенно писать правильный SQL-код (то есть такой, который делает в точности то, что вам нужно, не больше и не меньше), необходимо четко следовать некоторому правилу. А основной тезис настоящей книги заключается в том, что таким правилом может стать *использование SQL в соответствии с реляционной теорией*. Но что это означает? Разве SQL не является изначально реляционным языком?

Да, конечно, SQL – стандартный язык для всех реляционных баз данных, но сам по себе этот факт не делает его реляционным. Как это ни печально, SQL слишком часто отходит от принципов реляционной теории; строки-дубликаты и null-значения – два наиболее очевидных примера, которыми проблема отнюдь не ограничивается. Следовательно, он предлагает вам путь, который может привести в западню. А если вы не желаете попадать в западню, то должны понимать реляционную теорию (что она собой представляет и для чего предназначена), знать, в чем именно SQL отклоняется от этой теории, и уметь избегать проблем, которые могут из этого проистекать. Одним словом, SQL следует использовать в реляционном духе. Тогда вы сможете действовать так, будто SQL на самом деле реляционный язык, и получать все преимущества от работы с тем, что по сути является истинно реляционной системой.

В подобной книге не возникло бы необходимости, если бы все и без того использовали SQL реляционным образом, – но, увы, это не так. Напротив, в современном применении SQL я вижу проявление множества вредных тенденций. Более того, эти способы применения относятся к рекомендуемым в различных учебниках и иных публикациях авторами, которые должны были бы относиться к своей работе более ответственно (имен не привожу и на посмешище выставлять не хочу); анализ литературы в этом отношении оставляет удручающее впечатление. Реляционная модель появилась на свет в 1969 году, и вот – почти сорок лет спустя – она, похоже, так и не понята по-настоящему сообществом пользователей СУБД в целом. Отчасти поэтому в основу организации настоящей книги положена сама реляционная модель; подробно объясняются различные аспекты этой модели, и для каждого аспекта демонстрируется, как лучше всего реализовать его средствами SQL.

Предварительные требования

Я предполагаю, что вы применяете СУБД в своей работе, поэтому уже в какой-то мере знакомы с языком SQL. Точнее, я считаю, что вы имеете практическое представление либо о стандарте SQL, либо (что более вероятно) о каком-либо продукте, где применяется SQL. Однако я не рассчитываю на глубокое знание реляционной теории как таковой (хотя надеюсь, что вы все же согласны с тем, что реляционная теория – вещь хорошая, и по возможности ее следует придерживаться). Поэтому во избежание недопонимания я буду подробно описывать различные свойства реляционной модели, а также показывать, как SQL согласуется с этими свойствами. Однако я не стану пытаться обосновывать существование этих свойств, а буду предполагать, что вы достаточно подкованы в тематике баз данных, чтобы понимать, к примеру, зачем используется понятие ключа, или почему иногда приходится выполнять операцию соединения, или зачем требуется поддержка отношений многие-ко-многим. (Если бы я решил включать все определения и обоснования, то получилась бы совсем другая книга – гораздо больше по размеру, не говоря уже обо всем остальном; да и в любом случае такая книга уже написана.)

Я сказал, что ожидаю от вас достаточно близкого знакомства с SQL. Добавлю, однако, что некоторые аспекты SQL я все равно буду объяснять подробно – особенно те, что на практике применяются нечасто. (В качестве примера упомяну «потенциально недетерминированные выражения». См. главу 12.)

«Database in Depth»

Эта книга базируется на ранее изданной книге «Database in Depth: Relational Theory for Practitioners» (O'Reilly, 2005) и должна заменить ее. В предыдущей книге я ставил перед собой такую цель (цитата взята из предисловия):

> Посвятив много лет работе с базами данных в разном качестве, я пришел к выводу, что существует настоятельная потребность в книге для практиков (не новичков), где принципы реляционной теории излагались бы вне связи с особенностями конкретных существующих продуктов, коммерческими обычаями или стандартом SQL. Таким образом, в качестве читательской аудитории я вижу опытных пользователей СУБД, достаточно честных, чтобы сознаться в том, что они не понимают теоретических основ той области, в которой работают, в той мере, в какой могли бы и должны были бы понимать. Этой теорией, естественно, является реляционная модель, и хотя фундаментальные идеи, положенные в ее основу, очень просты, надо признать, что они чуть ли не повсеместно неверно представляются, или недооцениваются,

> или то и другое вместе. Фактически часто их не понимают вовсе. Вот, к примеру, несколько вопросов по реляционной теории[1]... На сколько из них вы сможете ответить?
>
> Что в точности означает первая нормальная форма?
>
> Какая связь существует между отношениями и предикатами?
>
> Что такое семантическая оптимизация?
>
> Что такое отношение-образ?
>
> Почему так важна полуразность?
>
> Почему отложенная проверка ограничений целостности не имеет смысла?
>
> Что такое переменная-отношение?
>
> Что такое предваренная нормальная форма?
>
> Может ли отношение иметь атрибут, значениями которого являются отношения?
>
> Является ли язык SQL реляционно полным?
>
> Почему так важен *принцип информации*?
>
> Как XML укладывается в реляционную модель?
>
> Настоящая книга дает ответы на эти и многие другие вопросы. В общем и целом, ее цель – помочь пользователям-практикам баз данных глубоко разобраться в реляционной теории и применить полученные знания в повседневной профессиональной деятельности.

Как следует из последнего предложения, я надеялся, что читатели той книги смогут применить изложенные идеи в своей работе без дальнейшей помощи с моей стороны. Но с тех пор я осознал, что вопреки устоявшемуся мнению язык SQL настолько труден, что вопрос о том, как применять его, не нарушая реляционных принципов, далеко не праздный. Поэтому я решил дополнить первоначальную книгу, включив в нее явные, конкретные рекомендации именно в этом направлении (то есть как использовать SQL в реляционном духе). Таким образом, в этой книге я преследовал ту же цель, что и раньше (помочь пользователям-практикам баз данных глубоко разобраться в реляционной теории и применить полученные знания в повседневной профессиональной деятельности), но постарался представить материал в виде, быть может, более удобном для усвоения и уж точно – для применения. Иными словами, я включил много материала, относящегося конкретно к языку SQL (и именно поэтому размер так сильно увеличился по сравнению с предыдущим вариантом).

---

[1] По причинам, которые здесь несущественны, я заменил несколько вопросов из оригинального перечня.

Дополнительные замечания о тексте

Я должен высказать еще несколько предварительных замечаний. Прежде всего, мое собственное понимание реляционной модели с годами развивалось. В этой книге изложены мои нынешние представления о предмете; поэтому если вы обнаружите технические расхождения – а они есть – между этой и другими моими книгами (в частности, и той, которую эта призвана заменить), правильным следует считать написанное здесь. Впрочем, сразу оговорюсь, что расхождения по большей части не слишком существенны; более того, я всегда соотношу новые термины и понятия со старыми, если считаю, что в этом есть необходимость.

Во-вторых, я, как и обещано, собираюсь говорить о теории, но опираясь на свою уверенность, что *нет ничего практичнее хорошей теории*. Я специально выделил этот момент, потому что многие, похоже, придерживаются противоположного мнения и считают: все, что отдает теоретизированием, на практике неприменимо. Но истина в том, что теория (по крайней мере, реляционная, а именно о ней я и веду речь) очень даже приближена к практике. Она создавалась как теория *не* ради теории, а ради построения систем, на все сто процентов практических. Каждая деталь этой теории обусловлена весьма практическими соображениями. Один из рецензентов предыдущей книги, Стефан Фарульт (Stéphane Faroult), писал: «Приобретя некоторый практический опыт, вы начинаете осознавать, что без знания теории не обойтись». Более того, эта теория не только практична, она еще и фундаментальна, проста, понятна, полезна, а иногда еще и *забавна* (как я надеюсь продемонстрировать в этой книге).

Разумеется, за самой яркой иллюстрацией вышеприведенного тезиса не нужно ходить дальше самой реляционной модели. Вообще вряд ли необходимо отстаивать мнение о практичности теории в том контексте, который мы имеем: существование многомиллиардной индустрии, целиком основанной на одной замечательной теоретической идее. Однако я предвижу и такую циничную позицию: «Ну ладно, а что эта теория сделала лично для меня в последнее время?» Иными словами, те из нас, кто действительно убежден в важности теории, должны постоянно противостоять критикам – и это еще одна причина, по которой я считаю эту книгу полезной.

В-третьих, как я уже говорил, в этой книге детально излагаются различные аспекты SQL и реляционной модели. (Я сознательно почти не затрагивал темы, не имеющие отношения собственно к реляционной теории, в частности транзакции.) Я всюду старался ясно отмечать, когда обсуждение относится только к SQL, когда – только к реляционной модели, а когда – к тому и другому. Однако должен подчеркнуть, что не стремился к исчерпывающему рассмотрению всех деталей SQL. Язык SQL очень сложен и предоставляет много разных способов решения одной

и той же задачи, в нем так много исключений и особых случаев, что попытка охватить все – даже если бы это было возможно, в чем я лично сомневаюсь, – оказалась бы непродуктивной, а книга при этом выросла бы до необозримых размеров. Поэтому я старался уделить внимание тому, что считаю действительно важным, не растекаясь при этом мыслью по древу. Хотелось бы заявить, что если вы будете делать все, что я советую, и не будете делать того, что я не советую, то это станет первым приближением к безопасной работе: вы будете использовать SQL реляционно. Но только вам судить, насколько это заявление оправдано и оправдано ли вообще.

Ко всему сказанному следует добавить, что, к сожалению, существуют ситуации, когда SQL невозможно использовать реляционно. Например, проверку некоторых ограничений целостности приходится откладывать (обычно, чтобы выиграть время), хотя реляционная модель считает такую отложенную проверку логической ошибкой. В этой книге приводятся рекомендации о том, как поступать в подобных случаях, но боюсь, что по большей части они сводятся к тривиальному совету: *делайте так, как считаете наиболее правильным*. Надеюсь, по крайней мере, что вы будете понимать, в чем состоит опасность отклонения от модели.

Должен также сказать, что некоторые предлагаемые рекомендации не относятся к реляционной теории, а имеют общий характер, хотя иногда у них есть и «реляционные» последствия (не всегда очевидные, кстати говоря). Хорошим примером может служить совет *избегать приведения типов*.

В-четвертых, обратите внимание, что под словом *SQL* я в этой книге понимаю исключительно стандартную версию языка, а не какой-то конкретный диалект (если только противное не оговорено явно). В частности, в согласии со стандартом я подразумеваю произношение «эс кью эл», а не «сиквел»[1] (хотя на практике последнее широко распространено).

В-пятых, эту книгу следует читать в основном последовательно за немногими исключениями, оговоренными здесь и далее в самом тексте (большинство глав в той или иной мере зависят от ранее изложенного материала, поэтому старайтесь не перескакивать из одного места в другое). Кроме того, в каждую главу включены упражнения. Конечно, выполнять их необязательно, но мне кажется, что было бы разумно попробовать свои силы хотя бы на некоторых. Ответы, в которых часто содержится дополнительная информация по теме, приведены в приложении С.

И наконец, хотелось бы упомянуть о некоторых видеосеминарах, основанных на материалах этой книги. Дополнительную информацию см. по адресу *http://www.clik.to/chris_date* или *http://www.thethirdmanifesto.com*.

[1] Поэтому мы пишем «об SQL», а не «о SQL». – *Прим. перев.*

Типографские соглашения

В книге применяются следующие соглашения:

Курсив

Служит для визуального выделения. Этим шрифтом обозначаются новые термины. Кроме того, он применяется в основном тексте, когда нужно сказать, что вместо чего-то следует подставить некоторую переменную, например x.

`Моноширинный шрифт`

Применяется для примеров кода.

`Моноширинный курсив`

В примерах кода обозначает переменную или значение, вводимое пользователем.

О примерах кода

Эта книга призвана помогать вам в работе. Поэтому вы можете использовать приведенный в ней код в собственных программах и в документации. Спрашивать у нас разрешение необязательно, если только вы не собираетесь воспроизводить значительную часть кода. Например, не требуется разрешение, чтобы включить в свою программу несколько фрагментов кода из книги. Однако для продажи или распространения примеров на компакт-диске нужно получить разрешение. Можно без ограничений цитировать книгу и примеры в ответах на вопросы. Но чтобы включить значительные объемы кода в документацию по собственному продукту, нужно получить разрешение.

Мы высоко ценим, хотя и не требуем, ссылки на наши издания. В ссылке обычно указываются название книги, имя автора, издательство и ISBN, например: «SQL and Relational Theory, C. J. Date», Copyright 2009 C. J. Date, 978-0-596-52306-0.

Если вы полагаете, что планируемое использование кода выходит за рамки изложенной выше лицензии, пожалуйста, обратитесь к нам по адресу *permissions@oreilly.com*.

Замечания и вопросы

Я очень старался, чтобы в книге не было ошибок, но что-то мог упустить. Если вы найдете какой-то огрех, просьба уведомить издательство по адресу:

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

800-998-9938 (для США и Канады)
707-829-0515 (международный или местный)
707-829-0104 (факс)

Можно также посылать сообщения по электронной почте. Чтобы подписаться на рассылку или заказать каталог, отправьте письмо на адрес:

info@oreilly.com

Замечания и вопросы технического характера следует отправлять по адресу:

bookquestions@oreilly.com

Для этой книги создана веб-страница, на которой выкладываются примеры и списки замеченных ошибок (ранее отправленные извещения об ошибках и исправления доступны для всеобщего обозрения). Адрес страницы:

http://www.oreilly.com/catalog/9780596523060

Дополнительную информацию об этой и других книгах можно найти на сайте издательства O'Reilly:

http://www.oreilly.com

Доступность на Safari

Если на обложке вашей любимой книги присутствует значок Safari® Enabled, это означает, что книга доступна в сетевой библиотеке Safari издательства O'Reilly.

У Safari есть преимущество перед обычными электронными книгами. Это виртуальная библиотека, которая позволяет легко находить тысячи технических книг, копировать примеры программ, загружать отдельные главы и быстро получать точную и актуальную информацию. Бесплатный доступ по адресу *http://safari.oreilly.com*.

Благодарности

О том, чтобы переработать предыдущую книгу, включив в нее дополнительный материал по SQL, я подумывал довольно давно, но последним толчком, заставившим меня наконец засесть за эту работу, явилось мое присутствие в 2007 году на одном курсе, предназначенном для практических пользователей СУБД. Курс вел Тоон Коппелаарс (Toon Koppelaars) на базе книги, которую он написал совместно с Лексом де Хааном (Lex de Haan) (рецензентом этой книги), очень неплохой, кстати. Но поразительным оказалось то, насколько беспомощно выглядели попытки слушателей применить реляционные и логические принципы к привычному для них использованию SQL. Конечно, у слушателей были какие-то знания по этой теме – в конце концов, все они рабо-

тали с базами данных на практике, – но мне показалось, что их определенно нужно направить в сторону применения этих идей в повседневной деятельности. Поэтому я и написал эту книгу. И потому я благодарен в первую очередь Тоону и Лексу, которые придали импульс, необходимый для начала работы над проектом. Я благодарен также рецензентам Хербу Эдельштейну (Herb Edelstein), Шеери Ктитцеру (Sheeri Ktitzer), Энди Ораму (Andy Oram), Питеру Робсону (Peter Robson) и Бэрону Шварцу (Baron Schwartz) за замечания по первым черновикам рукописи и Хью Дарвену (Hugh Darwen) и Джиму Мелтону (Jim Melton) за техническую помощь иного характера. Кроме того, я как всегда благодарен своей жене Линди за неизменную многолетнюю поддержку этого и прочих моих проектов. Наконец, спасибо всему коллективу издательства O'Reilly и особенно Изабель Канкл (Isabel Kunkle), Энди Ораму (Andy Oram) и Адаму Уитверу (Adam Witwer) за ценные советы и за то, что они ободряли и поддерживали меня на протяжении всего времени работы над книгой.

К. Дж. Дейт

Хилдсбург, штат Калифорния,
2008

1

Введение

Реляционный подход к SQL – вот тема или одна из тем настоящей книги. Разумеется, чтобы должным образом раскрыть эту тему, мне придется рассматривать вопросы, касающиеся как реляционной теории, так и самого языка SQL. И хотя это замечание очевидным образом применимо ко всей книге, к первой главе оно относится в особенности. Поэтому здесь мы почти не будем говорить о языке SQL как таковом. Моя цель сейчас – дать обзор материала, большая часть которого вам, скорее всего, уже известна. Я хочу обозначить некую отправную точку, то есть заложить фундамент, на котором можно будет возводить здание книги. Но, даже искренне рассчитывая на то, что вы в основном знакомы с тем, о чем я собираюсь рассказать в этой главе, я все же предлагаю не пропускать ее. Вы должны знать то, что знать надлежит (надеюсь, вы понимаете, что я хочу сказать); в частности, вы должны владеть всем, что понадобится для понимания материала следующих глав. На самом деле я – со всем уважением – порекомендовал бы вообще не пропускать рассмотрение тем, которые вам кажутся знакомыми. Например, так ли вы уверены, что знаете, что в реляционной терминологии понимается под ключом? Или что такое соединение?[1]

---

[1] По крайней мере один ученый муж этого не понимает. Следующий отрывок взят из документа, который (как и эта книга!) имеет целью наставить пользователей SQL: «Не применяйте соединения... Подход Oracle и SQL Server к этой концепции принципиально различен... Вы можете получить неожиданные результирующие наборы... Необходимо понимать основные типы соединений... Экви-соединение строится путем выборки всех данных двух отдельных источников и создания из них одной большой таблицы... В случае внутреннего соединения две таблицы соединяются по внутренним столбцам... В случае внешнего соединения две таблицы соединяются по внешним столбцам... (*да-да, именно так написано в оригинале. – Прим. перев.*). В случае левостороннего соединения две таблицы соединяются по левым столбцам. В случае правостороннего соединения две таблицы соединяются по правым столбцам».

Реляционная модель очень плохо понята

Профессионалы в любой области должны знать лежащие в ее основе фундаментальные идеи. Поэтому профессионал в области баз данных должен знать реляционную теорию, поскольку именно реляционная модель является фундаментом (уж во всяком случае здоровенной частью фундамента), на котором покоится вся эта отрасль. В наше время любой курс по управлению базами данных, академический или коммерческий, на словах привержен идее преподавания реляционной модели, но в большинстве случаев преподавание поставлено очень плохо, если судить по результатам; безусловно, в сообществе пользователей баз данных эта модель недопонята. Перечислим некоторые возможные причины такого положения вещей.

- Модель преподносится в отрыве от всего, в «вакууме». Поэтому начинающему очень трудно уловить значимость материала или понять, какие проблемы модель призвана решить, или то и другое одновременно.
- Сами преподаватели не до конца понимают или не в состоянии оценить важность материала.
- Чаще всего на практике модель как таковая не рассматривается вовсе – вместо нее преподается язык SQL или какой-то его диалект, например принятый в Oracle.

Поэтому эта книга обращена к тем людям, которые на практике работают с базами данных и в особенности с языком SQL и каким-то боком связаны с реляционной моделью, но знают о ней не так много, как должны или хотели бы знать. Она определенно *не рассчитана* на начинающих, однако не является курсом повышения квалификации. Конкретно, я уверен, что вы что-то знаете о языке SQL, но – не сочтите за обиду – если ваши знания о реляционной модели проистекают исключительно из знания SQL, то, боюсь, вы не понимаете ее так хорошо, как следовало бы, и очень может статься, что какие-то «ваши знания неверны». Не устану повторять: *SQL и реляционная модель – вовсе не одно и то же*. В качестве иллюстрации приведу некоторые вопросы реляционной теории, которые в SQL трактуются недостаточно ясно (и это еще мягко сказано):

- Что в действительности представляют собой базы данных, отношения и кортежи
- В чем разница между отношениями-значениями и переменными-отношениями
- Значимость предикатов и высказываний
- Важность имен атрибутов
- Важнейшая роль ограничений целостности

и так далее (список далеко не полный). Все эти и многие другие вопросы рассматриваются в настоящей книге.

Еще раз повторю: если ваши знания о реляционной модели проистекают исключительно из знания SQL, то ваши знания могут оказаться неверными. Отсюда, в частности, следует, что, читая эту книгу, вы, возможно, обнаружите, что кое-что из уже известного следует забыть, а переучиваться, к сожалению, всегда трудно.

Некоторые замечания о терминологии

Вероятно, вы сразу же обратили внимание на то, что в перечне вопросов реляционной теории из предыдущего раздела я употребил формальные термины «отношение», «кортеж» и «атрибут». Но в SQL эти термины не используются, там применяются более «дружественные» слова: таблицы, строка и столбец. Вообще говоря, я с пониманием отношусь к идее употребления понятных пользователю слов, если это помогает усвоению идей. Но в данном случае мне кажется, что как раз усвоению идей такая терминология не способствует – как это ни печально; напротив, она лишь искажает суть и оказывает дурную услугу пониманию истинного смысла.

А истина заключается в том, что отношение – это не таблица, кортеж – не строка, а атрибут – не столбец. И хотя в неформальном контексте такое словоупотребление может показаться приемлемым – я и сам так часто говорю, – я настаиваю, что приемлемо оно лишь в том случае, когда мы ясно понимаем, что эти дружественные термины – не более чем приближение к истине, они совершенно не ухватывают смысл того, что происходит на самом деле. Скажу по-другому: если вы понимаете истинное положение вещей, то благоразумное употребление дружественных терминов может оказаться здравой идеей, но, чтобы понять и оценить это истинное положение, необходимо освоить более формальную терминологию. Поэтому в этой книге я буду, как правило, пользоваться формальными терминами – по крайней мере тогда, когда говорю о реляционной модели, противопоставляя ее SQL, – и в нужном месте приведу их строгие определения. Напротив, в контексте SQL я буду употреблять присущую ему терминологию.

И еще одно замечание о терминологии: упомянув, что SQL пытается упростить один набор терминов, я должен добавить, что он сделал все возможное для усложнения другого. Я имею в виду использование терминов *оператор*, *функция*, *процедура*, *подпрограмма* и *метод*; все они обозначают по существу одно и то же (быть может, с незначительными вариациями). В этой книге я всюду буду употреблять слово *оператор*.

Раз уж зашла речь об SQL, позвольте напомнить, что (как отмечалось в предисловии) под этим я понимаю исключительно стандартную вер-

сию языка[1], если не считать нескольких мест, где по контексту требуется иное.

- Иногда я употребляю терминологию, отличающуюся от стандартной. Например, я пишу *табличное выражение* (table expression) вместо принятого в стандарте *выражение запроса* (query expression), потому что (а) значением такого выражения является таблица, а не запрос, и (б) запросы – не единственный контекст, в котором такие выражения используются. (На самом деле, в стандарте применяется термин *табличное выражение*, но в гораздо более узком смысле; конкретно, он обозначает то, что следует за фразой SELECT в выражении SELECT.)
- Продолжая предыдущую мысль, должен добавить, что не все табличные выражения допустимы в SQL в любом контексте, где их использование можно было бы предположить. В частности, результат явной операции JOIN, хотя он, безусловно, обозначает таблицу, не может встречаться в качестве «отдельно стоящего» табличного выражения (то есть на самом внешнем уровне вложенности), а также не может выступать в качестве табличного выражения в скобках, которое составляет подзапрос (см. главу 12). *Обратите внимание, что подобные замечания применимы ко многим обсуждениям в основном тексте книги, но повторять их каждый раз было бы скучно, поэтому я этого делать не буду.* (Однако все это сведено в БНФ-грамматике в главе 12.)
- Я игнорирую те аспекты стандарта, которые можно считать чрезмерно эзотерическими, особенно если они не являются частью того, что в стандарте называется Core SQL (Базовый SQL), или имеют мало общего с реляционной обработкой как таковой. В качестве примеров упомяну так называемые аналитические или оконные функции (OLAP), динамический SQL, рекурсивные запросы, временные таблицы и детали типов, определенных пользователем.
- По причинам, отчасти связанным с типографским набором, я применяю для комментариев стиль, отличающийся от стандартного. Точнее, комментарии печатаются курсивом и обрамлены ограничителями «/*» и «*/».

Не забывайте, что все реализации SQL включают средства, не описанные в стандарте. Типичный пример – идентификаторы строк. Моя общая рекомендация относительно таких средств такова: пользуйтесь ради бога, но только если они не нарушают реляционных принципов (в конце концов, эта книга посвящена описанию *реляционного* подхода к SQL). Например, применение идентификаторов строк, скорее всего, нарушит так называемый *принцип взаимозаменяемости* (см. гла-

[1] International Organization for Standardization (ISO): *Database Language SQL*, Document ISO/IEC 9075:2003 (2003).

ву 9), и если это так, я бы точно не стал их использовать. Но и в этом, и во всех остальных случаях действует универсальное правило: можете делать все, что хотите, если только знаете, что делаете.

Принципы, а не продукты

Стоит потратить немного времени на вопрос о том, почему профессионалу в области баз данных необходимо (как я уже отмечал выше) знать реляционную модель. Причина в том, что реляционная модель не связана ни с каким конкретным продуктом; она имеет дело только с принципами. Что я понимаю под принципами? Словарь дает такие определения:

> **Принцип** – основное, исходное положение теории, учения, мировоззрения; убеждение, взгляд на вещи; основная особенность в устройстве чего-либо.[1]

Важнейшая особенность принципов состоит в их долговечности. Продукты и технологии (и язык SQL в том числе) все время изменяются – принципы остаются постоянными. Допустим, к примеру, что вы знаете СУБД Oracle; предположим даже, что вы крупный специалист по Oracle. Но если ваши знания ограничены только Oracle, то они могут оказаться непригодными, скажем, в среде DB2 или SQL Server (а могут даже затруднить освоение новой среды). Однако если вы знаете основополагающие принципы – иными словами, реляционную модель, – ваши знания и навыки *переносимы*: вы сможете применить их в любой среде, и они никогда не устареют.

Поэтому в этой книге мы будем иметь дело с принципами, а не с продуктами, с основами, а не с преходящими увлечениями. Но я понимаю, что в реальном мире иногда приходится идти на компромиссы. Например, иногда имеются веские причины проектировать базу данных способом, далеким от теоретически оптимального. В качестве другого примера снова обратимся к SQL. Хотя нет сомнений, что SQL можно использовать реляционно (по крайней мере, в большинстве случаев), иногда обнаруживается – ведь существующие реализации так далеки от совершенства, – что это влечет за собой существенное падение производительности... и тогда вы более или менее вынуждены делать что-то не «истинно реляционное» (например, писать запрос неестественным образом, чтобы заставить реализацию воспользоваться индексом). Однако я абсолютно убежден, что такие компромиссы всегда следует оценивать с *концептуальных позиций*. Это означает, что:

- Идя на такой компромисс, вы должны понимать, что делаете.
- Вы должны понимать, как выглядит теоретически правильное решение, и иметь основательные причины для отхода от него.

---

[1] Перевод дан по изданию Толкового словаря русского языка С. И. Ожегова. – *Прим. перев.*

- Вы должны документировать эти причины; тогда в случае, если в будущем они отпадут (например, из-за того, что в новой версии продукта какая-то функция реализована более удачно), можно будет отказаться от первоначального компромисса.

Следующий афоризм – который приписывается Леонардо да Винчи (1452–1519), и, стало быть, ему уже около 500 лет – великолепно описывает эту ситуацию:

> Всякий, кто полагается на практику, не зная теории, подобен кормчему, вступающему на судно без руля и компаса, – он не знает, куда плывет. *Практика всегда должна опираться на твердые теоретические основания.*

(Курсив добавлен мной.)

Обзор оригинальной модели

В этом разделе я хочу задать отправную точку для последующего обсуждения; здесь приводится обзор некоторых базовых аспектов реляционной модели в том виде, в котором она была определена изначально. Обратите внимание на уточнение – «была определена изначально»! Бытует широко распространенное заблуждение, будто реляционная модель – вещь абсолютно неизменная. Это не так. В этом отношении она напоминает математику: математика тоже не статична, а изменяется со временем. Да ведь и сама реляционная модель – это тоже отрасль математики и как таковая эволюционирует по мере доказательства новых теорем и получения новых результатов. Более того, новый вклад может быть внесен любым компетентным специалистом. И, подобно математике, реляционная модель, хотя и была первоначально изобретена одним человеком, ныне стала плодом совместных усилий и принадлежит всему человечеству.

Кстати говоря, если вы не в курсе, этим человеком был Э. Ф. Кодд, в то время работавший исследователем в корпорации IBM (Э – это Эдгар, Ф – Фрэнк, но он всегда подписывался одними инициалами; а для друзей, к которым я с гордостью отношу и себя, он был просто Тедом). В конце 1968 года Кодд, математик по образованию, впервые понял, как применить математику для закладывания строгих принципов в основание области знания – управления базами данных, – которой в то время таких качеств остро недоставало. Его первоначальное определение реляционной модели появилось в научно-исследовательском отчете IBM в 1969 году, и я еще вернусь к этой работе в приложении D.

Структурные свойства

В оригинальной модели было три основных компонента – структура, целостность и средства манипулирования, и я коротко опишу каждый из них. Но сразу прошу учесть, что все даваемые мной «определе-

ния» очень нестроги; я уточню их в последующих главах, когда это будет уместно.

Итак, начнем со структуры. Конечно, главным структурным свойством является само отношение, и, как всем известно, отношения принято изображать на бумаге в виде таблиц (пример на рис. 1.1 не нуждается в пояснениях). Отношения определены над *типами* (их называют также *доменами*). По существу, тип представляет собой концептуальное множество значений, которые могут принимать фактические атрибуты фактических отношений. Обращаясь к простой базе данных об отделах и служащих, которая показана на рис. 1.1, мы можем выделить тип DNO («номера отделов»), представляющий все допустимые номера отделов, и тогда атрибут DNO в отношении DEPT и атрибут DNO в отношении EMP будут принимать значения из соответствующего ему концептуального множества. (Кстати, атрибуты не обязательно – хотя иногда это и разумно – называть так же, как соответствующий тип, и часто так не делают. Ниже мы увидим много контрпримеров.)

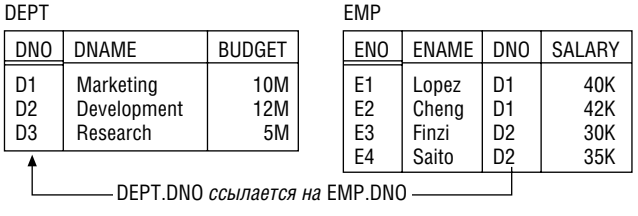


Рис. 1.1. *База данных об отделах и служащих – тестовые значения*

Как я уже сказал, таблицы, подобные представленной на рис. 1.1, изображают *отношения*, точнее *n*-арные отношения. *N*-арные отношения можно нарисовать в виде таблицы с *n* столбцами; столбцы будут соответствовать *атрибутам* отношения, а строки – *кортежам*. Здесь *n* – произвольное неотрицательное целое число. 1-арное отношение называют *унарным*, 2-арное – *бинарным*, 3-арное – *тернарным* и т. д.

Реляционная модель поддерживает также различные виды *ключей*. Начнем с того – и это крайне важно! – что в каждом отношении есть по меньшей мере один *потенциальный ключ*[1]. Потенциальный ключ – это просто уникальный идентификатор; иными словами, это комбинация атрибутов – часто, но не всегда, «комбинация» состоит всего из одного атрибута, – такая, что значения этой комбинации во всех кортежах отношения различны. Так, на рис. 1.1 у каждого отдела имеется уникаль-

[1] Строго говоря, это предложение должно выглядеть так: «Каждая *переменная-отношение* имеет по меньшей мере один потенциальный ключ» (см. раздел «Отношения и переменные-отношения» ниже). Аналогичное замечание относится и к другим местам в этой главе. См. упражнение 1.1 в конце главы.

ный номер отдела, а у каждого служащего – уникальный номер служащего, поэтому мы можем сказать, что {DNO} – потенциальный ключ отношения DEPT, а {ENO} – потенциальный ключ отношения EMP. Кстати, обратите внимание на фигурные скобки; повторю, что потенциальные ключи всегда являются комбинациями, или множествами, атрибутов – даже если конкретное множество состоит всего из одного атрибута, а множества на бумаге традиционно изображают в виде заключенного в фигурные скобки списка элементов, перечисленных через запятую.

Отступление

Здесь я впервые употребил термин *список* (разделенный запятыми) (*comma-list*), который далее будет встречаться очень часто. Его можно определить следующим образом. Пусть *xyz* – некоторая синтаксическая конструкция (например, «имя атрибута»). Тогда *список xyz* означает последовательность из нуля или более элементов *xyz*, в котором каждая пара соседних элементов разделена запятой (дополнительно перед и после запятой может находиться один или несколько пробелов). Так, если *A*, *B* и *C* – имена атрибутов, то каждая из показанных ниже конструкций представляет собой список имен атрибутов:

A , B , C

C , A , B

A , C

B

Списком является и пустая последовательность имен атрибутов. Кроме того, если список заключен в фигурные скобки и, следовательно, обозначает множество, то (а) порядок следования элементов в списке не существенен (так как множества по определению неупорядочены) и (б) если элемент встречается более одного раза, то считается, что он встретился только один раз (так как множества не содержат дубликатов).

Далее, *первичным* ключом называется такой потенциальный ключ, который по каким-то причинам интерпретируется специальным образом. Если рассматриваемое отношение имеет только один потенциальный ключ, то его с равным успехом можно назвать первичным. Но если потенциальных ключей несколько, то обычно один из них выбирается в качестве первичного, то есть считается, что он в каком-то отношении «равнее других». Предположим, к примеру, что у каждого служащего имеется как уникальный номер, так и уникальное имя – пример, пожалуй, не слишком реалистичен, но вполне пригоден для иллюстрации, – тогда и {ENO}, и {ENAME} являются потенциальными ключами EMP. В этом случае мы можем назначить, например, {ENO} первичным ключом.

Обратите внимание: я сказал, что первичный ключ *обычно* выбирается. Обычно – но не обязательно. Если имеется всего один потенциальный ключ, то вопрос о выборе вообще не встает; но если таких ключей два или больше, то выбор одного из них в качестве первичного от-

дает произвольностью (по крайней мере, мне так кажется). Разумеется, бывают ситуации, когда нет веских оснований отдать предпочтение конкретному кандидату. В этой книге я, как правило, буду выбирать какой-то первичный ключ – и на рисунках, подобных рис. 1.1, обозначать составляющие первичный ключ атрибуты двойным подчеркиванием, – но хочу особо подчеркнуть, что с точки зрения реляционной теории важность представляют именно потенциальные, а не первичные ключи. Отчасти по этой причине я в дальнейшем под словом *ключ*, без уточнения, буду понимать любой потенциальный ключ. (Если вам интересно, то скажу, что «специальная интерпретация» первичных ключей по сравнению с потенциальными носит в основном синтаксический характер; она не является фундаментальной особенностью и вообще не очень существенна.)

Наконец, *внешним* ключом называется множество атрибутов одного отношения, значения которых должны совпадать со значениями некоторого потенциального ключа в каком-то другом (или в том же самом) отношении. Так, на рис. 1.1 {DNO} – внешний ключ в отношении EMP, значения которого должны совпадать со значениями потенциального ключа {DNO} в отношении DEPT (что я и попытался отразить на рисунке посредством соответственно надписанной стрелочки). Говоря «должны совпадать», я имею в виду, что если отношение EMP содержит, к примеру, кортеж, в котором DNO имеет значение D2, то и DEPT должно содержать кортеж, в котором DNO имеет значение D2, – иначе в отношении EMP окажется служащий, который работает в несуществующем отделе, и база данных перестанет быть «верной моделью реальности».

Свойства целостности

Ограничение целостности (или, для краткости, просто *ограничение*) по существу представляет собой булево выражение, которое должно принимать значение TRUE. Например, для отделов и служащих мы могли бы ввести ограничение, которое гласит, что значение атрибута SALARY (зарплата) должно быть больше нуля. Вообще говоря, на любую базу данных налагается много разнообразных ограничений, однако все они по необходимости специфичны для конкретной базы и потому должны выражаться в терминах отношений в этой базе. Но в оригинальную реляционную модель включены также два *обобщенных* ограничения целостности – обобщенных в том смысле, что они применимы к любой базе данных (если говорить неформально). Одно касается первичных ключей, другое – внешних.

Правило целостности сущностей

Атрибуты, входящие в состав первичного ключа, не могут принимать null-значений.

Правило ссылочной целостности

Не должно быть внешних ключей, не имеющих соответствия.

Сначала я объясню смысл второго правила. Говоря о *внешнем ключе, не имеющем соответствия*, я имею в виду значение внешнего ключа, для которого не существует соответствующего ему потенциального ключа с таким же значением; например, в базе данных об отделах и служащих правило ссылочной целостности было бы нарушено, если бы в отношении EMP встретилось, к примеру, значение D2 атрибута DNO, но в отношении DEPT не оказалось бы кортежа с таким же значением DNO. Таким образом, правило ссылочной целостности просто выражает семантику внешних ключей, а название «ссылочная целостность» напоминает о том факте, что значение внешнего ключа можно рассматривать как *ссылку* на кортеж с таким же значением соответственного потенциального ключа. По существу, правило говорит: если *B* ссылается на *A*, то *A* должно существовать.

Что же касается правила целостности сущностей, то я, скажем так, испытываю затруднение. Дело в том, что я полностью отвергаю концепцию «null-значений»; по моему глубокому убеждению, *им не место в реляционной модели*. (Кодд считал иначе, очевидно, но и у меня есть сильные аргументы в защиту собственной позиции.) Поэтому, чтобы объяснить смысл правила целостности сущностей, мне придется (по крайней мере, на время) забыть о своем неверии. Что я и сделаю... но имейте в виду, что я еще вернусь к вопросу о null-значениях в главах 3 и 4.

По сути своей, null – это «маркер», говорящий, что *значение неизвестно* (подчеркну – и это принципиально важно, что null-значение – и не значение вовсе, а только *маркер*, или *флаг*). Допустим, к примеру, что мы не знаем величину зарплаты служащего E2. Тогда вместо того, чтобы вводить какое-то реальное значение SALARY в кортеж, описывающий этого служащего в отношении EMP, – по определению, мы не можем этого сделать, так как не знаем нужного значения, – мы *помечаем* атрибут SALARY в этом кортеже флагом null:

ENO	ENAME	DNO	SALARY
E2	Cheng	D1	

Важно понимать, что этот кортеж не содержит *ничего* в позиции SALARY. Но очень трудно изобразить «ничто» на рисунке! На рисунке выше я попытался показать, что позиция SALARY пуста, затенением ячейки, но было бы точнее не показывать ее вовсе. Так или иначе, я и дальше буду придерживаться того же соглашения: обозначать пустые позиции затенением, памятуя о том, что затенение говорит об отсутствии всякого значения. Если вам так удобнее, можете считать, что она представляет собой маркер, или флаг null.

Возвращаясь к соотношению EMP, правило целостности сущностей говорит (неформально выражаясь), что у любого служащего может быть неизвестно имя, отдел или зарплата, но не номер служащего, поскольку

ку, если неизвестен номер, то мы вообще не знаем, о каком служащем (сущности) идет речь.

Вот и все, что я хочу сказать о null-значениях на данный момент. И пока забудем о них.

Средства манипулирования

Раздел модели, относящийся к манипулированию, состоит из двух частей:

- *Реляционная алгебра*, в которой определяется набор операторов, например разность (или MINUS), применимых к отношениям.
- Оператор *реляционного присваивания*, который позволяет присвоить значение реляционного выражения (например, *r1 MINUS r2*, где *r1* и *r2* – отношения) какому-то отношению.

Оператор реляционного присваивания по существу описывает способ выполнения обновлений в реляционной модели, и я еще вернусь к нему ниже в разделе «Отношения и переменные-отношения». *Примечание*: в этой книге я придерживаюсь традиционного соглашения о том, что общим термином *обновление* (*update*) обозначается совокупность реляционных операторов INSERT, DELETE и UPDATE (а также присваивания). Когда речь идет о конкретном операторе UPDATE, я записываю его заглавными буквами.

Что касается реляционной алгебры, то она состоит из набора операторов, которые – говоря очень нестрого – позволяют порождать «новые» отношения из «старых». Каждый оператор принимает на входе одно или несколько отношений и на выходе возвращает новое отношение; например, оператор разности (MINUS) принимает на входе два отношения и «вычитает» одного из другого, порождая на выходе третье отношение. Очень важно, что результатом является отношение: это хорошо известное свойство *замкнутости* реляционной алгебры. Именно свойство замкнутости позволяет записывать вложенные реляционные выражения; так как результат любой операции – объект того же вида, что и входные операнды, то результат одной операции можно подать на вход другой, – например, разность *r1 MINUS r2* может быть объединена с отношением *r3*, затем результат пересечен с отношением *r4* и т. д.

Можно определить сколько угодно операторов, удовлетворяющих простому условию: «одно или несколько отношений на входе, единственное отношение на выходе». Ниже я кратко опишу те операторы, которые принято считать исходными (по существу, те, что были определены Коддом в ранних работах)[1]; в главах 6 и 7 я остановлюсь на них бо-

[1] Исключение составляет дополнительно определенный Коддом оператор деления *divide*. В главе 7 я объясню, почему опустил его здесь.

лее подробно и определю некоторые дополнительные операторы. На рис. 1.2 показано графическое представление исходных операторов. Примечание: если вы незнакомы с этими операторами и не понимаете, что означают картинки, не расстраивайтесь; как я уже сказал, в последующих главах они будут описаны детально, с множеством примеров.

Ограничение[1]

Возвращает отношение, содержащее все кортежи заданного отношения, которые удовлетворяют заданному условию. Например, можно ограничить отношение EMP только кортежами, для которых атрибут DNO имеет значение D2.

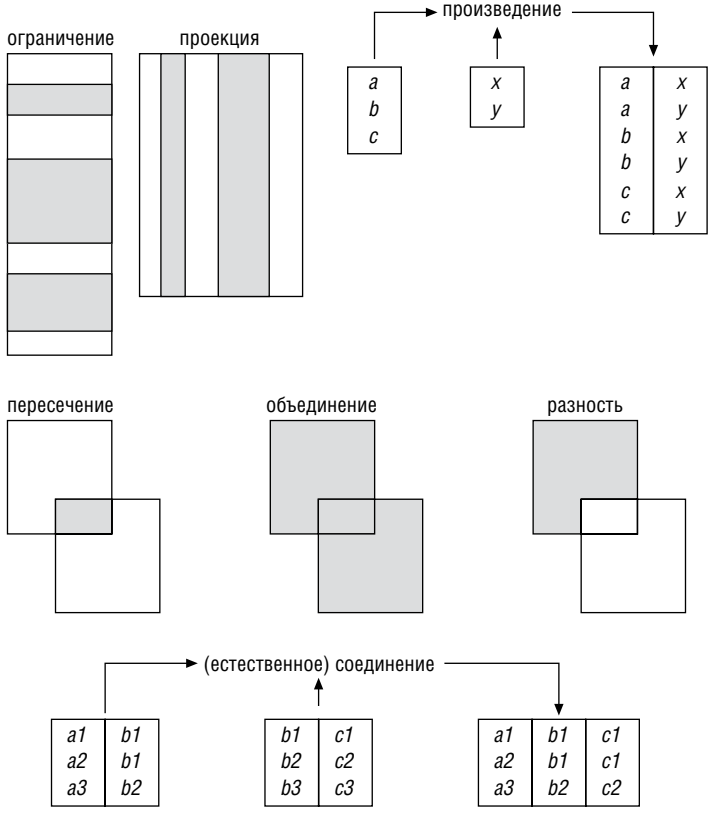


***Рис. 1.2.** Оригинальная реляционная алгебра*

[1] Слова *constraint* (ограничение целостности) и *restriction* (выборка) переводятся на русский язык одним словом *ограничение*. К счастью, из контекста всегда понятно, о чем именно идет речь. *–Прим. перев.*

Проекция

Возвращает отношение, содержащее все (под)кортежи заданного отношения, которые остались после исключения из него некоторых атрибутов. Например, можно спроецировать отношение EMP на атрибуты ENO и SALARY (исключив тем самым атрибуты ENAME и DNO).

Произведение

Возвращает отношение, содержащее все возможные кортежи, которые являются комбинацией двух кортежей, принадлежащих соответственно двум заданным отношениям. Этот оператор известен также как *декартово произведение* (иногда он называется *расширенным декартовым произведением*), *перекрестное произведение*, *перекрестное соединение* или *декартово соединение*; по сути дела, это просто частный случай операции соединения, как будет показано в главе 6.

Пересечение

Возвращает отношение, содержащее все кортежи, которые принадлежат одновременно двум заданным отношениям. (На самом деле, пересечение также является частным случаем соединения, как будет показано в главе 6.)

Объединение

Возвращает отношение, содержащее все кортежи, которые принадлежат либо одному из двух заданных отношений, либо им обоим.

Разность

Возвращает отношение, содержащее все кортежи, которые принадлежат первому из двух заданных отношений, но не принадлежат второму.

Соединение

Возвращает отношение, содержащее все возможные кортежи, которые представляют собой комбинации двух кортежей, принадлежащих двум заданным отношениям, при условии, что в комбинируемых кортежах присутствуют одинаковые значения в одном или нескольких общих для исходных отношений атрибутах (причем эти общие значения появляются в результирующем кортеже один раз, а не дважды). *Примечание*: Соединение такого вида изначально называлось *естественным*. Однако, поскольку естественное соединение несомненно является наиболее важным, то теперь принято называть его просто соединением, опуская уточняющий квалификатор; я тоже буду придерживаться такого соглашения.

И еще одно заключительное замечание: как вы, возможно, знаете, существует еще так называемое реляционное исчисление. Его можно счи-

тать альтернативой реляционной алгебре, то есть с равным успехом можно сказать, что манипуляционная часть реляционной модели состоит из реляционной алгебры (и оператора реляционного присваивания) или из реляционного исчисления (и оператора реляционного присваивания). Они эквивалентны и взаимозаменяемы в том смысле, что для каждого алгебраического выражения существует логически эквивалентное ему выражение реляционного исчисления и наоборот. Я еще буду говорить о реляционном исчислении, в основном в главах 10 и 11.

Сквозной пример

Я завершу этот краткий обзор описанием примера, который будет основой при обсуждении большинства, если не всех, тем в этой книге: всем известной – если не сказать навязшей в зубах – базы данных о поставщиках и деталях. (Прошу прощения за то, что еще раз вывел этого престарелого боевого коня из стойла, но я полагаю, что использование одного и того же примера в разных публикациях способствует, а не мешает обучению.) На рис. 1.3 приведены тестовые значения.

S

SNO	SNAME	STATUS	CITY
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

SP

SNO	PNO	QTY
S1	P1	300
S1	P2	200
S1	P3	400
S1	P4	200
S1	P5	100
S1	P6	100
S2	P1	300
S2	P2	400
S3	P2	200
S4	P2	200
S4	P4	300
S4	P5	400

P

PNO	PNAME	COLOR	WEIGHT	CITY
P1	Nut	Red	12.0	London
P2	Bolt	Green	17.0	Paris
P3	Screw	Blue	17.0	Oslo
P4	Screw	Red	14.0	London
P5	Cam	Blue	12.0	Paris
P6	Cog	Red	19.0	London

Рис. 1.3. *База данных о поставщиках и деталях – тестовые значения*

Уточним понятия.

Поставщики

Отношение S описывает поставщиков (точнее, поставщиков по контракту). У каждого поставщика имеется уникально определяющий его номер (SNO) (как видите, я сделал {SNO} первичным ключом), название (SNAME), необязательно уникальный (пусть даже на рис. 1.3 все значения SNAME различаются) статус (STATUS), представляющий некое свойство, позволяющее предпочесть одного поставщика другому, и местоположение (CITY).

Детали

Отношение P представляет детали (точнее, виды деталей). У каждой детали есть уникальный номер (PNO) ({PNO} – первичный ключ), одно наименование (PNAME), один цвет (COLOR), один вес (WEIGHT) и одно местоположение, то есть склад, где хранятся детали этого вида (CITY).

Поставки

Отношение SP описывает поставки (то есть показывает, какие детали поставляются какими поставщиками). У каждой поставки имеется один номер поставщика (SNO), один номер детали (PNO) и одно количество (QTY). В этом примере я буду предполагать, что в любой момент времени существует не более одной поставки с данным поставщиком и данной деталью ({SNO,PNO} – первичный ключ, кроме того, {SNO} и {PNO} являются внешними ключами, сопоставляемыми первичным ключам S и P, соответственно). Обратите внимание, что в базе данных на рис. 1.3 есть один поставщик, S5, для которого нет ни одной поставки.

Модель и реализация

Прежде чем двигаться дальше, я хочу остановиться на одном моменте, существенном для всего, что будет обсуждаться ниже. Разумеется, реляционная модель – это модель данных. Но, к сожалению, последний термин имеет в мире баз данных два совершенно разных значения. Первое и наиболее фундаментальное таково:

> **Определение**: *Модель данных* (в первом смысле) – это абстрактное, независимое, логическое определение структур данных, операторов над данными и прочего, что в совокупности составляет *абстрактную систему*, с которой взаимодействует пользователь.

Именно это значение слова мы имеем в виду, когда говорим о реляционной модели. И, вооружившись этим определением, мы можем провести полезное, и важное, различие между моделью данных в первом смысле и ее реализацией, которая определяется следующим образом.

> **Определение**: *Реализацией* данной модели данных называется физическое воплощение на реальной машине тех компонентов абстрактной системы, которые в совокупности составляют модель.

Я проиллюстрирую эти определения в терминах реляционной модели. Прежде всего, само понятие *отношения* является частью модели данных: пользователи должны знать, что такое отношение, понимать, что оно состоит из кортежей и атрибутов, уметь их интерпретировать и т. д.

Все это части модели. Но пользователям необязательно знать, как отношения физически хранятся на диске, как физически кодируются отдельные значения данных, какие существуют индексы или другие пути доступа к данным – это части реализации, а не модели.

Или возьмем концепцию *соединения*: пользователи должны знать, что такое соединение, как его осуществить, как выглядит результат соединения и т. д. Опять-таки, все это части модели. Но совершенно ни к чему знать, как физически реализуется соединение, какие при этом производятся трансформации выражений, какие используются индексы или иные пути доступа к данным, какие требуются операции ввода/вывода – это части реализации, а не модели.

И еще один пример: *потенциальные ключи* (для краткости просто *ключи*) – часть модели, и пользователям определенно необходимо знать, что представляют собой ключи. На практике уникальность ключей часто гарантируется посредством так называемого уникального индекса; но индексы вообще и уникальные индексы в частности не являются частью модели, это часть реализации. Таким образом, не следует путать индекс с ключом в реляционном смысле, даже если первый применяется для реализации второго (точнее, для реализации некоторого *ограничения ключа*, см. главу 8).

Короче говоря:

- Модель (в первом смысле) – это то, что пользователь должен знать.
- Реализация – это то, что пользователю знать необязательно.

Я вовсе не хочу сказать, что пользователям запрещено знать о реализации; я лишь говорю, что это необязательно. Иными словами, все касающееся реализации должно быть, по крайней мере потенциально, *скрыто от пользователя*.

Из данных выше определений вытекают некоторые важные следствия. Прежде всего (и вопреки чрезвычайно распространенному заблуждению), все связанное с производительностью принципиально является деталью реализации, а не модели. Например, мы часто слышим, что «соединение – медленная операция». Но такое замечание лишено всякого смысла! Соединение – это часть модели, а модель как таковая не может быть ни медленной, ни быстрой; такими качествами может обладать только реализация. Поэтому допустимо сказать, что в некотором продукте X конкретная операция соединения реализована быстрее или медленнее, чем в продукте Y, – но это и все.

Я не хочу, чтобы у вас в этом месте сложилось ложное впечатление. Да, производительность принципиально является деталью реализации; однако это не означает, что хорошая реализация будет работать быстро, если модель используется неправильно. На самом деле, это как раз одна из причин, по которым вы должны знать устройство модели: чтобы не использовать ее неправильно. Написав выражение S JOIN SP, вы впра-

ве ожидать, что реализация сделает все необходимое и сделает хорошо; но если вы настаиваете на ручном кодировании соединения, например таким образом (в виде псевдокода):

```
do for all tuples in S ;
   fetch S tuple into TNO , TN , TS , TC ;
   do for all tuples in SP with SNO = TNO ;
      fetch SP tuple into TNO , TP , TQ ;
      emit tuple TNO , TN , TS , TC , TP , TQ ;
   end ;
end ;
```

то рассчитывать на достижение высокой производительности было бы глупо. **Рекомендация**: Не поступайте так. Реляционные системы не следует использовать в качестве простых методов доступа.[1]

Кстати, эти замечания по поводу производительности относятся и к SQL. Утверждения о быстроте или медленности точно так же бессмысленны в применении к SQL, как и к реляционным операторам (соединению и прочим); в этих терминах имеет смысл говорить только о реализациях. Однако использовать SQL можно и так, что это гарантированно приведет к плохой производительности. Хотя в этой книге тема производительности почти не затрагивается, иногда я все же буду отмечать последствия своих рекомендаций с этой точки зрения.

Отступление

Я хотел бы ненадолго задержаться на вопросе о производительности. Вообще говоря, приводя в этой книге ту или иную рекомендацию, я не руководствовался соображениями производительности; в конце концов, реляционная модель всегда ставила целью передать заботу о производительности от пользователя системе. Однако не стоит и говорить, что эта задача так и не была решена в полной мере, и потому (как я уже отмечал) поставленной целью – реляционное использование SQL – иногда приходится жертвовать в интересах достижения приемлемой производительности. И это еще одна причина, по которой действует универсальное правило: *можете делать все, что хотите, если только знаете, что делаете*.

Но вернемся к различию между моделью и реализацией. Второй момент, как вы, вероятно, догадались, состоит в том, что именно разделение модели и реализации позволяет добиться *физической независимости от данных*. Физическая независимость от данных – термин, по-

[1] Сразу несколько рецензентов обратили внимание на то, что это предложение лишено смысла (как можно использовать систему в качестве метода?). Что ж, если вы настолько молоды, что не знакомы с термином *метод доступа*, могу только позавидовать; но факт в том, что этот термин, пусть не очень подходящий, в прошлом широко использовался для обозначения того или иного простого механизма ввода/вывода на уровне отдельных записей.

жалуй, не слишком удачный, но он уже устоялся, – означает, что мы можем как угодно изменять способ физического хранения и доступа к данным, не внося соответствующих изменений в то, как данные представляются пользователю. Причина, по которой может возникнуть желание изменить детали хранения и доступа, как правило, связана с производительностью; а тот факт, что такие изменения можно производить, не меняя способа представления данных пользователю, означает, что существующие программы, запросы и прочее будут продолжать работать. Таким образом, – и это очень важно – физическая независимость от данных позволяет *уменьшить капиталовложения в обучение пользователей и разработку приложений* и, добавлю еще, капиталовложения в проектирование логической структуры базы данных.

Из всего сказанного следует, что (как отмечалось выше) индексы, да и вообще все виды физических путей доступа, являются составной частью реализации, а не модели; их место «под капотом», где они не видны пользователю. (Отмечу, что пути доступа как таковые вообще не упоминаются в реляционной модели.) По тем же причинам их, строго говоря, следовало бы исключить и из SQL. **Рекомендация**: Избегайте любых конструкций SQL, нарушающих эту заповедь. (Насколько я знаю, в стандарте нет ничего, вступающего в противоречие с этой рекомендацией, чего нельзя сказать о некоторых продуктах на основе SQL.)

Как бы то ни было, из приведенных выше определений видно, что различие между моделью и реализацией в действительности является частным – хотя и очень важным – случаем всем знакомого общего различия между логической и физической организацией. Однако, как это ни печально, в большинстве современных SQL-систем оно не проводится так четко, как следовало бы. Отсюда немедленно следует, что они в гораздо меньшей степени физически независимы от данных, чем хотелось бы, и не обеспечивают тех свойств, которые должны присутствовать в реляционной системе. В следующем разделе я еще вернусь к этой теме.

А теперь я хочу обратиться ко второму значению термина *модель данных*, с которым, смею предположить, вы отлично знакомы. Его можно определить следующим образом:

> **Определение**: *Моделью данных* (во втором смысле) называется модель данных (особенно сохраняемых) конкретного предприятия.

Иными словами, модель данных во втором смысле – это просто (логическая и, возможно, до некоторой степени абстрактная) структура базы данных. Например, можно говорить о модели данных банка, больницы или правительственного учреждения.

Объяснив оба смысла, я хочу привлечь ваше внимание к аналогии, которая, как мне кажется, отлично иллюстрирует соотношение между ними.

- Модель данных в первом смысле – аналог языка программирования, конструкции которого применимы для решения многих конкретных задач, но сами по себе не связаны ни с какой отдельной задачей.
- Модель данных во втором смысле – аналог конкретной программы, написанной на этом языке, – в ней используются средства, предоставляемые моделью в первом смысле, для решения конкретной задачи.

Кстати говоря, из сказанного выше следует, что если мы ведем речь о моделях данных во втором смысле, то можем без опасения говорить о «реляционных моделях» во множественном числе или о «некоторой» реляционной модели. Но модель данных в первом смысле только одна, и это *та самая реляционная модель*, которую придумал Кодд. Последнюю мысль я еще разовью в приложении А.

Далее в книге я буду употреблять термин *модель данных*, или для краткости просто *модель*, исключительно в первом смысле.

Свойства отношений

Теперь вернемся к изучению основных понятий реляционной теории. В этом разделе я хочу остановиться на некоторых свойствах самих отношений. Прежде всего, у каждого отношения есть *заголовок* и *тело*. Заголовком называется множество атрибутов (здесь под *атрибутом* я понимаю пару имя-атрибута/имя-типа), а телом – множество кортежей, согласованных с заголовком. Так, у отношения «поставщики» на рис. 1.3 заголовок состоит из четырех атрибутов, а тело – из пяти кортежей. Таким образом, отношение не содержит кортежей – оно содержит тело, которое, в свою очередь, содержит кортежи, – но обычно мы для простоты говорим, что кортежи содержит само отношение.

Попутно отмечу, что хотя правильно было бы говорить, что заголовок состоит из пар имя-атрибута/имя-типа, обычно имена типов на рисунках, подобных рис. 1.3, опускаются, из-за чего создается впечатление, будто заголовок состоит только из имен атрибутов. Например, атрибут STATUS имеет тип, – допустим, INTEGER, – но на рис. 1.3 он не показан. Однако не следует забывать, что он все-таки существует!

Далее, количество атрибутов в заголовке называется *степенью* (или *арностью*), а количество кортежей в теле – *кардинальностью*. Например, для отношений S, P и SP на рис. 1.3 степень равна соответственно 4, 5 и 3, а кардинальность – 5, 6 и 12. *Примечание*: термин *степень* применяется также к кортежам. Так, все кортежи в отношении S имеют степень 4 (как и само отношение S).

Отношения *никогда* не содержат кортежей-дубликатов. Это следует из того, что тело определяется как множество кортежей, а математическое множество по определению не содержит дубликатов. В этом от-

ношении SQL, как вы, конечно, знаете, отступает от теории: таблицы в SQL могут содержать строки-дубликаты и потому в общем случае не являются отношениями. Хочу подчеркнуть, что в этой книге термином *отношение* я всегда называю истинное отношение – без кортежей-дубликатов, – а не таблицу SQL. Кроме того, вы должны понимать, что реляционные операции всегда порождают результат без кортежей-дубликатов, в полном соответствии с определением. Например, проекция отношения «поставщики» на рис. 1.3 на атрибут CITY дает результат, показанный ниже на левом, а не на правом рисунке:

CITY
London
Paris
Athens

CITY
London
Paris
Paris
London
Athens

(Изображенный на левом рисунке результат можно получить SQL-запросом SELECT DISTINCT CITY FROM S. Если опустить слово DISTINCT, то получится нереляционный результат, показанный справа. Особо отметим, что в таблице справа нет двойного подчеркивания, поскольку в ней нет никакого ключа, а уж тем более первичного.)

Далее, кортежи отношения не упорядочены. Это свойство также следует из того, что тело определено как множество, а элементы математического множества не упорядочены (стало быть, {*a*,*b*,*c*} и {*c*,*a*,*b*} в математике считаются одним и тем же множеством, и, естественно, то же справедливо и для реляционной модели). Конечно, когда мы рисуем отношение на бумаге в виде таблицы, мы должны располагать строки сверху вниз, но в таком порядке нет ничего реляционного. Так, строки отношения «поставщики» на рис. 1.3 я мог бы расположить в любом порядке, скажем, сначала S3, потом S1, потом S5, S4 и S2, и это было бы то же самое отношение. *Примечание*: Тот факт, что отношения не упорядочены, еще не означает, что в запрос нельзя включать фразу ORDER BY, но означает, что результат, возвращенный таким запросом, не является отношением. Фраза ORDER BY полезна для представления результатов, но сама по себе не является реляционным оператором.

Аналогично, не упорядочены и атрибуты отношения, поскольку заголовок также является математическим множеством. Изображая отношение в виде таблицы на бумаге, мы по необходимости располагаем столбцы в некотором порядке – слева направо, но в таком порядке нет ничего реляционного. Так, для отношения «поставщики» на рис. 1.3 я мог бы расположить столбцы в любом другом порядке, скажем, STATUS, SNAME, CITY, SNO, и с точки зрения реляционной модели это было бы то же самое отношение (это еще одна причина, по которой таблицы SQL вообще говоря не являются отношениями). Например, на обоих рисунках ниже представлено одно и то же отношение, но разные таблицы SQL:

SNO	CITY
S1	London
S2	Paris
S3	Paris
S4	London
S5	Athens

CITY	SNO
London	S1
Paris	S2
Paris	S3
London	S4
Athens	S5

(В SQL эти представления порождаются соответственно запросами SELECT SNO, CITY FROM S и SELECT CITY, SNO FROM S. Возможно, вам кажется, что различие между этими двумя запросами и таблицами несущественно, но на самом деле последствия весьма серьезны, и о некоторых из них я расскажу в последующих главах. См., например, обсуждение SQL-оператора JOIN в главе 6.)

Наконец, отношения всегда *нормализованы* (или, что то же самое, находятся *в первой нормальной форме*, 1НФ).[1] Неформально это означает, что в табличном представлении отношения на пересечении любой строки и любого столбца мы видим только одно значение. Более формально – каждый кортеж отношения содержит единственное значение соответствующего типа в позиции любого атрибута. В следующей главе я буду говорить об этом гораздо более подробно.

И в заключение я хотел бы еще раз акцентировать ваше внимание на моменте, о котором упоминал неоднократно: существует логическое различие между отношением как таковым и его изображением, например, на рис. 1.1 и 1.3. Повторю, что конструкции, показанные на рис. 1.1 и 1.3, вообще не являются отношениями, а лишь изображениями отношений, которые я обычно называю таблицами, несмотря на то, что это слово уже наделено определенной семантикой в контексте SQL. Конечно, между отношениями и таблицами есть определенное сходство, и в неформальном контексте о них обычно говорят, как об одном и том же, – и, пожалуй, это допустимо. Но если требуется точность – а как раз сейчас я пытаюсь быть точным, – то необходимо понимать, что эти два понятия не идентичны.

Попутно отмечу, что и в более общем плане существует логическое различие между произвольной сущностью и изображением этой сущности. Эта мысль замечательно проиллюстрирована на знаменитой картине Магритта. На картине изображена обычная курительная трубка, но под ней Магритт написал *Ceçi n'est pas une pipe*... – смысл, конечно, в том, что рисунок – это не трубка, а лишь изображение трубки.

Ко всему вышесказанному я хочу добавить, что на самом деле тот факт, что основной абстрактный объект реляционной модели – отношение –

[1] Мы говорим о «первой» нормальной форме, потому что, как вы наверняка знаете, можно определить последовательность нормальных форм «высших порядков» – вторую нормальную форму, третью нормальную форму и т. д., – что существенно для проектирования баз данных. См. упражнение 2.9 в главе 2, а также приложение В.

имеет такое простое представление на бумаге, составляет важное достоинство этой модели; именно наличие простого представления и делает реляционные системы простыми для понимания и использования и позволяет без труда рассуждать об их поведении. Но, к сожалению, это же свойство может приводить к неверным выводам (например, о существовании некоего порядка кортежей – сверху вниз).

И еще один момент: я сказал, что существует логическое различие между отношением и его изображением. Концепция *логического различия* вытекает из следующего афоризма Виттгенштейна:

> Любое логическое различие является существенным различием.

Это замечание необычайно полезно; как «мыслительный инструмент» оно весьма способствует четким и ясным рассуждениям и очень помогает при выявлении и анализе некоторых путаных мест, которые, к несчастью, так часто встречаются в мире баз данных. Я еще не раз вернусь к нему на страницах этой книги. А пока позвольте мне выделить те логические различия, с которыми мы уже столкнулись:

- SQL и реляционная модель
- Модель и реализация
- Модель данных (в первом смысле) и модель данных (во втором смысле)

В последующих трех разделах будут описаны и другие различия.

Базовые и производные отношения

Выше я уже объяснял, что операторы реляционной алгебры позволяют, начав с некоторого набора исходных отношений, – скажем, изображенных на рис. 1.3, – получить новые отношения (например, с помощью запросов). Исходные отношения называются *базовыми*, остальные – *производными*. Следовательно, чтобы было от чего отталкиваться, реляционная система должна предоставлять средства для определения базовых отношений. В SQL эта задача решается предложением CREATE TABLE (в SQL базовому отношению, естественно, соответствует базовая таблица). Очевидно, что базовые отношения должны быть поименованы, например:

```
CREATE TABLE S ... ;
```

Но некоторым производным отношениям, в частности так называемым представлениям, также присваиваются имена. *Представлением* (или *виртуальным отношением*) называется именованное отношение, значением которого в каждый момент времени t является результат вычисления некоторого реляционного выражения в этот момент. Вот как это может выглядеть на языке SQL:

```
CREATE VIEW SST_PARIS AS
   SELECT SNO , STATUS
```

```
FROM   S
WHERE  CITY = 'Paris' ;
```

В принципе, представлениями можно оперировать так же, как базовыми отношениями[1], но в действительности они таковыми не являются. Альтернативно можно считать представление «материализованным», то есть мысленно воображать, что в тот момент, когда производится ссылка на представление, создается базовое отношение, значением которого является результат вычисления заданного реляционного выражения. Однако я должен подчеркнуть, что идея о подобной материализации представлений в момент ссылки является чисто концептуальной; ничего подобного в реальности не происходит, и для операций обновления это в любом случае не сработало бы. Как на самом деле задумывалось функционирование представлений, мы рассмотрим в главе 9.

Попутно хочу сделать важное замечание. Часто приходится слышать такое описание различий между базовыми отношениями и представлениями:

- Базовые отношения реально существуют, то есть физически хранятся в базе данных.
- Напротив, представления «реально не существуют» – это лишь альтернативный способ взглянуть на базовое отношение.

Но реляционная модель ничего не говорит о том, что именно физически хранится! В частности, нигде не утверждается, что базовые отношения физически хранятся. Требуется лишь, чтобы существовало некоторое отображение между тем, что физически хранится, и базовыми отношениями, так чтобы базовое отношение можно было каким-то образом получить, когда в нем возникнет необходимость (по крайней мере, концептуально). Если таким образом можно получить базовое отношение, то можно получить и все остальное. Например, мы могли бы физически хранить соединение поставщиков и поставок, но не хранить их по отдельности; тогда базовые отношения S и SP концептуально можно было бы получить с помощью подходящих проекций этого соединения. Скажем по-другому: с точки зрения реляционной модели, базовые отношения не более (но и не менее!) «физические», чем представления.

Реляционная модель вполне сознательно ничего не говорит о физическом хранении. Идея заключалась в том, чтобы оставить разработчикам максимум свободы в реализации модели любым удобным для них способом – в частности, наиболее подходящим для обеспечения высокой производительности, – не жертвуя принципом физической неза-

[1] Возможно, вам кажется, что это утверждение не может быть стопроцентно справедливым для операций обновления. Если так, то вы правы с точки зрения современных продуктов; однако я все же настаиваю на том, что в принципе это утверждение верно. Дальнейшее обсуждение см. в разделе «Операции обновления» в главе 9.

висимости от данных. Печально, однако, что поставщики SQL-систем по большей части, похоже, недопоняли этот аспект; они напрямую отображают базовые таблицы на физические хранимые объекты[1], и потому (как уже отмечалось выше) созданные ими продукты демонстрируют гораздо меньшую физическую независимость от данных, чем заложено в реляционной модели. Более того, такое положение вещей отражено и в самом стандарте SQL (а также в большинстве других относящихся к SQL документов), где обычно – и на самом деле, в очень многих местах – употребляются выражения типа «таблицы и представления». Очевидно, что человек, который так говорит, полагает, что таблицы и представления – вещи разные, и, возможно, находится под впечатлением, будто таблицы – физические объекты, а представления – нет. Но весь смысл представления в том, что это такая же таблица (хотя я предпочел бы сказать – отношение), поэтому к представлениям применимы те же операции, что к обычным отношениям (по крайней мере, в реляционной модели), так как представления *и есть* «обычные отношения». В этой книге я буду употреблять термин *отношение* для обозначения любого отношения (базового, представления, результата запроса и т. д.); если же мне понадобится (например) уточнить, что речь идет о базовом отношении, я так и напишу «базовое отношение». **Рекомендация**: настоятельно рекомендую и вам придерживаться такой же дисциплины. Не делайте распространенную ошибку, мысленно применяя термин *отношение* только к базовым отношениям, или – если говорить в терминах SQL – называя *таблицами* только базовые таблицы.

Отношения и переменные-отношения

Очень может быть, что все, о чем я рассказывал в этой главе до сих пор, вам уже знакомо; я даже искренне надеюсь, что это так, хотя не в меньшей степени надеюсь, что вы не сочли текст скучным. Но так или иначе, сейчас я подхожу к теме, которая вам, возможно, незнакома. Дело в том, что исторически возникла серьезная путаница, связанная еще с одним логическим различием: между отношениями и *переменными*-отношениями.

Давайте на минутку забудем о базах данных и отношениях и рассмотрим пример простого языка программирования. Предположим, что я написал на некотором языке такое предложение:

```
DECLARE N INTEGER ... ;
```

[1] Я говорю это, отчетливо понимая, что современные SQL-системы предлагают многообразные средства для хеширования, секционирования, индексирования, кластеризации и иных методов организации данных на диске. И тем не менее я считаю, что отображение на физически хранимые объекты в этих продуктах излишне прямолинейно.

Здесь N – *не целое число*, а *переменная*, которая может принимать целые числа в качестве *значений* – разных в разные моменты времени. Все мы это прекрасно понимаем. Точно так же, когда в SQL я пишу:

```
CREATE TABLE T ... ;
```

T *не является таблицей*; это табличная переменная, или (как я предпочитаю говорить, игнорируя такие особенности SQL, как null-значения и строки-дубликаты) переменная-отношение, значениями которой являются отношения (разные в разные моменты времени).

Взгляните еще раз на рис. 1.3, где изображена база данных о поставщиках и деталях. На этом рисунке мы видим три отношения-*значения*, а именно те, которые имели место в базе данных в какой-то момент времени. Но, взглянув на ту же базу данных в другой момент, мы, возможно, увидели бы другие три отношения-значения. Иными словами, S, P и SP в этой базе данных действительно являются переменными – точнее, *переменными-отношениями*. Например, предположим, что переменная-отношение S в настоящий момент времени имеет значение – отношение-значение, – показанное на рис. 1.3, и мы удаляем кортежи (на самом деле, только один кортеж) для поставщиков в Афинах:

```
DELETE S WHERE CITY = 'Athens' ;
```

Вот как будет выглядеть результат:

S

SNO	SNAME	STATUS	CITY
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London

Концептуально старое значение S было целиком заменено новым. Конечно, старое значение (с пятью кортежами) и новое (с четырьмя кортежами) в некотором смысле очень похожи, но все-таки это разные значения. Фактически показанная выше операция DELETE логически эквивалентна и по существу является сокращенной записью следующего реляционного присваивания:

```
S := S WHERE NOT ( CITY = 'Athens' ) ;
```

Как и при любом присваивании, здесь выполняются два действия: (а) вычисляется выражение в правой части и (б) результат вычисления присваивается переменной в левой части. А что получается в результате, я уже объяснил.

Отступление

Я не могу записать показанное выше присваивание на языке SQL, потому что SQL не поддерживает реляционное присваивание. Вместо этого я записал его (равно как и исходное предложение DELETE) на специальном языке **Tutorial D**, синтаксис которого более-менее очевиден. Язык **Tutorial D** придумал Хью

Дарвен (Hugh Darwen), я использовал его для иллюстрации реляционных идей в нашей совместной книге «Databases, Types, and the Relational Model: The Third Manifesto» (см. приложение D) и буду использовать также в этой книге, когда понадобится объяснить какие-то реляционные концепции. Но поскольку предполагаемая аудитория на этот раз состоит из практических пользователей SQL, то я в большинстве случаев буду приводить эквивалентные конструкции на SQL.

Всем известные предложения INSERT и UPDATE также по существу являются сокращенной записью реляционного присваивания. Поэтому, как я уже отмечал в разделе «Обзор оригинальной модели», реляционное присваивание – это фундаментальный оператор обновления в реляционной модели; логически это вообще единственный оператор обновления, который нам нужен.

Таким образом, существует логическое различие между отношениями-значениями и переменными-отношениями. Беда в том, что в литературе по базам данных исторически применялся один и тот же термин, *отношение*, для обозначения обоих понятий, и такая практика, конечно же, привела к путанице[1]. Но, начиная с этого момента, я буду скрупулезно различать эти смыслы, четко обозначая, что имеется в виду: отношение-значение или переменная-отношение. Однако выражение отношение-значение я как правило буду сокращать до просто «отношение» (точно так же, как вместо целочисленное значение мы обычно говорим *целое число*).

В качестве упражнения можете еще раз прочитать текст настоящей главы и отметить те места, где я использовал термин *отношение*, когда должен был бы написать *переменная-отношение*.

Значения и переменные

Логическое различие между отношениями и переменными-отношениями на самом деле является частным случаем общего логического различия между значениями и переменными, и я хочу немного задержаться на рассмотрении общего случая. (Конечно, это отклонение от темы, но думаю, что время будет потрачено не зря, поскольку ясное понимание этого вопроса может оказать неоценимую помощь и в других случаях.) Сначала некоторые определения.

> **Определение**: *Значение* – это то, что в логике называется «индивидуальной константой», например целое число 3. У значения нет позиции во времени и пространстве. Однако значения могут быть представлены в памяти посред-

---

[1] Разумеется, такая же ошибка присутствует и в языке SQL, где одним и тем же термином, *таблица*, иногда обозначается табличное значение, а иногда – табличная переменная.

ством некоторой кодировки, и у таких представлений, или видов, имеется позиция во времени и пространстве. Неформально говоря, это означает, что сколько угодно различных переменных (см. следующее определение) могут иметь одно и то же значение одновременно или в разное время. Особо отметим, что (по определению) значение невозможно модифицировать; если бы это было допустимо, то после изменения оно перестало бы быть тем же самым значением.

Определение: *Переменная* – это контейнер для представления значения. Переменная имеет позицию во времени и пространстве. Переменные, в отличие от значений, можно модифицировать, то есть текущее значение данной переменной можно заменить другим значением, возможно, отличным от исходного. (Ведь само слово «переменная» подразумевает возможность изменения; иначе говоря, переменной можно присваивать значения.)

Отметим, что допустимы не только такие простые значения, как целое число 3. Значение может быть сколь угодно сложным: геометрической точкой, многоугольником, рентгеновским лучом, XML-документом, отпечатком пальца, массивом, стеком, списком, отношением (перечень можно продолжать до бесконечности). То же самое, конечно, относится и к переменным. Я еще буду говорить на эту тему в следующих двух главах.

Может показаться, что трудно не увидеть различия между такими очевидными и фундаментальными понятиями, как значения и переменные. Но на самом деле в эту ловушку попасть очень просто. Вот, к примеру, цитата из пособия по объектным базам данных (комментарии курсивом в скобках добавлены мной):

> Мы отличаем объявленный тип переменной от ... типа объекта, являющегося текущим значением этой переменной [*следовательно, объект является значением*]... Мы различаем объекты и значения [*следовательно, объект таки не является значением*]... Изменяющий оператор – [*это такой оператор, для которого,*] возможно наблюдать эффект его применения к некоторому объекту [*получается, что объект является переменной*].

Заключительные замечания

Основная цель этой вступительной главы заключалась в том, чтобы рассказать о том, что вы, как я надеюсь, уже знаете (и у вас могло сложиться впечатление, что она мало что добавляет к техническим материям). Но давайте все же резюмируем:

- Я объяснил, почему нас должны интересовать принципы, а не продукты, и почему (по крайней мере, в реляционных контекстах) я буду

пользоваться формальной терминологией (*отношения*, *кортежи*, *атрибуты*) вместо «более дружественной» терминологии SQL.

- Я привел обзор оригинальной модели, затронув, в частности, следующие понятия: *тип*, *n-арное отношение*, *кортеж*, *атрибут*, *потенциальный ключ*, *первичный ключ*, *внешний ключ*, *целостность сущностей*, *ссылочная целостность*, *реляционное присваивание* и *реляционная алгебра*. Говоря об алгебре, я упомянул свойство *замкнутости* и очень коротко остановился на операторах *ограничения*, *проекции*, *произведения*, *пересечения*, *объединения*, *разности* и *соединения*.
- Я рассмотрел различные свойства отношений, ввел понятия *заголовка*, *тела*, *кардинальности* и *степени*. Отношение не может содержать кортежей-дубликатов, кортежи не упорядочены сверху вниз, а атрибуты не упорядочены слева направо. Мы также обсудили различия между *базовыми отношениями* (точнее, базовыми переменными-отношениями) и *представлениями*.
- Я рассмотрел логические различия между *моделью* и *реализацией*, *значениями* и *переменными* вообще и *отношениями* и *переменными-отношениями* в частности. Обсуждение различий между моделью и реализацией привело нас к принципу *физической независимости от данных*.
- Я высказал утверждение о том, что SQL и реляционная модель – не одно и то же. Несколько различий между ними мы уже видели, например: SQL допускает null-значения и строки-дубликаты; столбцы в таблицах SQL упорядочены слева направо; в SQL не проводится четкое разграничение между табличными значениями и табличными переменными (поскольку для того и другого употребляется один и тот же термин, *таблица*). А ниже мы встретимся и со многими другими различиями. Все эти вопросы будут подробно рассматриваться в последующих главах.

И последнее замечание (явно я не говорил об этом раньше, но надеюсь, что оно очевидно из всего вышесказанного): реляционная модель по природе своей декларативная, а не процедурная, то есть всюду, где возможно, она отдает предпочтение декларативным решениям. Причина понятна: декларативность означает, что работу выполняет система, а процедурность – что работа возлагается на пользователя (так что, помимо всего прочего, речь идет и о продуктивности). Вот почему реляционная модель поддерживает декларативные запросы, декларативные обновления, декларативные определения представлений, декларативные ограничения целостности и т. д.

Примечание

Уже после того как предыдущий абзац был написан, мне сообщили, что по меньшей мере в одном широко известном SQL-продукте слово «декларативный» употребляется в том смысле, что система *не выполняет* предписанной работы! Иными словами, пользователю разрешается записать некое деклара-

тивное предписание (например, указать, что представление имеет ключ), но система ничего не делает для проверки этого ограничения, а лишь предполагает, что его будет проверять пользователь. Такая терминологическая неразбериха не способствует правильному пониманию. *Caveat lector*[1].

Упражнения

Упражнение 1.1. (*Уже приводилось в тексте главы в слегка измененной формулировке.*) Если вы еще этого не сделали, прочитайте главу с начала и найдите все места, где я употребил термин «отношение», хотя должен был бы написать *переменная-отношение*.

Упражнение 1.2. Кто такой Э. Ф. Кодд?

Упражнение 1.3. Что такое домен?

Упражнение 1.4. Что вы понимаете под *ссылочной целостностью*?

Упражнение 1.5. Термины *заголовок*, *тело*, *атрибут*, *кортеж*, *кардинальность* и *степень*, определенные в тексте главы для отношений-значений, можно очевидным образом интерпретировать и для переменных-отношений. Убедитесь, что это замечание вам понятно.

Упражнение 1.6. Опишите различие между двумя смыслами термина *модель данных*.

Упражнение 1.7. Объясните своими словами, (а) что такое физическая независимость от данных; (б) в чем различие между моделью и реализацией.

Упражнение 1.8. В тексте главы я написал, что таблицы, подобные приведенным на рис. 1.1 и 1.3, являются не отношениями, а лишь изображениями отношений. Назовите некоторые различия между такими рисунками и соответствующими отношениями.

Упражнение 1.9. (*Попытайтесь выполнить это упражнение, не заглядывая в текст главы.*) Какие переменные-отношения содержит база данных о поставщиках и деталях? Какие у них имеются атрибуты? Какие потенциальные и внешние ключи? (Смысл этого упражнения в том, чтобы вы как можно лучше освоились со структурой сквозного примера. Помнить фактические данные не так важно, хотя это не повредило бы.)

Упражнение 1.10. «Существует только одна реляционная модель». Прокомментируйте это утверждение.

Упражнение 1.11. Следующий отрывок взят из недавно вышедшего учебника по базам данных: «Важно различать хранимые отношения, то есть *таблицы*, и виртуальные отношения, то есть *представления*... Мы будем использовать термин *отношение* только там, где

[1] Читатель предупрежден (*лат.*). – *Прим. перев.*

можно было бы подставить слово «таблица» или «представление». Если мы захотим подчеркнуть, что речь идет о хранимом отношении, а не о представлении, то иногда будем употреблять термин *базовое отношение*, или *базовая таблица*. В этом тексте есть несколько неправильных интерпретаций реляционной модели. Назовите все, что сумеете обнаружить.

Упражнение 1.12. Следующий отрывок взят из еще одной недавно вышедшей книги по базам данных: «[Реляционная] модель... определяет простые таблицы для каждого отношения и для связей многие-ко-многим. Таблицы связываются между собой с помощью перекрестных ключей, описывающих связи между сущностями. Первичные и вторичные индексы обеспечивают быстрый доступ к данным на основе заданных параметров». Этот текст подается как *определение* реляционной модели... Что в нем не так?

Упражнение 1.13. Напишите предложения CREATE TABLE для SQL-версии базы данных о поставщиках и деталях.

Упражнение 1.14. Ниже приведено типичное SQL-предложение INSERT для базы данных о поставщиках и деталях:

```
INSERT INTO SP ( SNO , PNO , QTY ) VALUES ( 'S5' , 'P6' , 250 ) ;
```

Напишите эквивалентную операцию реляционного присваивания. *Примечание*: я понимаю, что еще не объяснял подробно синтаксис реляционного присваивания, поэтому ответ не обязательно должен быть синтаксически корректным – сделайте, как сможете.

Упражнение 1.15 (*более трудное*). Ниже приведено типичное SQL-предложение UPDATE для базы данных о поставщиках и деталях:

```
UPDATE S SET STATUS = 25 WHERE CITY = 'Paris' ;
```

Напишите эквивалентную операцию реляционного присваивания. (Цель этого упражнения – заставить вас задуматься о том, что необходимо для решения поставленной задачи. В этой главе рассказано недостаточно для формулировки полного ответа. Дополнительную информацию см. в главе 7.)

Упражнение 1.16. В тексте главы я сказал, что SQL напрямую не поддерживает реляционное присваивание. А косвенно? Если да, то как? Попутный вопрос: все ли реляционные присваивания можно выразить в терминах предложений INSERT / DELETE / UPDATE? Если нет, то почему? Что отсюда следует?

Упражнение 1.17. С *практической* точки зрения, почему вы думаете, что кортежи-дубликаты, упорядочение кортежей сверху вниз и упорядочение атрибутов слева направо – крайне неудачные идеи? (На эти вопросы намеренно не были даны ответы в тексте главы, а само это упражнение может послужить неплохой основой для коллективной дискуссии. Мы еще вернемся к этим материям ниже в этой книге.)

2

Типы и домены

Эта глава лишь косвенно связана с основной темой книги. Разумеется, типы – фундаментальное понятие, и обсуждаемые в этой главе идеи, безусловно, важны (заодно они помогут развеять некоторые широко распространенные заблуждения). Но теория типов как таковая не относится к реляционной тематике, и связанная с типами проблематика, по крайней мере в первом приближении, имеет мало общего с повседневной практикой применения SQL. Более того, хотя в SQL несомненно имеются проблемы в этой области, вы, как правило, тут ничего поделать не можете; я хочу сказать, что почти никаких конкретных рекомендаций по реляционному использованию SQL в этой части я предложить не могу (впрочем, как вы увидите, кое-что все же нашлось). Поэтому при первом чтении вы можете просто пробежать эту главу глазами и вернуться к ней после усвоения материала из последующих глав.

Типы и отношения

Типы данных (для краткости просто типы) – фундаментальное понятие информатики. Для реляционной теории поддержка со стороны теории типов особенно важна, так как отношения определены над типами, то есть для любого атрибута отношения определен некоторый тип (и то же самое, разумеется, справедливо для переменных-отношений). Например, для атрибута STATUS переменной-отношения S (поставщики) можно было бы задать тип INTEGER. В таком случае каждое отношение, которое потенциально может быть значением переменной-отношения S, должно иметь атрибут STATUS типа INTEGER, а это, в свою очередь, означает, что в каждом кортеже такого отношения должен присутствовать атрибут STATUS типа INTEGER, то есть значением атрибута STATUS такого кортежа будет целое число.

Все эти вопросы я рассмотрю более подробно далее в этой главе. А пока скажу лишь, что (с некоторыми важными исключениями, о которых

речь пойдет ниже) атрибут отношения может иметь любой тип, в том числе произвольно сложный. В частности, типы могут быть определены как системой, так и пользователем. Однако в этой книге я не собираюсь много распространяться на тему определенных пользователем типов, потому что:

- Весь смысл определенных пользователем типов (во всяком случае с точки зрения человека, который ими просто пользуется, в отличие от человека, который их определяет) заключается в том, что они должны вести себя как системные.
- Не так уж много есть разработчиков, которым когда-нибудь придется столкнуться с задачей определения нового типа, да и все равно создание типа не сопряжено с какими-то специфически реляционными проблемами.

Поэтому, начиная с этого места, вы можете считать, что термин *тип* означает системный тип, если из контекста не следует противное. Реляционная модель предписывает только один такой тип – BOOLEAN (наиболее фундаментальный). Тип BOOLEAN может принимать ровно два значения, точнее, значения истинности, которые обозначаются литералами TRUE и FALSE. Разумеется, любая реальная система поддерживает много других типов, и для определенности я буду считать, что определены типы INTEGER (целые числа), FIXED (числа с фиксированной запятой) и CHAR (строки символов произвольной длины). *Примечание*: В разделе «Скалярные типы в SQL» ниже я рассмотрю системные типы, поддерживаемые в языке SQL.

Ради исторической точности я должен объяснить, что в исходном определении реляционной модели Кодд говорил, что отношения определены над *доменами*, а не над типами. Но на самом деле домены и типы – в точности одно и то же. Если хотите, можете считать это заявление отражением моей личной позиции, но я хотел бы привести ряд аргументов в ее защиту. Начну с реляционной модели в том виде, в котором ее определил Кодд, то есть буду временно использовать термин *домен*, а не *тип*. Я хочу рассмотреть два важных вопроса, каждому из которых отведу отдельный раздел:

Сравнения на равенство и «подавление проверки доменов»

В этой части обсуждения я надеюсь убедить вас, что домены – это типы.

Атомарность значений данных и первая нормальная форма

А в этой части я рассчитываю доказать, что типы могут быть сколь угодно сложными.

Сравнения на равенство

Вопреки сказанному выше об игнорировании определенных пользователем типов, в этом разделе я буду предполагать, что атрибут «номер по-

ставщика» (SNO) в переменных-отношениях S и SP объявлен как имеющий некоторый пользовательский тип – прошу прощения, домен, – который для простоты назовем также SNO. (*Примечание*: В этой книге я рассматриваю слова *объявление* и *определение* как синонимы.) Аналогично, предположим, что атрибут «номер детали» (PNO) в переменных-отношениях P и SP также имеет некоторый пользовательский тип (или домен), PNO. Отмечу, что эти предположения не слишком существенны для моего рассуждения; мне просто кажется, что они придают ему убедительности и, возможно, делают несколько понятнее.

Начну с того, что, как всем известно (?), в реляционной модели два значения можно сравнивать только, если они принадлежат одному и тому же домену. Например, следующее сравнение (которое могло бы встретиться во фразе WHERE SQL-запроса), очевидно, допустимо:

```
SP.SNO = S.SNO          /* правильно   */
```

Напротив, это сравнение, очевидно (?), недопустимо:

```
SP.PNO = S.SNO          /* неправильно */
```

Почему недопустимо? Потому что номера деталей и номера поставщиков – совершенно разные вещи, они берутся из разных доменов. Таким образом, общая идея состоит в том, что СУБД[1] должна отвергать любую попытку выполнить реляционную операцию (соединение, объединение и т. п.), в которой явно или неявно присутствует сравнение на равенство значений из разных доменов. Вот, например, SQL-запрос, в котором пользователь пытается найти поставщиков, не поставляющих никаких деталей:

```
SELECT S.SNO , S.SNAME , S.STATUS , S.CITY
FROM   S
WHERE  NOT EXISTS
     ( SELECT *
       FROM   SP
       WHERE  SP.PNO = S.SNO )      /* неправильно */
```

(В конце нет точки с запятой, потому что это выражение, а не предложение. См. упражнение 2.23 в конце главы.)

В комментарии говорится, что этот запрос неправилен. Причина в том, что в последней строке пользователь, вероятно, хотел написать WHERE SP.SNO = S.SNO, но по ошибке – может быть, просто попал не по той клавише – написал WHERE SP.*PNO* = S.SNO. А поскольку мы говорим о простой опечатке (по всей видимости), то со стороны СУБД было бы

[1] СУБД = система управления базами данных. Кстати, в чем разница между СУБД и базой данных? (Это не праздный вопрос, поскольку очень часто говорят *база данных*, имея в виду либо конкретную СУБД, например Oracle, либо экземпляр такой СУБД, установленный на конкретном компьютере. Проблема в том, что если называть СУБД базой данных, то что же такое база данных?)

любезно в этот момент прервать пользователя, подсветить место, где он ошибся, и, быть может, спросить, хочет ли пользователь исправить ошибку, прежде чем продолжить.

Но я не знаю ни одной SQL-системы, которая вела бы себя подобным образом; современные продукты в зависимости от того, как настроена база данных, либо просто выдают ошибку, либо возвращают неправильный ответ. Ну не то чтобы неправильный, скорее, правильный ответ на неверно заданный вопрос. (Вам от этого легче?)

Еще раз повторю: по идее СУБД должна отвергать сравнение, подобное SP.PNO = S.SNO, если оно недопустимо. Однако Кодд полагал, что в такой ситуации у пользователя должна быть возможность заставить СУБД продолжить работу и выполнить сравнение, даже если оно представляется недопустимым, считая, что иногда пользователь знает больше, чем СУБД. Мне такое обоснование принять трудно, потому что, честно говоря, я не вижу в нем смысла, но я все-таки попытаюсь. Предположим, что вам поручено спроектировать базу данных, в которой есть, к примеру, заказчики и поставщики, и вы решили определить домен номеров заказчиков и домен номеров поставщиков. База данных спроектирована, загружена, и все работает прекрасно год-другой. А затем какой-то пользователь приходит с запросом, о котором вы раньше даже не задумывались: «Есть ли у нас заказчики, которые одновременно являются нашими поставщиками?» Отметим, что запрос вполне осмысленный. Отметим также, что он подразумевает сравнение на равенство номера заказчика с номером поставщика (кросс-доменное сравнение). А коль скоро это так, система, конечно, не должна запрещать подобное сравнение; разумеется, система не вправе отказать в выполнении осмысленного запроса.

Исходя из таких соображений, Кодд предложил варианты некоторых реляционных операторов с «подавлением проверки доменов» (domain check override – DCO). Например, вариант соединения с подавлением проверки доменов должен выполнять соединение, даже если соединяемые атрибуты определены над разными доменами. В терминах SQL можно было бы реализовать это предложение, введя новую фразу IGNORE DOMAIN CHECKS, которая включалась бы в SQL-запрос, например, следующим образом:

```
SELECT ...
FROM   ...
WHERE  CUSTNO = SNO
IGNORE DOMAIN CHECKS
```

Причем эта фраза должна была бы подвергаться независимой авторизации – большинству пользователей было бы запрещено ее использовать (быть может, разрешение предоставлялось бы только администратору).

Прежде чем переходить к детальному анализу подавления проверки доменов, я хочу привести более простой пример. Рассмотрим следующие два запроса к базе данных о поставщиках и деталях:

```
SELECT ...                    |    SELECT ...
FROM   P , SP                 |    FROM   P , SP
WHERE  P.WEIGHT = SP.QTY      |    WHERE  P.WEIGHT - SP.QTY = 0
```

В предположении (достаточно разумном), что вес и количество определены над разными доменами, запрос слева, очевидно, недопустим. Но что сказать о запросе справа? Согласно Кодду, он является допустимым! В своей книге «The Relational Model for Database Management Version 2» (Addison-Wesley, 1990), на странице 47 он говорит, что в такой ситуации «СУБД [просто] проверяет, что базовые типы данных одинаковы»; в рассматриваемом случае «базовые типы данных» – это числа (говоря нестрого), поэтому проверка проходит.

Мне такой вывод представляется неприемлемым. Очевидно, что выражения P.WEIGHT = SP.QTY и P.WEIGHT – SP.QTY = 0 означают по существу одно и то же. Поэтому они должны быть допустимы или недопустимы одновременно; мысль о том, что одно может быть допустимо, а другое – нет, просто не имеет смысла. Поэтому мне кажется, что в проверке доменов по Кодду изначально есть что-то странное, даже если забыть о ее подавлении. (На самом деле, проверка доменов по Кодду применяется только в очень частном случае, когда оба сравниваемых операнда определены как ссылки на простые атрибуты. Отметим, что сравнение P.WEIGHT = SP.QTY попадает в эту категорию, а P.WEIGHT – SP.QTY = 0 – нет.)

Рассмотрим еще более простой пример. Взгляните на следующие сравнения (то и другое могли бы встретиться во фразе WHERE предложения SQL):

```
S.SNO = 'X4'          P.PNO = 'X4'          S.SNO = P.PNO
```

Надеюсь, вы согласитесь, что по крайней мере первые два выглядят вполне допустимыми (и выполнятся успешно и даже вернут TRUE), тогда как третье – нет. Но если так, то я полагаю, что вы согласитесь с тем, что происходит нечто странное: получается, что у нас есть три значения *a*, *b* и *c* такие, что *a* = *c* истинно, *b* = *c* истинно, но что касается *a* = *b*, так мы даже не можем выполнить сравнение, что уж говорить о его истинности! Так в чем же дело?

А теперь я вернусь к тому факту, что атрибуты S.SNO и P.PNO определены соответственно над доменами SNO и PNO, и к своему утверждению о том, что домены на самом деле являются типами; фактически я уже предположил ранее, что конкретные домены SNO и PNO – это определенные пользователем типы. Очень может быть, что оба эти типа физически представлены в терминах системного типа CHAR, но, как мы видели в главе 1, такие представления являются частью реализации, а не модели; для пользователя это несущественно и даже скрыто от него (по крайней мере, так должно быть)[1]. В частности, операторы, которые

[1] В этой книге под словом *представление* (representation) я понимаю физическое представление, если контекст не подразумевает иной семантики.

применяются к номерам поставщиков и к номерам деталей, – это операторы, входящие в определения соответствующих типов, а не те, что определены для типа CHAR (см. раздел «Что такое тип?» ниже в этой главе). Например, мы можем конкатенировать две символьные строки, но не два номера поставщиков (это можно было бы сделать, если бы для типа SNO был определен оператор конкатенации).

При определении типа мы должны также определить операторы, которые можно применять к значениям и переменным данного типа (снова отсылаю вас к разделу «Что такое тип?»). И одним из подлежащих определению операторов является так называемый *селектор*, который позволяет выбрать, или задать, произвольное значение определяемого типа[1]. Например, в случае типа SNO селектор (который на практике, скорее всего, назывался бы тоже SNO) позволяет выбрать конкретное значение типа SNO, для которого задано представление в виде CHAR. Например:

```
SNO('S1')
```

Это выражение представляет собой селектор SNO и возвращает некоторый номер поставщика (точнее, тот, что представлен символьной строкой 'S1'). Аналогично выражение:

```
PNO('P1')
```

представляет собой селектор PNO и возвращает некоторый номер детали (точнее, тот, что представлен символьной строкой 'P1'). Иными словами, селекторы SNO и PNO принимают значение типа CHAR и преобразуют его в значение типа SNO или PNO, соответственно.

Вернемся к сравнению S.SNO = 'X4'. Как видите, сравниваемые величины имеют разные типы (если быть точным, то SNO и CHAR). Так как типы различны, то величины не могут быть равны (напомним, что два значения можно сравнить на равенство, «только если они принадлежат одному домену»). Но системе хотя бы известно, что существует оператор, а именно селектор SNO, который преобразует тип CHAR в тип SNO. Поэтому она может неявно вызвать этот оператор для преобразования операнда типа CHAR в номер поставщика, заменив тем самым исходное сравнение таким:

```
S.SNO = SNO('X4')
```

Теперь мы сравниваем два номера поставщика, что вполне законно.

Точно так же система может заменить сравнение P.PNO = 'X4' таким:

---

[1] Это наблюдение остается справедливым вне зависимости от того, говорим мы об SQL (как сейчас) или о чем-то другом, но следует четко понимать, что селекторы в SQL не так просты, как могло бы показаться, и что термина *селектор* как такового в SQL вообще нет. Кроме того, я должен уточнить, что селекторы не имеют ничего общего с SQL-предложением SELECT.

```
P.PNO = PNO('X4')
```

Но для сравнения S.SNO = P.PNO системе неизвестен (по крайней мере, мы так предполагаем) оператор, который смог бы преобразовать номер поставщика в номер детали или наоборот, поэтому сравнение завершается *ошибкой типизации*: сравниваемые величины имеют разные типы и не существует способа привести их к одному типу.

Примечание

Неявное преобразование типов в литературе часто называется *приведением* (*coercion*). Таким образом, в первом примере символьная строка 'X4' приводится к типу SNO, а во втором – к типу PNO. Я еще вернусь к теме приведения типов в SQL в разделе «Проверка и приведение типов в SQL» ниже.

Продолжим рассмотрение примера. При определении пользовательского типа, такого как SNO или PNO, мы должны определить специальный оператор THE_, который преобразует значение типа SNO или PNO в строку символов (или иную форму), служащую для его представления[1]. Предположим, что в данном примере операторы THE_ для типов SNO и PNO называются THE_SC и THE_PC соответственно. Тогда у желания сравнить S.SNO и P.PNO на равенство, на мой взгляд, есть единственная разумная интерпретация – нас интересует, совпадают ли строковые представления значений. Это можно было бы проверить следующим образом:

```
THE_SC ( S.SNO ) = THE_PC ( P.PNO )
```

Другими словами: преобразовать номер поставщика в строку, преобразовать номер детали в строку и сравнить обе строки.

Уверен, вы понимаете, что набросанный только что механизм, в котором участвуют селекторы и операторы THE_, эффективно решает обе проблемы: (а) проверка доменов и (б) подавление этой проверки в случае необходимости. К тому же он делает это элегантным, полностью ортогональным, а не придуманным для частного случая способом. Напротив, механизм подавления проверки доменов на самом деле эту задачу не решает; фактически он вообще не имеет смысла, так как смешивает в одну кучу типы и представления (как уже отмечалось, типы – это понятие уровня модели, а представления – понятие уровня реализации).

Вы, вероятно, уже поняли, что я все это время говорил о том, что в языках программирования называется *строгой типизацией*. Разные авто-

[1] Опять-таки это наблюдение справедливо вне зависимости от того, идет ли речь об SQL или ином контексте, хотя (как и в случае селекторов) «операторы THE_» в SQL не так просты, как кажется, а сам термин «*оператор THE_*» в SQL отсутствует. Замечу также, что для данного типа может быть определено несколько операторов THE_; пример такого рода см. в обсуждении типа POINT в разделе «Что такое тип?».

ры дают слегка отличающиеся определения этого термина, но по существу он означает, что (а) все – в том числе все значения и все переменные – имеет тип и (б) при попытке выполнить любую операцию система проверяет, что операнды имеют подходящие для этой операции типы (или приводятся к подходящим типам). Отметим также, что этот механизм применим ко всем операциям, а не только к сравнению на равенство, которое мы обсуждали; упор именно на операции сравнения в обсуждениях проверки доменов, встречающихся в литературе, обусловлен чисто историческими причинами и, честно говоря, неуместен. Рассмотрим, к примеру, следующие выражения:

```
P.WEIGHT * SP.QTY

P.WEIGHT + SP.QTY
```

Первое, по-видимому, допустимо (в результате получается вес, точнее, полный вес одной поставки). Напротив, второе, скорее всего, недопустимо (как можно складывать вес с количеством?).

Я хотел бы завершить этот раздел, подчеркнув поистине фундаментальную роль оператора сравнения на равенство («=»). Совсем не случайно наше обсуждение сфокусировалось на сравнении двух значений на равенство. Дело в том, что равенство является важнейшей концепцией, и реляционная модель требует, чтобы она была поддержана во всех типах. Действительно, коль скоро тип по существу представляет собой множество значений (см. раздел «Что такое тип?»), то без оператора «=» мы даже не смогли бы сказать, какие значения составляют тип! Иными словами, если дан тип *T* и значение *v*, то без этого оператора мы не могли бы сказать, совпадает ли *v* с каким-то элементом множества значений, составляющих тип *T*.

Более того, реляционная модель также определяет семантику оператора «=» следующим образом: если *v1* и *v2* – значения одного типа, то выражение *v1* = *v2* равно TRUE, если *v1* и *v2* – одно и то же значение, и FALSE – в противном случае. (Напротив, если *v1* и *v2* – значения разных типов, то выражение *v1* = *v2* не имеет смысла – даже не является допустимым сравнением, – если только не существует приведения *v1* к типу *v2* или наоборот, а в таком случае речь уже не идет о сравнении *v1* и *v2*.)

Атомарность значений данных

Надеюсь, что в предыдущем разделе мне удалось убедить вас, что домены – это в действительности типы, не более и не менее. А теперь я хочу обратиться к вопросу об атомарности значений данных и связанному с ним понятию первой нормальной формы (для краткости 1НФ). В главе 1 я сказал, что 1НФ означает, что каждый кортеж любого отношения содержит единственное значение (соответствующего типа) в позиции каждого атрибута, – и обычно к этому добавляют, что «единствен-

ное значение» предполагается атомарным. Но сразу же возникает вопрос: а что такое атомарные данные?

На странице 6 уже упоминавшейся книги («The Relational Model for Database Management Version 2») Кодд определяет атомарные данные как такие данные, «которые СУБД не может разложить на более мелкие составляющие (исключая некоторые специальные функции)». Даже если не обращать внимания на примечание в скобках, это определение выглядит несколько загадочным и/или не слишком точным. Например, что сказать о строках символов? Они атомарны? Любой известный мне продукт предоставляет многочисленные операторы – LIKE, SUBSTR (подстрока), «||» (конкатенация) и прочие, – которые исходят из того, что для строк символов СУБД может выполнить разложение на более простые составляющие. Так атомарны строки символов или нет? Как вы думаете?

Приведу еще несколько примеров значений, атомарность которых проблематична, но которые мы тем не менее хотели бы видеть в качестве значений атрибутов отношений:

- Целые числа, которые разлагаются в произведение простых множителей (я понимаю, что это не то разложение, которое мы обычно подразумеваем в данном контексте, но просто хочу показать, что самое понятие разложимости, или декомпозиции, открыто для интерпретации).
- Числа с фиксированной запятой, которые можно разложить на целую и дробную части.
- Дата и время, которые можно представить в виде композиции год/месяц/день или час/минута/секунда соответственно.

И так далее.

А теперь я хочу предложить более удивительный пример. Взгляните на рис. 2.1. Отношение R1 на этом рисунке – это редуцированная версия отношения «поставки» из нашего сквозного примера; оно показывает, что какие-то поставщики поставляют какие-то детали, и содержит по одному кортежу для каждой допустимой комбинации SNO/PNO. Давайте пока согласимся, что номера поставщиков и номера деталей действительно «атомарны», тогда можно признать, что отношение R1 находится по меньшей мере в первой нормальной форме.

Допустим теперь, что мы заменили R1 отношением R2, в котором показано, что какие-то поставщики поставляют некоторые *группы* деталей (атрибут PNO в R2, по выражению некоторых авторов, *многозначный*, и значениями этого атрибута являются группы номеров деталей). В этом случае большинство людей наверняка сказали бы, что R2 не приведена к 1НФ; фактически это выглядит как пример «повторяющихся групп», а повторяющиеся группы – это то, что, по мнению большинства, 1НФ как раз и призвана запретить (поскольку такие группы уж точно не атомарны, правильно?).

R1

SNO	PNO
S2	P1
S2	P2
S3	P2
S4	P2
S4	P4
S4	P5

R2

SNO	PNO
S2	P1, P2
S3	P2
S4	P2, P4, P5

R3

SNO	PNO_SET
S2	{P1, P2}
S3	{P2}
S4	{P2, P4, P5}

Рис. 2.1. *Отношения R1, R2 и R3*

Хорошо, согласимся пока, что R2 не приведена к 1НФ. Но давайте теперь заменим R2 на R3. Я утверждаю, что отношение *R3 находится в 1НФ*! (Я не утверждаю, что это отношение удачно спроектировано, – скорее, нет, – но не в этом дело. Сейчас меня интересует, что считать допустимым, а не правильные подходы к проектированию. Структура R3 допустима.) В поддержку своего мнения приведу следующие аргументы:

- Во-первых, отмечу, что я переименовал атрибут в PNO_SET и показал, что группы номеров деталей являются значениями типа PNO_SET, заключив их в фигурные скобки, чтобы подчеркнуть тот факт, что каждая такая группа на самом деле представляет собой единственное значение: значение-множество, если быть точным, но множество на определенном уровне абстракции тоже можно считать единственным значением.
- Во-вторых (и независимо от того, что вы думаете о моем первом аргументе), *множество, подобное {P2,P4,P5}, разложимо СУБД не более и не менее, чем строка символов*, – и это неоспоримый факт. Как и строки символов, множества имеют внутреннюю структуру, но для определенных целей эту структуру удобно игнорировать. Другими словами, если строки символов совместимы с требованиями 1НФ, то есть являются атомарными, значит, то же самое должно быть справедливо в отношении множеств.

А веду я к тому, что понятие атомарности *вообще не имеет смысла*; оно зависит от того, что мы собираемся делать с данными. Иногда мы хотим обращаться с множеством номеров деталей, как с единым целым, а иногда рассматривать отдельные номера деталей из этого множества – но в этом случае мы опускаемся на более низкий уровень детализации, или абстракции. Возможно, поможет такая аналогия. В физике (откуда, собственно, и пришло понятие атомарности) прослеживается точная параллель: иногда мы хотим рассматривать отдельные атомы как неделимые сущности, а иногда рассуждаем об элементарных частицах (протонах, нейтронах и электронах), из которых состоит атом. Более того, по крайней мере протоны и нейтроны сами не являются неделимыми, они состоят из «субэлементарных» частиц, называемых кварками. И возможно, на этом дело не кончается.

Вернемся ненадолго к отношению R3. На рис. 2.1 я показал значения PNO_SET в виде множеств общего вида. Но на практике было бы полезнее, если бы это были отношения (см. рис. 2.2, где я изменил имя атрибута на PNO_REL). Почему это было бы полезнее? Потому что именно отношения, а не множества вообще, – предмет реляционной теории[1]. Следовательно, к отношениям применим весь аппарат реляционной алгебры – их можно ограничивать, проецировать, соединять и т. д. А если бы мы использовали множества общего вида, то должны были бы вводить новые операторы (объединение множеств, пересечение множеств и прочие) для работы с ними... Куда лучше извлечь максимум пользы из тех операторов, что уже есть!

Атрибут PNO_REL на рис. 2.2 – пример атрибута, *значением которого является отношение* (relation valued attribute – RVA). Я еще буду говорить об RVA-атрибутах в главе 7, а пока отмечу лишь, что SQL их не поддерживает. (Точнее, не поддерживается то, что можно было бы назвать аналогом RVA-атрибутов, – *столбцы с табличными значениями*. Как ни странно, SQL все же поддерживает столбцы со значениями-массивами, столбцы со значениями-строками и даже столбцы, значениями которых являются «мультимножества строк», где под *мультимножеством* понимается множество, допускающее дубликаты. Столбцы, значениями которых являются мультимножества строк, немного напоминают «столбцы с табличными значениями», но на самом деле таковыми не являются, так как к содержащимся в них значениям нельзя применять стандартные SQL-операторы для работы с таблицами и потому они, по определению, не могут считаться табличными значениями в смысле SQL.

Я намеренно выбрал предыдущий – шокирующий – пример. В конце концов, отношения с RVA-атрибутами действительно выглядят, как отношения с повторяющимися группами, а вы наверняка слышали, что повторяющиеся группы в реляционном мире находятся под строгим запретом. Но я бы мог проиллюстрировать свою точку зрения на ряде других примеров; я мог бы показать атрибуты (и, следовательно, домены), которые содержат массивы, мультимножества, списки, фотографии, аудио- или видеозаписи, рентгеновские лучи, отпечатки пальцев, XML-документы – что угодно, «атомарное» или «неатомарное». Атрибуты и, стало быть, домены, могут содержать *произвольные значения*.

Кстати говоря, предыдущий абзац – прямой путь к объяснению того, почему настоящая «объектно-реляционная» система – не что иное, как

[1] Если вы недоумеваете, скажу, что разница заключается в том, что множество общего вида может содержать все, что угодно, тогда как отношение содержит кортежи. Отмечу, однако, что отношение, конечно, аналогично множеству общего вида в том смысле, что тоже может рассматриваться как единственное значение.

настоящая реляционная система, то есть система, которая поддерживает реляционную модель и все вытекающие из нее требования. Ведь смысл «объектно-реляционной» системы в том и заключается, что значения атрибутов отношения могут быть сколь угодно сложными. Быть может, лучше выразить эту мысль по-другому: истинная объектно-реляционная система – это просто реляционная система с надлежащей поддержкой типов (в частности, типов, определенных пользователем), а это как раз и есть истинная реляционная система, не больше, не меньше. И то, что некоторые любят называть «объектно-реляционной моделью» – это просто реляционная модель, ни больше, ни меньше.

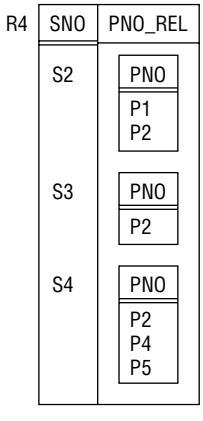


Рис. 2.2. *Отношение R4 (пересмотренный вариант отношения R3)*

Что такое тип?

Начиная с этого места, я буду употреблять термин *тип* вместо *домен*. Так что же такое тип? По сути дела, это *именованное конечное множество значений*[1] – всех значений данного типа, например: всех возможных целых чисел, всех возможных строк, всех возможных номеров поставщиков, всех возможных XML-документов, всех возможных отношений с некоторым заголовком и т. д. Кроме того:

- Любое *значение* принадлежит некоторому типу – на самом деле, в точности одному типу, если не рассматривать поддержку наследования, о чем мы в этой книге говорить не будем. *Примечание*: Из этого определения следует, что типы дизъюнктны (не перекрываются). Однако на этом моменте, пожалуй, имеет смысл остановиться более подробно. Как сказал один из рецензентов, типы *ТеплокровноеЖивотное* и *ЧетырехногоеЖивотное*, безусловно, перекрываются, разве нет? Конеч-

[1] Конечное, поскольку мы имеем дело с компьютерами, которые конечны по определению. Обратите также внимание на прилагательное *именованный*: типы с разными именами – это разные типы.

но, перекрываются, но я хочу сказать, что, допуская перекрытие типов, мы по различным причинам вступаем в область наследования типов, фактически даже в область *множественного* наследования. Поскольку эти причины, да и вообще вся тематика наследования, не зависят от рассматриваемого нами контекста, реляционного или любого другого, я не собираюсь останавливаться на них в этой книге.

- Любая *переменная*, любой *атрибут*, любой *оператор*, возвращающий результат, и любой *параметр* любого оператора объявляется как принадлежащий некоторому типу. И если, например, объявлено, что переменная *V* имеет тип *T*, то это в точности означает, что любое значение *v*, которое допустимо присвоить *V*, само должно принадлежать типу *T*.
- Всякое *выражение* обозначает некоторое значение и потому имеет тип, а именно тип своего значения, то есть тип того значения, которое возвращает самый внешний оператор в выражении (под «самым внешним» я понимаю оператор, который выполняется последним). Например, типом выражения

```
( a / b ) + ( x - y )
```

будет объявленный тип оператора «+», каким бы он ни оказался.

Тот факт, что и параметры имеют тип, подводит нас к вопросу, о котором я уже упоминал, но пока не обсуждал подробно, а именно, что *с каждым типом ассоциирован набор операторов, применяемых к значениям и переменным данного типа*. Здесь слова «оператор *Op* ассоциирован с типом *T*» в точности означают, что у оператора *Op* имеется параметр типа *T*.[1] Например, для целых чисел определены обычные арифметические операторы; для даты и времени имеются специальные календарные арифметические операторы; для XML-документов определены так называемые XPath-операторы; для отношений – операторы реляционной алгебры. И для любого типа определены операторы присваивания («:=») и сравнения на равенство («=»). Таким образом, любая система, обеспечивающая надлежащую поддержку типов, – а «надлежащая поддержка», конечно, предполагает и возможность определения типов пользователями – должна допускать и определение пользователями собственных операторов, поскольку типы без операторов бесполезны. *Примеча-*

[1] Логическое различие между типом и представлением здесь особенно важно. Подчеркну еще раз: оператор, ассоциированный с типом *T*, ассоциирован именно с типом *T*, а *не* с представлением типа *T*. Например, из того, что представлением типа SNO является CHAR, вовсе не следует, что мы можем конкатенировать два номера поставщиков; это возможно лишь в случае, когда оператор конкатенации («||») определен для типа SNO. (Если помните, я уже мимоходом упоминал именно этот пример в разделе «Сравнение на равенство».)

ние: Пользовательские операторы можно ассоциировать и с системными типами, а не только с определенными пользователем.

Отметим, что, по определению, к значениям и переменным данного типа можно применять только операторы, ассоциированные с этим типом. Рассмотрим, к примеру, системный тип INTEGER:

- Система предоставляет оператор присваивания «:=», который позволяет присвоить целочисленное значение целочисленной переменной.
- Она также предоставляет операторы сравнения «=», «≠», «<» и т. д. для сравнения целочисленных значений.
- Она также предоставляет арифметические операторы «+», «*» и прочие для выполнения арифметических операций над целочисленными значениями.
- Она не предоставляет операторов «‖» (конкатенация), SUBSTR (подстрока) и других для выполнения строковых операций над целыми числами; иначе говоря, строковые операции над целочисленными значениями не поддерживаются.

Что касается определенного пользователем типа SNO, то мы, конечно, определили бы для него операторы присваивания и сравнения («:=», «=», «≠», возможно, «<» и другие), но, скорее всего, не стали бы определять операторы «+», «*» и им подобные, то есть арифметические операции над номерами поставщиков не поддерживались бы. (В самом деле, что понимать под сложением или умножением двух номеров поставщиков?)

Из всего вышесказанного должно быть понятно, что для определения нового типа необходимо сделать по меньшей мере следующее:

1. Задать имя типа (это очевидно).
2. Определить значения, составляющие тип. Подробнее я буду говорить об этом в главе 8.
3. Задать скрытое физическое представление значений типа. Как уже отмечалось выше, это деталь реализации, а не модели, поэтому больше мы к этому вопросу возвращаться не будем.
4. Определить оператор-селектор для выбора, или специфицирования, значений типа.
5. Определить операторы – в частности, операторы THE_, – которые будут применяться к значениям и переменным данного типа (см. ниже).
6. Для операторов, возвращающих результат, определить тип результата (также см. ниже).

Отметим, что пункты 4, 5 и 6, взятые вместе, означают, что система точно знает, какие выражения допустимы, а для допустимых выражений знает также тип результата.

В качестве примера предположим, что имеется пользовательский тип POINT для представления геометрических точек на двумерной плоскости. Ниже приведено определение на языке **Tutorial D** – я мог бы воспользоваться и языком SQL, но с определением операторов в SQL сопряжено слишком много деталей, в которые я сейчас не хочу вдаваться, – оператора с именем REFLECT, который для точки *P* с декартовыми координатами *(x, y)* возвращает «отраженную» точку с координатами *(– x, –y)*:

```
1 OPERATOR REFLECT ( P POINT ) RETURNS POINT ;
2   RETURN POINT ( - THE_X ( P ) , - THE_Y ( P ) ) ;
3 END OPERATOR ;
```

Пояснение:

- В строке 1 говорится, что оператор называется REFLECT, принимает единственный параметр P типа POINT и возвращает результат тоже типа POINT (таким образом, объявленный тип этого оператора – POINT).
- В строке 2 содержится код, реализующий оператор. Это всего одно предложение RETURN. В качестве значения возвращается точка, для получения которой вызывается селектор POINT; при его вызове передается два аргумента, соответствующих координатам X и Y рассматриваемой точки. Для каждого аргумента производится обращение к оператору THE_; эти обращения дают координаты X и Y точки-аргумента, соответствующей параметру P, а взяв противоположные значения, мы получаем желаемый результат[1].
- Строка 3 завершает определение.

До сих пор мы в этом разделе ограничивались по большей части типами, определенными пользователем. Но аналогичные рассуждения применимы и к системным типам, только в этом случае определения предоставляются системой, а не пользователем. Например, если INTEGER – системный тип, то система задает его имя, определяет множество допустимых целых чисел, скрытое представление и – как мы уже видели – соответствующие операторы «:=», «=», «+» и т. д. (хотя пользователь, конечно, может определить и дополнительные операторы).

И последнее замечание. Я уже несколько раз упоминал операторы-селекторы, а вот о чем я не сказал, так это о том, что селекторы – точнее, вызовы селекторов – это в действительности обобщение всем известно-

[1] Кстати говоря, в этом абзаце мы затронули еще одно важное логическое различие: между аргументами и параметрами (см. упражнение 2.5 в конце главы). Отметим также, что селектор POINT, в отличие от встречавшихся ранее, принимает два аргумента (поскольку точка представляется двумя, а не одним значением).

го понятия *литерала*. Я имею в виду, что все литералы суть вызовы селекторов, хотя не все вызовы селекторов – литералы (на самом деле, вызов селектора является литералом тогда и только тогда, когда все его аргументы также заданы в виде литералов); например, и POINT(X,Y), и POINT(1.0,2.5) – вызовы селектора POINT, но только вторая точка POINT – литерал. Отсюда следует, что с любым типом необходимо ассоциировать не только селектор, но и формат для записи литералов. (И, быть может, для полноты стоит добавить, что для любого значения любого типа должно существовать обозначение посредством литерала.)

Скалярные и нескалярные типы

Обычно принято разделять типы на скалярные и нескалярные. Неформально говоря, тип является *скалярным*, если у него нет никаких видимых компонентов, и *нескалярным* – в противном случае. Значения, переменные, атрибуты, операторы, параметры и выражения типа *T* считаются скалярными или нескалярными в зависимости от того, каков сам тип *T*. Например:

- Тип INTEGER – скалярный; следовательно, значения, переменные и т. д. типа INTEGER также скалярные, то есть у них нет видимых компонентов.
- Типы кортежа и отношения – нескалярные; видимые пользователю компоненты – это соответствующие атрибуты; следовательно, значения, переменные и т. д. типа кортежа или отношения – нескалярные.

Но следует обязательно подчеркнуть, что это неформальные понятия. Действительно, мы уже видели, что понятие атомарности значения данных лишено всякого смысла, а «скалярность» – то же самое понятие, только называется по-другому. Поэтому реляционная модель ни в коем случае не полагается на какое-либо формальное различие между скалярами и нескалярами. Однако в этой книге я все же буду неформально различать их (поскольку нахожу это интуитивно полезным); точнее, я буду употреблять термин *скаляр* в применении к типам, которые не являются ни кортежами, ни отношениями, а термин *нескаляр* – в применении к типам, которые являются либо кортежами, либо отношениями.

Рассмотрим пример. Ниже приведено определение на языке **Tutorial D** базовой переменной-отношения S («поставщики»); заметьте, что для простоты я сейчас определяю все атрибуты как принадлежащие тем или иным системным типам:

```
1 VAR S BASE
2   RELATION { SNO CHAR , SNAME CHAR , STATUS INTEGER , CITY CHAR }
3   KEY { SNO } ;
```

Пояснение:

- Ключевое слово VAR в строке 1 означает, что это определение переменной с именем S, а ключевое слово BASE говорит, что эта переменная представляет собой базовую переменную-отношение.
- В строке 2 определяется тип переменной. Ключевое слово RELATION показывает, что это тип отношения; в оставшейся части задается набор атрибутов, составляющих заголовок отношения (напомню, что в главе 1 объяснялось, что атрибут описывается парой имя-атрибута/имя-типа). Этот тип, конечно, нескалярный. Порядок следования атрибутов несущественен.
- В строке 3 говорится, что {SNO} – потенциальный ключ этой переменной-отношения.

Этот пример иллюстрирует еще один момент, а именно, что тип:

```
RELATION { SNO CHAR , SNAME CHAR , STATUS INTEGER , CITY CHAR }
```

является образчиком *сгенерированного* типа. Сгенерированным называется тип, который получен в результате вызова некоторого *генератора типа* (в данном случае таким генератором типа является RELATION). Вы можете считать, что генератор типа – это особый вид оператора; особый – потому что (а) он возвращает тип вместо (к примеру) скалярного значения и (б) он вызывается на этапе компиляции, а не выполнения. Так, в большинстве языков программирования поддерживается генератор типа ARRAY, который позволяет пользователю определять разнообразные конкретные типы массивов. Но в этой книге нас будут интересовать только генераторы типа TUPLE и RELATION. Вот пример вызова генератора типа TUPLE:

```
VAR S_TUPLE
    TUPLE { STATUS INTEGER , SNO CHAR , CITY CHAR , SNAME CHAR } ;
```

В любой момент времени значением переменной S_TUPLE является кортеж с таким же заголовком, как у переменной-отношения S (я сознательно записал атрибуты в другом порядке, просто чтобы показать, что порядок не играет роли). Таким образом, можно представить себе фрагмент кода, который сначала извлекает отношение, содержащее один кортеж (быть может, только кортеж для поставщика S1), из текущего значения переменной-отношения S, затем извлекает единственный кортеж из этого «однокортежного» отношения и, наконец, присваивает этот кортеж переменной S_TUPLE. На языке **Tutorial D** это записывается так:

```
S_TUPLE := TUPLE FROM ( S WHERE SNO = 'S1' ) ;
```

Важное замечание: Тут не должно быть недопонимания. Хотя переменная, подобная S_TUPLE, безусловно, может понадобиться в какой-то прикладной программе, которая обращается к базе данных о постав-

щиках и деталях, я вовсе не хочу сказать, что такая переменная может появляться внутри самой базы данных. Реляционная база данных содержит переменные только одного вида: переменные-отношения, то есть только такие переменные допустимы в реляционной базе данных. В приложении А я еще вернусь к этой теме в связи с так называемым *принципом информации*.

Кстати говоря, обратите внимание на (отчетливо видное в приведенном выше примере) логическое различие между кортежем *t* и отношением *r*, которое содержит только кортеж *t*. Это совершенно разные типы: тип *t* – кортеж, а тип *r* – отношение (хотя у того и другого одинаковые заголовки).

И в завершение этого раздела еще несколько замечаний:

- Хотя и у кортежа, и у отношения есть видимые пользователю компоненты (их атрибуты), нигде не требуется, чтобы эти компоненты физически хранились. На самом деле физическое представление значений таких типов должно быть скрыто от пользователя точно так же, как оно скрыто для скалярных типов. (Вспомните обсуждение физической независимости от данных в главе 1.)
- Как и скалярные типы, типы кортежей и отношений, безусловно, нуждаются в ассоциированных селекторах (и литералах как частном случае селекторов). Детали я отложу до следующей главы. Однако операторы THE_ им ни к чему; их место занимают операторы, предоставляющие доступ к соответствующим атрибутам (эти операторы играют примерно такую же роль, как операторы THE_ для скалярных типов).
- В типах кортежей и отношений должны быть также определены операторы присваивания и сравнения на равенство. Пример присваивания кортежу был приведен в этом разделе; детали, касающиеся других операторов, отложим до следующей главы.

Скалярные типы в SQL

Теперь обратимся к языку SQL. Он поддерживает следующие системные скалярные типы, понятные без дополнительных пояснений[1]:

```
BOOLEAN                  NUMERIC(p,q)      DATE
CHARACTER(n)             DECIMAL(p,q)      TIME
CHARACTER VARYING(n)     INTEGER           TIMESTAMP
FLOAT(p)                 SMALLINT          INTERVAL
```

[1] Он также позволяет пользователям определять собственные типы, но я уже говорил, что в этой главе не собираюсь уделять много внимания пользовательским типам.

Поддерживается также целый ряд умолчаний, сокращений и альтернативных написаний, например: CHAR вместо CHARACTER, VARCHAR вместо CHARACTER VARYING, INT вместо INTEGER; я не хочу вдаваться в детали, отмечу лишь, что SQL, в отличие от **Tutorial D**, требует, чтобы в типах символьных строк задавалась длина. Добавлю следующее:

- Для всех этих типов поддерживаются литералы, записываемые в более-менее традиционном формате. *Примечание*: по причинам, которые здесь не так уж важны, для этих типов не нужны более общие селекторы, равно как и операторы THE_.
- Для всех этих типов поддерживается явный оператор присваивания. Его синтаксис таков:

```
SET <scalar variable ref> = <scalar exp> ;
```

  Скалярное присваивание также производится неявно при выполнении других операций (например, FETCH). *Примечание*: в этой книге в формальных определениях синтаксиса, подобных приведенному выше, я буду использовать *ref* и *exp* как сокращения слов *reference* (ссылка) и *expression* (выражение) соответственно.
- Для всех этих типов также поддерживается явный оператор сравнения на равенство[1]. Операторы сравнения на равенство также вызываются неявно при выполнении других операций (например, соединения, объединения, группирования и устранения дубликатов).
- По поводу типа BOOLEAN следует добавить, что хотя он и включен в стандарт SQL, но поддерживается очень немногими из основных представленных на рынке продуктов, если вообще поддерживается. Разумеется, булевы выражения могут встречаться во фразах WHERE, ON и HAVING, даже если система не поддерживает тип BOOLEAN как таковой. Однако в таблицах не может быть столбцов типа BOOLEAN, и объявлять переменные этого типа тоже нельзя. Поэтому иногда приходится искать обходные пути.
- Поддерживается еще несколько системных типов, в том числе «большие символьные объекты» (CLOB), «большие двоичные объекты» (BLOB), «символьные строки в национальной кодировке» (NCHAR и т. п.) и другие (в частности, XML). Но подробное рассмотрение этих типов выходит за рамки настоящей книги.

---

[1] К сожалению, в этой поддержке имеется существенный изъян. Например, в случае типов CHAR и VARCHAR оператор «=» может вернуть TRUE, несмотря на то что типы сравниваемых операндов, очевидно, различны (см. раздел «Приведения типов в SQL»). Кроме того, для всех типов, а не только для CHAR и VARCHAR, оператор «=» может не вернуть TRUE, даже если операнды неразличимы; в частности, так бывает, когда оба сравниваемых операнда имеют «значение» null (но это не единственный случай).

- Наконец, помимо вышеупомянутых скалярных типов, SQL также поддерживает нечто, именуемое доменами, но домены в смысле SQL – это вообще не типы, а просто вынесенное «определение общего столбца» с целым рядом довольно странных свойств, которые также выходят за рамки этой книги. Можете пользоваться ими, если хотите, но не допускайте ошибки, считая их настоящими реляционными доменами (то есть типами).

Проверка и приведение типов в SQL

SQL поддерживает лишь слабую форму строгой типизации (если вы понимаете, что я имею в виду). Точнее:

- Значение типа BOOLEAN можно присваивать только переменным типа BOOLEAN и сравнивать только со значениями типа BOOLEAN.
- Значения типа символьной строки можно присваивать только переменным типа символьной строки и сравнивать только со значениями типа символьной строки (где под «символьной строкой» понимается тип CHAR или VARCHAR).
- Числовые значения можно присваивать числовым переменным и сравнивать только с числовыми значениями (где под «числовыми» понимаются типы FLOAT, NUMERIC, DECIMAL, INTEGER или SMALLINT).
- Для дат, времени, временных меток и интервалов также действуют определенные правила проверки типов, но эти детали выходят за рамки настоящей книги.

Таким образом, сравнение числа с символьной строкой не допускается. Однако сравнение двух чисел допустимо, даже если они принадлежат разным типам, например INTEGER и FLOAT (в данном случае значение типа INTEGER будет приведено к типу FLOAT еще до выполнения сравнения). Что подводит нас к вопросу о приведении типов... В информатике признан общий принцип: по возможности избегать приведения типов, поскольку оно чревато ошибками. Конкретно в SQL приведение типов приводит к одному странному последствию: некоторые операции объединения, пересечения и разности могут давать результат, содержащий строки, которых не было ни в одном операнде! Рассмотрим, к примеру, SQL-таблицы T1 и T2, показанные на рис. 2.3. Предположим, что столбец X в таблице T1 имеет тип INTEGER, а в таблице T2 – NUMERIC(5,1), тогда как столбец Y в таблице T1 имеет тип NUMERIC(5,1), а в таблице T2 – INTEGER. Теперь рассмотрим такой SQL-запрос:

```
SELECT X , Y FROM T1
UNION
SELECT X , Y FROM T2
```

Как можно заключить из рис. 2.3, оба столбца X и Y в результате этого запроса (см. самую правую таблицу) имеют тип NUMERIC(5,1), и все значения в этих столбцах получаются путем приведения некоторого значения типа INTEGER к типу NUMERIC(5,1). Следовательно, результат состоит исключительно из строк, которых нет ни в T1, ни в T2, – согласитесь, это какое-то странное объединение.[1]

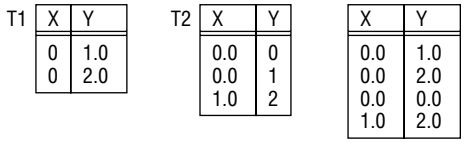


Рис. 2.3. *Очень странное «объединение»*

Настоятельная рекомендация: Старайтесь по-возможности избегать приведения типов. (Лично я предпочел бы отказаться от них вовсе, вне зависимости от того, говорим мы об SQL или еще о чем-то.) Но в контексте SQL я советую задавать один и тот же тип для одноименных столбцов; если вы будете придерживаться этой и других приведенных в книге рекомендаций, то сделаете большой шаг к тому, чтобы исключить приведения типов из своей практики. А когда обойтись без них все-таки не удается, я рекомендую выполнять их явно – с помощью оператора CAST или эквивалентного ему. Например (в применении к обсуждавшемуся выше запросу с UNION):

```
SELECT CAST ( X AS NUMERIC(5,1) ) AS X , Y FROM T1
UNION
SELECT X , CAST ( Y AS NUMERIC(5,1) ) AS Y FROM T2
```

Однако для полноты картины добавлю, что некоторые приведения типов, к сожалению, встроены в SQL, так что избежать их невозможно. (Я понимаю, что в этом месте книги приведенные ниже замечания вряд ли уместны, но не хочу опустить их вовсе.) Точнее:

- Если табличное выражение *tx* используется в качестве однострочного подзапроса, то предполагается, что таблица *t*, которую обозна-

[1] В связи с этим примером один из рецензентов высказал предположение, что такая «странность» объединения на практике, возможно, не так уж существенна, поскольку информация при этом не теряется. Что ж, для данного конкретного примера это замечание, быть может, и справедливо. Но если SQL желает определить оператор, который с очевидностью ведет себя не так, как оператор объединения в реляционной модели, то мне кажется, что, во-первых, не стоит называть такой оператор «объединением» (UNION), а, во-вторых (и это важнее), не мое дело демонстрировать, что такое «объединение» иногда приводит к проблемам, – скорее, проектировщики SQL должны доказывать, что этого не может быть.

чает *tx*, содержит всего одну строку *r*, и эта таблица *t* приводится к типу строки *r*. *Примечание*: Термин *подзапрос* встречается в SQL-контекстах повсеместно. Подробно я расскажу об этом в главе 12, а пока вы можете просто считать, что это табличное выражение, заключенное в круглые скобки.

- Если табличное выражение *tx* используется в качестве скалярного подзапроса, то предполагается, что таблица *t*, которую обозначает *tx*, содержит всего один столбец и всего одну строку, то есть состоит из единственного значения *v*, и для этой таблицы *t* дважды выполняется приведение типа, дающее в результате значение *v*. *Примечание*: этот случай встречается, в частности, при выполнении агрегирования в духе SQL (см. главу 7).
- На практике строчное выражение *rx* в сравнениях вида *rx* θ *sq* с модификатором ALL или ANY (здесь θ – любой оператор сравнения, например >ALL или <ANY, а *sq* – подзапрос) часто состоит из простого *скалярного* выражения, и в таком случае скалярное значение, обозначаемое этим выражением, приводится к типу строки, состоящей только из этого скалярного значения. *Примечание*: В этой книге я употребляю термин *строчное выражение* для обозначения либо однострочного подзапроса, либо вызова селектора строки (где под селектором строки я в свою очередь понимаю то, что в SQL называется конструктором значения строки – см. главу 3). Иначе говоря, термином *строчное выражение* я обозначаю любое выражение, результатом вычисления которого является строка; а термином *табличное выражение* – выражение, результатом вычисления которого является таблица. Что касается сравнений с модификаторами ALL и ANY, то они обсуждаются в главе 11.

Наконец, в SQL термин *приведение* используется также в очень специальном смысле, когда речь идет о строках символов. Но эти детали выходят за рамки данной книги.

Схемы упорядочения в SQL

Правила проверки и приведения типов в SQL, особенно в случае строк символов, на самом деле гораздо сложнее, чем я описывал до сих пор, и тут необходимо внести некоторую ясность. Вообще-то, в такого рода книге этой темы можно коснуться лишь поверхностно, но основная идея такова: с любой строкой символов ассоциированы (а) некоторый *набор символов* (*кодировка*) и (б) некоторая *схема упорядочения* (collation, или collating sequence). Схема упорядочения – это ассоциированное с набором символов правило, описывающее сравнение строк, состоящих из символов, принадлежащих данному набору. Пусть *C* – схема упорядочения для набора символов *S*, и пусть *a* и *b* – любые два символа из набо-

ра *S*. Тогда *C* должна удовлетворять следующему условию: из трех сравнений $a < b$, $a = b$ и $a > b$ ровно одно дает TRUE, а два других – FALSE (при данной схеме *C*).

С основной идеей понятно. Но есть некоторые осложнения. Одно из них связано с тем, что для каждой схемы упорядочения может быть определен режим PAD SPACE (дополнять пробелами) или NO PAD (не дополнять). Предположим, что со строками 'AB' и 'AB ' (обратите внимание на пробел в конце второй строки) ассоциированы одинаковые набор символов и схема упорядочения. Очевидно, что эти строки различаются, и все же они считаются «одинаковыми», если задан режим PAD SPACE. **Рекомендация**: Не используйте режим PAD SPACE, по возможности старайтесь обходиться режимом NO PAD. (Отметим, однако, что режим PAD SPACE / NO PAD влияет только на сравнение, к присваиванию он не имеет ни малейшего отношения.)

Второе осложнение обусловлено тем, что сравнение $a = b$ может возвращать TRUE при данной схеме упорядочения, даже если символы *a* и *b* различны. Например, можно определить схему упорядочения CASE_INSENSITIVE, при которой строчные и заглавные буквы не различаются. И, стало быть, снова заведомо различные строки будут считаться одинаковыми.

Таким образом, мы видим, что в SQL некоторые сравнения вида *v1* = *v2* могут возвращать TRUE и тогда, когда *v1* и *v2* различны (и, возможно, даже принадлежат разным типам). Такие пары значений я буду называть «равными, но различимыми». Отмечу, что сравнение на равенство выполняется, часто неявно, в самых разных контекстах (например, MATCH, LIKE, UNIQUE, UNION и JOIN), и во всех этих случаях подразумевается семантика «равны, даже если различимы». Например, пусть определена вышеупомянутая схема упорядочения CASE_INSENSITIVE и для нее установлен режим PAD SPACE. Тогда, если для столбцов PNO в таблицах P и SP применяется эта схема упорядочения и если в какой-то строке P столбец PNO содержит значение 'P2', а в некоторой строке SP одноименный столбец содержит значение 'p2 ', то эти строки будут считаться удовлетворяющими ограничению внешнего ключа в направлении от SP к P, несмотря на то что внешний ключ содержит строчную букву 'p' и пробел в конце строки.

Более того, при вычислении выражений, содержащих такие операторы, как UNION, INTERSECT, EXCEPT, JOIN, GROUP BY и DISTINCT, система иногда вынуждена решать, какое из нескольких равных, но различимых значений следует включить в некоторый столбец результирующей строки. К сожалению, сам стандарт SQL в таких ситуациях не дает четких инструкций. Поэтому некоторые табличные выражения оказываются неоднозначными – в терминологии SQL, *потенциально недетерминированными* – в том смысле, что SQL не полностью опре-

деляет, как они должны вычисляться; и действительно, при различных обстоятельствах могут быть получены разные результаты – и это считается допустимым. Например, если для столбца Z в таблице T задана схема упорядочения CASE_INSENSITIVE, то запрос SELECT MAX(Z) FROM T может в одном случае вернуть 'ZZZ', а в другом – 'zzz', даже если между этими двумя моментами таблица T не изменялась.

Я не стану здесь приводить правила SQL, описывающие, какие выражения считаются «потенциально недетерминированными» (отложу до главы 12). Но важно отметить, что такие выражения недопустимы в ограничениях целостности (см. главу 8), поскольку они могли бы привести к непредсказуемому исходу операций обновления (иногда заканчивается успешно, а иногда с ошибкой). В частности, отсюда следует, что многие табличные выражения – иногда даже простые выражения SELECT – не допускаются в ограничениях целостности, если в них участвует столбец строкового типа! **Настоятельная рекомендация**: Старайтесь держаться подальше от потенциально недетерминированных выражений.

Тип строки и таблицы в SQL

Следующий пример определения переменной типа кортежа уже приводился в разделе «Скалярные и нескалярные типы»:

```
VAR S_TUPLE
    TUPLE { STATUS INTEGER , SNO CHAR , CITY CHAR , SNAME CHAR } ;
```

Как вы помните, выражение TUPLE{...} здесь – это вызов генератора типа TUPLE. В SQL имеется соответствующий генератор типа ROW (хотя называется он конструктором типа). Вот как в SQL записывается аналогичная конструкция:

```
DECLARE S_ROW /* строчная переменная SQL */
    ROW ( SNO    VARCHAR(5) ,
          SNAME  VARCHAR(25) ,
          STATUS INTEGER ,
          CITY   VARCHAR(20) ) ;
```

Но в отличие от кортежей, компоненты строк в SQL упорядочены слева направо[1]; в данном случае существует 24 (= 4 * 3 * 2 * 1) различных типа строк, состоящих из одних и тех же компонентов (!).

В SQL поддерживается также присваивание строк. Вот, например, как выглядит аналог приведенного выше присваивания кортежей на языке **Tutorial D**:

[1] Как ни странно, в SQL компоненты строчных типов, порождаемых явным вызовом конструктора типа ROW (и компоненты строк таких типов), называются не столбцами, а *полями*.

```
SET S_ROW = ( S WHERE SNO = 'S1' ) ;
```

Примечание

Здесь выражение в правой части является однострочным подзапросом, то есть, с точки зрения синтаксиса, табличным выражением, однако работает оно, как строчное выражение (см. выше раздел «Проверка и приведение типов в SQL»).

Присваивания строк участвуют и в SQL-предложениях UPDATE (см. главу 3).

Но вернемся к таблицам: интересно, что в SQL на самом деле совсем нет генератора (или конструктора) типа TABLE! – то есть не существует никакого прямого аналога генератора типа RELATION, описанного выше в этой главе. Однако имеется механизм CREATE TABLE для определения того, что по праву следовало бы называть табличными переменными. Вспомним, например, следующее определение из раздела «Скалярные и нескалярные типы»:

```
VAR S BASE
    RELATION { SNO CHAR , SNAME CHAR , STATUS INTEGER , CITY CHAR }
    KEY { SNO } ;
```

А вот его аналог на языке SQL:

```
CREATE TABLE S
  ( SNO    VARCHAR(5)   NOT NULL ,
    SNAME  VARCHAR(25)  NOT NULL ,
    STATUS INTEGER      NOT NULL ,
    CITY   VARCHAR(20)  NOT NULL ,
    UNIQUE ( SNO ) ) ;
```

Однако обратите внимание, что в этом примере нет ничего – никаких лексем, – что могло бы быть логически названо «вызовом конструктора типа TABLE». (Этот факт станет еще нагляднее, если принять во внимание, что спецификация UNIQUE(SNO) не обязана располагаться после определений столбцов, а может находиться почти в любом месте, например между определениями столбцов SNO и SNAME.) Если вообще можно говорить о каком-то типе переменной S (в SQL), то этот тип – не более чем *мультимножество строк*, каждая из которых имеет тип ROW (SNO VARCHAR(5), SNAME VARCHAR(25), STATUS INTEGER, CITY VARCHAR(20)).

И все-таки я должен сказать, что в SQL поддерживается нечто, именуемое «типизированными таблицами». Правда, термин не слишком удачный, поскольку если TT – типизированная таблица, для которой определен «тип *T*», то ни TT, ни какая-либо из входящих в нее строк типу *T* не принадлежат! И я считаю, что таких таблиц следует в любом случае

избегать, потому что они неразрывно переплетены с поддержкой в SQL указателей, а указатели в реляционной модели запрещены[1]. На самом деле, если в некоторой таблице имеется столбец, значениями которого являются указатели на строки в другой таблице, то эта таблица не может представлять отношение в смысле реляционной модели. Однако, как я только что отметил, такие таблицы, к сожалению, разрешены в SQL; указатели называются *ссылочными значениями*, а про содержащие их столбцы говорят, что они имеют *ссылочный тип* (типа REF). Откровенно говоря, непонятно, зачем эти средства вообще включены в SQL; не видно никакой полезной функциональности, которую нельзя было бы с тем же успехом реализовать по-другому – и даже лучше. **Настоятельная рекомендация:** Не используйте ни ссылочные переменные, ни все остальное, что с ними связано.

Отступление

Во избежание недоразумений я хочу добавить, что в SQL термин «ссылка» используется в двух совершенно разных смыслах. Один из них вкратце описан выше. Другой, более старый, связан с внешними ключами; говорят, что значение внешнего ключа в одной строке является «ссылкой» на строку, содержащую соответственное значение ключа. Отметим, однако, что внешние ключи – это не указатели! Между этими двумя понятиями существует несколько логических различий, в частности тот факт, что внешние ключи ссылаются на строки, которые являются значениями, тогда как указатели – это адреса и, следовательно, по определению ссылаются на переменные (вспомните главу 1, где говорилось, что именно переменные, а не значения имеют «позицию во времени и в пространстве»).

Заключительные замечания

Существует распространенное заблуждение, будто реляционная модель имеет дело только с относительно простыми типами: числами, строками, возможно, датой и временем и еще некоторыми. В этой главе я попытался, среди прочего, показать, что это не так. Напротив, отношения могут иметь атрибуты *произвольного типа* – реляционная модель ничего не говорит о том, каким должен быть тип, и в действительности он может быть сколь угодно сложным (с ограничением, о котором я ско-

[1] Быть может, следует пояснить, что я понимаю под *указателем*. Указатель – это значение (по сути дела, адрес), для которого должны быть определены некоторые специальные операторы, прежде всего ссылки и разыменования. Приведу нестрогие определения этих операторов: (а) если дана переменная *V*, то результатом применения оператора ссылки к *V* является адрес *V*; (б) если дано значение *v* типа указателя (то есть адрес), то результатом применения оператора разыменования к *v* является переменная, на которую указывает *v* (то есть переменная с данным адресом).

ро скажу). Другими словами, вопрос о том, какие типы поддерживаются, ортогонален вопросу о поддержке самой реляционной модели. Или менее точно, зато легче запоминается: *типы ортогональны таблицам*.

Я хочу также напомнить, что описанное выше положение вещей ни в коей мере не нарушает требований первой нормальной формы, которые означают лишь, что любой кортеж любого отношения содержит в позиции каждого атрибута единственное значение соответствующего типа. Но теперь, когда мы знаем, что тип может быть произвольным, оказывается, что все отношения находятся в первой нормальной форме по определению.

И наконец, во введении к этой главе я упомянул о существовании неких важных исключений из правила о том, что атрибут отношения может иметь любой тип. На самом деле, таких исключений два. Первое – которое я сейчас немного упрощу – состоит в том, что если отношение *r* принадлежит типу *T*, то никакой атрибут *r* не может иметь тот же тип *T* (поразмыслите над этим!). Второе – запрещает отношениям в базе данных иметь атрибуты типа указателя. До появления реляционной модели базы данных просто пестрели указателями, а доступ к хранящимся в них данным сводился к следованию по цепочке указателей; из-за этого прикладное программирование было чревато ошибками, а прямой доступ со стороны пользователя вообще был невозможен. (Это не единственные связанные с указателями проблемы, просто наиболее бросающиеся в глаза.) Кодд хотел уйти от таких проблем в своей реляционной модели и, разумеется, преуспел в этом.

Упражнения

Упражнение 2.1. Что такое тип? В чем разница между доменом и типом?

Упражнение 2.2. Как вы понимаете термин *селектор*? И что в точности означает слово *литерал*?

Упражнение 2.3. Что такое оператор THE_?

Упражнение 2.4. Верно или неверно следующее утверждение: физические представления всегда скрыты от пользователя?

Упражнение 2.5. В этой главе мы затронули еще несколько логических различий (вернитесь к главе 1, если вам требуется освежить в памяти это важное понятие), а именно:

аргумент	*и*	*параметр*
база данных	*и*	*СУБД*
внешний ключ	*и*	*указатель*
аргумент	*и*	*параметр*
сгенерированный тип	*и*	*несгенерированный тип*

отношение	*и*	*тип*
скаляр	*и*	*нескаляр*
тип	*и*	*представление*
определенный пользователем тип	*и*	*системный тип*
отношение	*и*	*тип*
определенный пользователем оператор	*и*	*системный оператор*

Для каждого случая точно объясните, в чем состоит логическое различие.

Упражнение 2.6. Как вы понимаете термин *приведение типа*? Почему приведения типов лучше избегать?

Упражнение 2.7. Почему подавление проверки доменов не имеет смысла?

Упражнение 2.8. Что такое генератор типа?

Упражнение 2.9. Дайте определение первой нормальной формы. Как вы думаете, почему она так называется?

Упражнение 2.10. Пусть *X* – выражение. Какой тип имеет *X*? Почему так важно, что у *X* есть какой-то тип?

Упражнение 2.11. Используя данное в этой главе определение оператора REFLECT (раздел «Что такое тип?») в качестве шаблона, определите на языке **Tutorial D** оператор, который получает на входе целое число и возвращает его кубическую степень.

Упражнение 2.12. На языке **Tutorial D** определите оператор, который получает точку с декартовыми координатами *x* и *y* и возвращает точку с координатами *f(x)* и *g(y)*, где *f* и *g* – предопределенные операторы.

Упражнение 2.13. Приведите пример типа отношения. Опишите различия между типами отношений, отношениями-значениями и переменными-отношениями.

Упражнение 2.14. На языке **Tutorial D** или на SQL или на том и другом определите переменные-отношения P и SP из базы данных о поставщиках и деталях. Если вы дадите определения и на SQL, и на **Tutorial D**, укажите между ними столько различий, сколько сможете. Насколько важен тот факт, что переменная-отношение P (к примеру) принадлежит некоторому типу отношения?

Упражнение 2.15. Предположим, что в базе данных об отделах и служащих из главы 1 (рис. 1.1) атрибуты имеют следующие определенные пользователем типы:

```
DNO    : DNO
DNAME  : NAME
```

```
BUDGET : MONEY
ENO    : ENO
ENAME  : NAME
SALARY : MONEY
```

Предположим также, что отделы имеют атрибут LOCATION определенного пользователем типа CITY (к примеру). Какие из приведенных ниже скалярных выражений допустимы? Для тех, что допустимы, назовите тип результата; для остальных напишите выражение, которое дало бы желаемый эффект.

a. `LOCATION = 'London'`

b. `ENAME = DNAME`

c. `SALARY * 5`

d. `BUDGET + 50000`

e. `ENO > 'E2'`

f. `ENAME || DNAME`

g. `LOCATION || 'burg'`

Упражнение 2.16. Иногда говорят, что типы в некотором смысле являются переменными. Например, по мере роста предприятия количество цифр в номере служащего может увеличиться с трех до четырех, поэтому может возникнуть необходимость изменить «множество всех возможных номеров служащих». Обсудите этот вопрос.

Упражнение 2.17. Тип – это множество значений, а множество может быть пустым. Следовательно, мы могли бы определить пустой тип, как тип с пустым множеством значений. Можете ли вы придумать применения такому типу? Поддерживается ли такой тип в SQL?

Упражнение 2.18. В реляционном мире оператор сравнения на равенство «=» применим к любому типу. Напротив, в SQL не требуется, чтобы для любого типа был определен оператор «=» (я не упомянул об этом в тексте главы, но это справедливо, в частности, для определенных пользователем типов), а в тех случаях, когда оператор имеется, его семантика не всегда полностью определена. Какие следствия вытекают из такого положения вещей?

Упражнение 2.19. Продолжая предыдущее упражнение, мы можем сказать, что в реляционном мире *v1 = v2* дает значение TRUE тогда и только тогда, когда применение оператора *Op* к *v1* и применение того же оператора *Op* к *v2* всегда дает один и тот же результат для всех возможных операторов *Op*. Но это еще одно правило, которое в SQL нарушено. Можете ли вы привести примеры такого нарушения? Каковы его последствия?

Упражнение 2.20. Почему из реляционной модели исключены указатели?

Упражнение 2.21. *Принцип присваивания* – очень простой, но вместе с тем фундаментальный – гласит, что после присваивания значения v переменной V результатом сравнения $V = v$ должно быть TRUE (см. главу 5). Но в SQL нарушается и этот принцип (и даже чуть ли не повсеместно). Можете ли вы привести примеры такого нарушения? Каковы его последствия?

Упражнение 2.22. Полагаете ли вы, что типы – это «принадлежность» базы данных в том же смысле, что переменные-отношения?

Упражнение 2.23. В первом примере SQL-выражения SELECT, приведенном в этой главе, я отметил, что завершающая точка с запятой отсутствует, потому что это выражение, а не предложение. Но в чем различие между ними?

Упражнение 2.24. Объясните максимально точно, в чем состоит логическое различие между отношением с RVA-атрибутом и «отношением» с повторяющейся группой.

Упражнение 2.25. Что такое подзапрос?

Упражнение 2.25. В тексте главы я сказал, что оператор «=» по идее должен быть применим к любому типу. Но как насчет типов строк и таблиц в SQL?

3

Кортежи и отношения, строки и таблицы

После первых двух глав у вас должно было сложиться довольно отчетливое понимание того, что такое кортежи и отношения, – по крайней мере, на интуитивном уровне. Теперь я хочу определить эти понятия более точно и рассмотреть вытекающие из этих определений следствия. Кроме того, я собираюсь описать аналогичные конструкции SQL (то есть строки и таблицы) и предложить ряд конкретных рекомендаций, которые будут способствовать достижению нашей цели – использованию SQL в реляционном духе. Наверное, стоит сразу предупредить, что формальные определения могут поначалу отпугнуть, но с определениями так всегда бывает. Сами понятия очень просты, нужно лишь продраться сквозь дебри формализма, но теперь вы к этому подготовлены, так как хотя бы терминология уже известна.

Что такое кортеж?

Это кортеж?

SNO CHAR	SNAME CHAR	STATUS INTEGER	CITY CHAR
S1	Smith	20	London

Нет, это лишь изображение кортежа, а не кортеж как таковой (и обратите внимание, что на этот раз я показал на рисунке имена типов, а не только имена атрибутов). В главе 1 мы видели, что существует различие между предметом и изображением предмета, и это различие может быть весьма существенным. Например, атрибуты в кортежах не упорядочены слева направо, поэтому следующий рисунок годится для изображения того же самого кортежа ничуть не хуже (или не лучше?):

STATUS : INTEGER	SNAME : CHAR	CITY : CHAR	SNO : CHAR
20	Smith	London	S1

Я, конечно, буду пользоваться подобными рисунками и дальше, но имейте в виду, что это всего лишь рисунки, из которых иногда можно сделать неверные выводы. Предупредив об этой опасности, я могу теперь точно определить, что такое кортеж:

> **Определение**: Пусть существуют имена типов *T1*, *T2*, ..., *Tn* ($n \geq 0$), необязательно все различные. Сопоставим с каждым *Ti* имя атрибута *Ai*; каждая из *n* получившихся пар имя-атрибута/имя-типа называется *атрибутом*. Сопоставим с каждым атрибутом значение атрибута *vi* типа *Ti*; каждая из *n* получившихся пар атрибут/значение называется *компонентом*. Множество, состоящее из всех *n* определенных таким образом компонентов, назовем его *t*, называется *значением кортежа* (или для краткости просто *кортежем*) над атрибутами *A1*, *A2*, ..., *An*. Значение *n* называется *степенью t*; кортеж степени 1 называется *унарным*, кортеж степени 2 – *бинарным*, кортеж степени 3 – *тернарным*, и, более общо, кортеж степени *n* называется *n-арным*. Множество, состоящее из всех *n* атрибутов, называется *заголовком t*.

Например, обращаясь к одному из предыдущих рисунков, на которых изображен наш привычный кортеж для поставщика S1, имеем:

Степень: 4. Считается, что заголовок тоже имеет степень 4.

Имена типов: CHAR, CHAR, INTEGER, CHAR.

Соответствующие имена атрибутов: SNO, SNAME, STATUS, CITY.

Соответствующие значения атрибутов: 'S1', 'Smith', 20, 'London'. Кстати, обратите внимание на кавычки, в которые заключены значения строк символов; я не показывал кавычки на рисунках, хотя, наверное, стоило бы – это было бы правильнее.

Отступление

Предположим на минутку, как мы делали в разделе «Сравнение на равенство» в главе 2, что атрибут SNO имеет тип SNO (определенный пользователем), а не CHAR. Тогда было бы совсем неправильно говорить, что значение SNO в рассматриваемом кортеже равно S1 или даже 'S1'; нужно было бы написать SNO('S1'). Значение типа SNO – это значение типа SNO, а не значение типа CHAR! Несовпадение типов, безусловно, является существенным логическим различием. (Напомню, что в главе 2 говорилось о том, что выражение SNO('S1') – это вызов селектора – фактически литерал – типа SNO.)

Заголовок: здесь проще всего привести еще один рисунок:

Разумеется, на этом рисунке представлено множество без какого-либо фиксированного порядка атрибутов. Вот еще одно изображение того же самого заголовка:

Упражнение: сколько существует различных рисунков такого вида, представляющих данный заголовок? (Ответ: 4 * 3 * 2 * 1 = 24)

Итак, кортеж – это значение; поэтому, как и все значения, он имеет тип (это мы уяснили в главе 2), и этот тип, как и все типы, имеет имя. В языке **Tutorial D** такие имена принимают вид TUPLE{*H*}, где {*H*} – заголовок. Стало быть, в нашем примере получается такое имя:

```
TUPLE { SNO CHAR , SNAME CHAR , STATUS INTEGER , CITY CHAR }
```

(но порядок следования атрибутов произволен).

Еще раз повторю: кортеж – это значение. Следовательно, как и все значения, он должен возвращаться каким-то *вызовом селектора* (естественно, вызовом селектора *кортежа*, коль скоро значением является кортеж). Вот как выглядит вызов селектора кортежа в нашем случае (опять-таки на языке **Tutorial D**):

```
TUPLE { SNO 'S1' , SNAME 'Smith' , STATUS 20 , CITY 'London' }
```

(порядок записи компонентов произволен). Отметим, что в **Tutorial D** каждый компонент задается просто в виде пары имя-атрибута/выражение, где *выражение* обозначает соответствующее значение атрибута; тип атрибута опускается, поскольку его всегда можно вывести из типа *выражения*.

Вот другой пример (в отличие от предыдущего, это не литерал, потому что не все его аргументы заданы в виде литералов):

```
TUPLE { SNO SX , SNAME 'James' , STATUS STX , CITY SCX }
```

Я предполагаю, что SX, STX и SCX – переменные типа CHAR, INTEGER и CHAR соответственно.

Как следует из этих примеров, вызов селектора кортежа в языке **Tutorial D** в общем случае состоит из ключевого слова TUPLE, за которым следует список пар имя-атрибута/выражение, разделенных запятыми, причем весь список заключен в фигурные скобки. Таким образом, ключевое слово TUPLE играет в языке **Tutorial D** двоякую роль: оно используется в вызовах селекторов кортежей, как мы только что видели, и в именах типов кортежей, как было показано выше. Анало-

гичное замечание относится к ключевому слову RELATION (см. раздел «Что такое отношение» ниже в этой главе).

Следствия из определений

Теперь я хочу остановиться на некоторых важных следствиях, вытекающих из приведенных выше определений. Первое – *кортеж никогда не содержит null-значений*. Причина в том, что, по определению, любой кортеж содержит значение (соответствующего типа) каждого из своих атрибутов, а, как мы видели в главе 1, null вообще не является значением – вопреки тому факту, что в SQL часто, хотя и не всегда, он явно называется null-*значением*. **Рекомендация**: Так как словосочетание «null-значение» содержит в себе терминологическое противоречие, не пользуйтесь им, говорите просто null[1].

Итак, если кортеж не содержит null-значений, их тем более не может содержать отношение; следовательно, мы немедленно получаем по меньшей мере формальную причину отвергнуть саму концепцию null-значений. Но в следующей главе я приведу и более прагматичные основания для этого.

Второе следствие – *любое подмножество кортежа тоже является кортежем, а любое подмножество заголовка – заголовком*. (*Примечание*: В соответствии со сложившейся практикой я будут употреблять выражения «*B* – подмножество *A*» и «*A* – надмножество *B*», подразумевая, что *A* и *B* могут, в частности, совпадать. Так, в рассматриваемом случае всякий кортеж является подмножеством себя самого, и то же самое справедливо для заголовков. Если мне понадобится исключить такую возможность, то я буду говорить о *собственных* подмножествах и надмножествах.) Если обратиться к нашему стандартному примеру кортежа для поставщика S1, – то, что можно было бы назвать «значением {SNO,CITY}» внутри кортежа, само является кортежем (степени 2):

SNO CHAR	CITY CHAR
S1	London

Заголовок этого кортежа показан на рисунке, а его тип – TUPLE {SNO CHAR, CITY CHAR}. Точно так же кортежем является и следующее подмножество:

SNO CHAR
S1

[1] Несмотря на эту рекомендацию, в переводе употребляется именно словосочетание null-значение в силу особенностей русского языка и устоявшейся в нем терминологии. – *Прим. перев.*

Это кортеж степени 1 типа TUPLE {SNO CHAR}. Таким образом, если мы имеем подобный кортеж и хотим получить доступ к *значению* конкретного атрибута – в данном случае 'S1', – то должны будем как-то извлечь его из объемлющего кортежа. Для этой цели язык **Tutorial D** предлагает синтаксис вида SNO FROM *t* (где *t* – произвольное выражение, обозначающее кортеж с атрибутом SNO). В SQL применяется квалификация с помощью точки: *t*.SNO. *Примечание*: В главе 2 мы видели, что кортеж *t* и отношение *r*, содержащее только данный кортеж *t* и ничего более, – не одно и то же. Аналогично, значение *v* и кортеж *t*, содержащий только это значение, – разные вещи; в частности, не совпадают их типы.

Уверен, вы знаете, что пустое множество является подмножеством любого множества. Отсюда следует, что кортеж с пустым заголовком и, значит, с пустым набором компонентов, допустим, хотя нарисовать такой кортеж на бумаге было бы несколько затруднительно, я даже пытаться не буду. Кортеж с пустым заголовком имеет тип TUPLE{} (у него нет компонентов); иногда мы будем называть его просто *0-кортежем*, чтобы подчеркнуть, что его степень равна 0. А иногда будем называть такой кортеж *пустым*. Если вы думаете, что у пустых кортежей нет практического применения, то смею заверить, что есть, – и, как это ни удивительно, очень важное. Я вернусь к этому вопросу в разделе «TABLE_DUM и TABLE_DEE».

И последнее, что я хочу обсудить в этом разделе, – понятие *равенства кортежей*. (Напомню, в главе 2 говорилось о том, что оператор сравнения «=» определен для любого типа, и типы кортежей – не исключение.) По сути дела, два кортежа равны тогда и только тогда, когда это один и тот же кортеж (точно так же, два целых числа равны тогда и только тогда, когда это одно и то же число). Но имеет смысл остановиться на семантике равенства кортежей более подробно, потому что она лежит в основе очень многих вещей в реляционной модели, например потенциальные ключи, внешние ключи и большинство операторов реляционной алгебры определяются в терминах равенства кортежей. Вот как звучит точное определение:

> **Определение**: Кортежи *tx* и *ty* считаются *равными* тогда и только тогда, когда они имеют одни и те же атрибуты *A1, A2, ..., An* – другими словами, принадлежат одному и тому же типу – и для любого *i* (*i = 1, 2, ..., n*) значение *vx* атрибута *Ai* в кортеже *tx* равно значению *vy* атрибута *Ai* в кортеже *ty*.

Кроме того (это может показаться очевидным, но уточнить все-таки стоит), два кортежа являются дубликатами тогда и только тогда, когда они равны.

Кстати, из этого определения сразу же следует, что все 0-кортежи являются дубликатами друг друга. Поэтому мы вправе говорить о единственном 0-кортеже, а не о каком-то из 0-кортежей.

Наконец, заметим, что операторы сравнения «<» и «>» к кортежам неприменимы. Причина состоит в том, что кортежи – это множества (точнее, множества компонентов), а для множеств общего вида эти операторы не имеют смысла.

Строки в SQL

В языке SQL поддерживаются не кортежи, а строки; в частности, поддержаны *типы строк*, *конструктор типа строки* и *конструкторы значений строк*, которые являются соответственно аналогами типов кортежей, генератора типа TUPLE и селекторов кортежей в языке **Tutorial D**. (Типы строк и конструкторы типов строк, но не конструкторы значений строк, обсуждались в главе 2.) Но эти аналогии в лучшем случае приблизительные, поскольку у строк, в отличие от кортежей, есть очень важное свойство: их компоненты упорядочены слева направо. Например, оба выражения ROW(1,2) и ROW(2,1) – допустимые вызовы конструктора значения строки, но в SQL они представляют две разных строки. *Примечание*: Ключевое слово ROW при вызове конструктора значения строки в SQL необязательно и на практике чаще всего опускается.

Благодаря наличию упорядоченности слева направо компоненты строки в SQL могут идентифицироваться (и часто идентифицируются) не по имени, а по порядковому номеру. Рассмотрим, к примеру, следующий вызов конструктора значения строки (фактически это строковый литерал, хотя в SQL такой термин не употребляется):

```
( 'S1' , 'Smith' , 20 , 'London' )
```

Очевидно, что (среди прочего) у этой строки есть компонент со значением 'Smith'; однако логически мы не можем сказать, что это «компонент SNAME», а можем лишь сказать, что это *второй* компонент.

Следует добавить, что в SQL любая строка содержит по крайней мере один компонент, в этом языке нет аналога 0-кортежей из реляционной модели.

Как отмечалось в главе 2, SQL поддерживает также операцию присваивания строк[1]. В частности, такие присваивания производятся (по существу) при выполнении SQL-предложения UPDATE. Например, следующее предложение UPDATE

```
UPDATE S
SET    STATUS = 20 , CITY = 'London'
WHERE  CITY = 'Paris' ;
```

[1] Строго говоря, мне не следовало в этой главе упоминать о каких бы то ни было присваиваниях, так как она посвящена значениям, а не переменным. Однако мне показалось удобным хотя бы вкратце упомянуть о присваивании строк в SQL именно в этом месте.

по определению логически эквивалентно такому (обратите внимание на присваивание строки во второй строчке):

```
UPDATE S
SET  ( STATUS , CITY ) = ( 20 , 'London' )
WHERE  CITY = 'Paris' ;
```

Что касается операторов сравнения, то большинство булевых выражений в SQL, в том числе (хотите верьте, хотите нет) простые «скалярные» сравнения, фактически определены в терминах строк, а не скаляров. Вот пример выражения SELECT, в котором фраза WHERE содержит явное сравнение строк:

```
SELECT SNO
FROM   S
WHERE  ( STATUS , CITY ) = ( 20 , 'London' )
```

Это выражение SELECT логически эквивалентно следующему:

```
SELECT SNO
FROM   S
WHERE  STATUS = 20 AND CITY = 'London'
```

А такое выражение

```
SELECT SNO
FROM   S
WHERE  ( STATUS , CITY ) <> ( 20 , 'London' )
```

логически эквивалентно следующему:

```
SELECT SNO
FROM   S
WHERE  STATUS <> 20
OR     CITY <> 'London'
```

Обратите внимание на OR в последней строчке!

Далее, поскольку компоненты строк упорядочены слева направо, SQL может также поддержать операторы «<» и «>» для сравнения строк, например:

```
SELECT SNO
FROM   S
WHERE  ( STATUS , CITY ) > ( 20 , 'London' )
```

Это выражение логически эквивалентно такому:

```
SELECT SNO
FROM   S
WHERE  STATUS > 20
OR   ( STATUS = 20 AND CITY > 'London' )
```

Однако на практике в подавляющем большинстве случаев сравниваются строки степени 1, например:

```
SELECT SNO
FROM   S
WHERE  ( STATUS ) = ( 20 )
```

До сих пор все встречавшиеся в выражениях сравнения операнды были конструкторами значения строки. Но если конструктор значения строки состоит из единственного скалярного выражения, заключенного в скобки, то эти скобки можно опустить:

```
SELECT SNO
FROM   S
WHERE  STATUS = 20
```

«Сравнение строк» во фразе WHERE на деле является скалярным сравнением (STATUS и 20 – скалярные выражения). Однако, строго говоря, в SQL такой вещи, как скалярное сравнение, не существует; выражение STATUS = 20 технически по-прежнему является сравнением строк (и «скалярные» операнды приводятся к типу строки).

Рекомендация: Если степень сравниваемых строк отлична от единицы (то есть речь не идет о сравнении скаляров), не пользуйтесь операторами сравнения «<», «<=», «>» и «>=»; они зависят от упорядочения столбцов слева направо и, следовательно, не имеют прямого аналога в реляционной модели, да и в любом случае чреваты серьезными ошибками. (В этой связи нелишне упомянуть, что когда эту функциональность было впервые предложено включить в SQL, авторы стандарта испытывали значительные трудности с надлежащим определением семантики, им даже потребовалось для этого несколько попыток.)

Что такое отношение?

Все примеры в этом разделе основаны на нашем привычном отношении «поставщики». Сначала приведу рисунок:

SNO CHAR	SNAME CHAR	STATUS INTEGER	CITY CHAR
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

а затем дам определение:

> **Определение:** Пусть {*H*} – заголовок кортежа, и пусть *t1*, *t2*, …, *tm* (m ≥ 0) – различные кортежи с заголовком {*H*}. Комбинация *r*, состоящая из *{H}* и множества кортежей {*t1, t2, ..., tm*}, называется *отношением-значением* (или для краткости просто *отношением*) над атрибутами *A1*, *A2*, …, *An*, где *A1*, *A2*, …, *An* – атрибуты, содержащиеся в заголовке *{H}*. *Заголовком r* является *{H}*; *r* имеет те же атрибуты (и, следовательно, те же имена и типы атрибутов), что

и заголовок. *Телом r* называется множество кортежей {*t1, t2, ..., tm*}. Величина *m* называется кардинальностью *r*.

Оставляю вам в качестве упражнения интерпретацию отношения «поставщики» в терминах приведенного выше определения. Однако я хочу по крайней мере, объяснить, почему эта штука называется отношением. По сути дела, любой кортеж в отношении представляет *n-арную* связь (relationship) в множестве из *n* значений (по одному для каждого атрибута кортежа), существующую в некоторый момент времени, а в математике такая связь называется отношением (relation). Таким образом, часто встречающееся «объяснение» происхождения названия «реляционная модель» – мол, в ней идет речь о «соотнесении (relating) одной таблицы с другой» – хотя в каком-то смысле правильно, но упускает самое главное. Реляционная модель названа так потому, что имеет дело с некоторыми абстракциями, которые в математике называются отношениями, пусть даже неформально мы представляем их в виде таблиц.

Итак, отношение, как и кортеж, само является значением и имеет тип, а у этого типа есть имя. В языке **Tutorial D** такие имена принимают вид RELATION{*H*}, где {*H*} – заголовок, например:

```
RELATION { SNO CHAR , SNAME CHAR , STATUS INTEGER , CITY CHAR }
```

(Атрибуты можно записывать в любом порядке.) Кроме того, любое отношение-значение возвращается вызовом некоторого селектора отношения, например:

```
RELATION
{ TUPLE { SNO 'S1' , SNAME 'Smith' , STATUS 20 , CITY 'London' } ,
  TUPLE { SNO 'S2' , SNAME 'Jones' , STATUS 10 , CITY 'Paris'  } ,
  TUPLE { SNO 'S3' , SNAME 'Blake' , STATUS 30 , CITY 'Paris'  } ,
  TUPLE { SNO 'S4' , SNAME 'Clark' , STATUS 20 , CITY 'London' } ,
  TUPLE { SNO 'S5' , SNAME 'Adams' , STATUS 30 , CITY 'Athens' } }
```

Порядок перечисления кортежей произвольный. Вот другой пример (в отличие от предыдущего, это не литерал):

```
RELATION { tx1 , tx2 , tx3 }
```

Я предполагаю, что *tx1*, *tx2* и *tx3* – выражения-кортежи одного и того же типа. Как видно из этих примеров, вызов селектора отношения в **Tutorial D** в общем случае включает ключевое слово RELATION, за которым следует список разделенных запятыми выражений-кортежей в скобках.

Следствия, вытекающие из определений

Большая часть свойств отношений, о которых я рассказывал в главе 1, являются прямыми следствиями приведенных выше определений, но о некоторых моментах я раньше не говорил вообще, а на других хочу остановиться подробнее. Первые два свойства, о которых я хочу упомянуть, таковы:

- Отношения никогда не содержат кортежей-дубликатов, потому что тело отношения – это множество (кортежей), а математические множества не содержат повторяющихся элементов.
- Отношения никогда не содержат null-значений, потому что тело отношения – это множество кортежей, а, как мы видели раньше, кортежи не могут содержать null-значений.

Но эти два аспекта настолько важны, и мне надо сказать о них так много, что я отложу детальное обсуждение до следующей главы. А в следующих нескольких разделах я рассмотрю ряд не столь существенных вопросов.

Отношения и их тела

Первым делом я хочу обсудить такой тезис: *любое подмножество тела отношения само является телом* (менее строго, любое подмножество отношения является отношением). В частности, поскольку пустое множество является подмножеством любого множества, то тело отношения может состоять из пустого множества кортежей (мы называем его *пустым отношением*). Предположим, к примеру, что в данный момент нет ни одной поставки. Тогда текущим значением переменной-отношения SP будет пустое отношение «поставки», которое можно изобразить, как показано на рисунке ниже (я снова возвращаюсь к соглашению о том, что в неформальном контексте имена типов не показываются в заголовке; далее в книге я буду то включать, то опускать имена типов – в зависимости от преследуемых целей):

SNO	PNO	QTY

Отметим, что для любого заданного типа отношения существует ровно одно пустое отношение этого типа, однако пустые отношения разных типов различаются – именно тем, что принадлежат разным типам. Например, пустое отношение «поставщики» не равно пустому отношению «детали» (тела равны, а заголовки нет).

Рассмотрим отношение, изображенное на следующем рисунке:

SNO	PNO	QTY
S1	P1	300

Это отношение содержит всего один кортеж (по-другому можно сказать, что его кардинальность равна 1). Чтобы обратиться к единственному содержащемуся в нем кортежу, нам необходимо как-то извлечь его из объемлющего отношения. В языке **Tutorial D** для этой цели предназначен синтаксис TUPLE FROM *rx* (где *rx* – любое выражение, обозначающее отношение кардинальности 1, например выражение RELATION {TUPLE {SNO 'S1', PNO 'P1', QTY 300}}, которое на самом деле представляет собой вызов селектора отношения). Напротив, в SQL применяется

приведение типа: если (а) *tx* – табличное выражение, которое используется как однострочный подзапрос (то есть встречается в том месте, где ожидается строчное выражение), то (б) предполагается, что таблица *t*, обозначаемая выражением *tx*, содержит всего одну строку *r* и (в) эта таблица приводится к строке *r*. Например:

```
SET S_ROW = ( S WHERE SNO = 'S1' ) ;
```

Нам также необходимо уметь проверять, что заданный кортеж *t* встречается в отношении *r*. В языке **Tutorial D** это выглядит так:

```
t ∈ r
```

Это выражение принимает значение TRUE, только если *t* встречается в *r*, и FALSE – в противном случае. Символом «∈» обозначается оператор *принадлежности множеству*, а выражение $t \in r$ произносится как «*t* входит в *r*». Вы, наверное, уже поняли, что «∈» – это по существу SQL-оператор IN с тем исключением, что в SQL левый операнд IN обычно является скаляром, а не строкой, то есть снова производится описанное выше приведение типа (скаляр приводится к типу содержащей его строки). Приведем пример («получить поставщиков, которые поставляют хотя бы одну деталь»):

```
SELECT SNO , SNAME , STATUS , CITY
FROM   S
WHERE  SNO IN ( SELECT SNO
                FROM   SP )
```

Отношения n-мерны

Я несколько раз подчеркивал, что хотя отношение можно изображать в виде таблицы, на самом деле это не таблица. (Еще раз повторю, что изображение предмета и сам предмет – вещи разные.) Конечно, представлять себе отношение как таблицу очень удобно, ведь таблицы так привычны. Более того, в главе 1 отмечалось, что именно тот факт, что отношение можно неформально рассматривать как таблицу – иногда даже откровенно «плоскую» или «двумерную» таблицу, – и делает реляционные системы такими простыми для понимания и использования и позволяет на интуитивном уровне рассуждать о поведении таких систем. Другими словами, то, что основная структура данных реляционной модели – отношение – имеет такое наглядное графическое представление, является весьма полезным свойством этой модели.

К сожалению, однако, многие люди, введенные в заблуждение обманчивой простотой графического представления, начинают полагать, что сами *отношения* являются «плоскими» или «двумерными». Это отнюдь не так. Раз отношение *r* имеет *n* атрибутов, то *каждый кортеж r представляет собой точку в n-мерном пространстве* (а отношение в целом оказывается множеством таких точек). Например, каждый из пяти кортежей в отношении «поставщики» представляет точку в 4-мерном

пространстве, поэтому отношение в целом можно назвать четырехмерным. Таким образом, отношения *n*-мерны, а не двумерны.[1] Как я уже писал в другой книге (и даже не в одной): «*Давайте все поклянемся никогда больше не говорить «плоские отношения»*.

Сравнение отношений

Для типов отношений, как и для любых других типов, должен быть определен оператор сравнения «=»; иначе говоря, если даны два отношения *rx* и *ry* одного и того же типа *T*, то, как минимум, должна иметься возможность определить, равны они или нет. Могут оказаться полезны и другие сравнения; например, иногда хочется проверить, является ли *rx подмножеством ry* (это означает, что каждый кортеж, принадлежащий *rx*, принадлежит также и *ry*) или *собственным* подмножеством *ry* (в этом случае каждый кортеж, принадлежащий *rx*, принадлежит также и *ry*, но в *ry* существует хотя бы один кортеж, не принадлежащий *rx*). Вот как записывается сравнение отношений на равенство в языке **Tutorial D**:

```
S { CITY } = P { CITY }
```

Здесь в левой части находится проекция поставщиков на CITY, а в правой – проекция деталей на CITY, и сравнение возвращает TRUE, если обе проекции совпадают, и FALSE – в противном случае. Иными словами, сравнение (которое является булевым выражением) отвечает на вопрос: совпадает ли множество городов, где находятся поставщики, с множеством городов, где складируются детали.

Вот еще один пример:

```
S { SNO } ⊃ SP { SNO }
```

Символ ⊃ означает «строго включает». Смысл этого сравнения таков: существует ли хотя бы один поставщик, не поставляющий никаких деталей?

К числу других полезных реляционных операторов относятся «⊇» (включает), «⊆» (включается) и «⊂» (строго включается). Очень часто возникает необходимость в сравнении «=» заданного отношения *rx* с пустым отношением того же типа; иначе говоря, требуется узнать, пусто ли *rx*. Для этой цели удобно определить сокращенную нотацию:

```
IS_EMPTY ( rx )
```

По определению, это выражение возвращает TRUE, если отношение, обозначенное реляционным выражением *rx*, пусто, и FALSE – в про-

[1] На самом деле я полагаю, что одна из причин, по которым мы так часто слышим о необходимости «многомерных баз данных» (в частности, для приложений поддержки принятий решений), как раз и состоит в том, что слишком многие не понимают, что отношения и так по природе своей многомерны.

тивном случае. Я буду постоянно пользоваться этим выражением в последующих главах (особенно в главе 8). Полезен и обратный оператор:

```
IS_NOT_EMPTY ( rx )
```

Логически это выражение эквивалентно NOT (IS_EMPTY(*rx*)).

TABLE_DUM и TABLE_DEE

Вспомните обсуждение кортежей выше в этой главе – мы говорили, что пустое множество является подмножеством любого множества, и, следовательно, существует такое понятие, как пустой кортеж (он также называется 0-кортежем), и, разумеется, понятие пустого заголовка. Раз так, то и отношение может иметь пустой заголовок, ведь заголовок – это просто множество атрибутов, и почему бы ему не быть пустым? Такое отношение имеет тип RELATION{}, а его степень равна нулю.

Итак, пусть *r* – отношение степени 0. Сколько существует таких отношений? Ответ: ровно два. Во-первых, *r* может быть пустым (то есть не содержит ни одного кортежа) – напомним, что существует ровно одно пустое отношение любого заданного типа. Во-вторых, если *r* не пусто, то оно может содержать только 0-кортежи. Но существует всего один 0-кортеж! – или, что то же самое, все 0-кортежи являются дубликатами друг друга, – поэтому *r* не может содержать более одного из них. Таким образом, действительно существует только два отношения без атрибутов: первое состоит из одного кортежа, а второе не содержит кортежей вовсе. По очевидным причинам я не буду изображать эти отношения на рисунках (на самом деле, это как раз тот случай, когда идея представлять отношения в виде таблиц оказывается совершенно несостоятельной).

Ну и что? – наверное, думаете вы. Для чего бы мне могло понадобиться отношение, не имеющее ни одного атрибута? Даже если с точки зрения математики они имеют полное право на существование (а это, конечно, так), то практической-то пользы от них чуть? Однако выясняется, что практическая польза есть, да еще какая, поэтому для этих отношений даже придуманы специальные названия: TABLE_DUM и TABLE_DEE, или просто DUM и DEE (DUM – пустое отношение, DEE – отношение с одним кортежем).[1] А причина в их *семантике*: FALSE (*нет*) для DUM

[1] Наверное, стоит немного сказать о происхождении этих названий. Во-первых, для читателей, не говорящих по-английски, объясню, что это игра слов: Tweedledum и Tweedledee первоначально были персонажами детского стишка, а потом Льюис Кэрролл включил их в свою сказку «Алиса в Зазеркалье» (в переводе Н. Демуровой они названы «Тру-ля-ля» и «Траля-ля»). Во-вторых, эти названия, пожалуй, не слишком удачны, так как обозначают как раз те отношения, которые невозможно изобразить в виде таблиц! Но они уже так давно используются в реляционной теории, что менять что-нибудь поздно.

и TRUE (*да*) для DEE. Это самая фундаментальная из всех возможных семантик. *Примечание*: Общий вопрос о семантике отношений я буду рассматривать в главах 5 и 6.

Кстати, есть неплохое мнемоническое правило, помогающее запомнить, что есть что: в словах DEE и «*yes*» есть буква «E», а в словах DUM и «*no*» ее нет.

Я еще не приводил в этой книге конкретных примеров использования отношений DUM и DEE, но впоследствии они появятся в изобилии. А сейчас упомяну лишь один момент, который поможет составить хотя бы интуитивное представление: оба эти отношения (особенно TABLE_DEE) играют в реляционной алгебре роль, аналогичную нулю в обычной арифметике. Все мы знаем, насколько важен нуль; на самом деле, трудно вообразить арифметику без него (древние римляне пытались, но далеко не ушли). Точно так же, трудно представить себе реляционную алгебру без TABLE_DEE. Что вновь возвращает нас к SQL... поскольку в SQL нет аналога 0-кортежа, в нем, очевидно (и к величайшему сожалению), нет и аналогов TABLE_DUM и TABLE_DEE.

Таблицы в SQL

Примечание

В этом разделе под термином *таблица* я буду понимать табличное значение – в смысле SQL, – а не табличную переменную (то, что создает предложение CREATE TABLE). Табличные переменные будут рассматриваться в главе 5.

В главе 2 я объяснял, что в SQL вообще нет никакого аналога понятию типа отношения; таблица в SQL считается состоящей из строк (в общем случае это мультимножество, а не множество строк), принадлежащих некоторому типу строки. Отсюда следует, что в SQL также нет ничего, аналогичного генератору типа RELATION, хотя поддерживаются другие генераторы типов, в частности, ARRAY, MULTISET и уже известный нам из главы 3 ROW. Однако же имеется нечто, именуемое *конструктором табличных значений*, и это можно в какой-то мере считать аналогом селектора отношения. Например:

```
VALUES ( 1, 2 ), ( 2, 1 ), ( 1, 1 ), ( 1, 2 )
```

Результатом вычисления этого выражения (в действительности это табличный литерал, хотя в SQL такой термин не употребляется) является таблица из четырех – не трех! – строк и двух столбцов. Причем эти столбцы не имеют имен. Как я уже объяснял, столбцы в SQL-таблице упорядочены слева направо; следовательно, эти столбцы можно идентифицировать (и часто так и делается) по порядковому номеру, а не по имени.

В качестве примера рассмотрим следующий вызов конструктора табличного значения:

```
VALUES ( 'S1' , 'Smith' , 20 , 'London' ) ,
       ( 'S2' , 'Jones' , 10 , 'Paris'  ) ,
       ( 'S3' , 'Blake' , 30 , 'Paris'  ) ,
       ( 'S4' , 'Clark' , 20 , 'London' ) ,
       ( 'S5' , 'Adams' , 30 , 'Athens' )
```

Заметим, что для того, чтобы это выражение можно было рассматривать как приближение к реляционному аналогу (то есть к реляционному литералу, обозначающему отношение, которое является текущим значением переменной-отношения S, показанной на рис. 1.3), мы должны:

1. Для каждого столбца таблицы, определяемой выражением VALUES, убедиться, что все его значения имеют подходящий тип. (В частности, если некоторый реляционный атрибут соответствует *i*-й позиции в какой-то из заданных строк, то необходимо проверить, что соответствует *i*-й позиции во всех строках).
2. Убедиться, что одна и та же строка не задана дважды.

Примечание

Как вы уже знаете, в реляционной модели заголовок является множеством атрибутов. Напротив, в SQL, где столбцы упорядочены слева направо, заголовок было бы правильнее рассматривать как последовательность, а не множество атрибутов (точнее, столбцов). Но если вы будете следовать рекомендациям, предлагаемым в настоящей книге, то это логическое различие в большинстве (?) случаев можно будет игнорировать.

А как насчет операторов присваивания и сравнения для таблиц? Табличное присваивание – обширная тема, детали которой я отложу до глав 5 и 7. Что касается сравнения таблиц, то в SQL прямой поддержки этой операции нет, но существуют обходные пути. Вот, например, как выглядит аналог реляционного сравнения в SQL:

```
NOT EXISTS ( SELECT CITY FROM S
             EXCEPT
             SELECT CITY FROM P )
AND
NOT EXISTS ( SELECT CITY FROM P
             EXCEPT
             SELECT CITY FROM S )
```

А вот аналог сравнения S{SNO} ⊃ SP{SNO}:

```
EXISTS ( SELECT SNO FROM S
         EXCEPT
         SELECT SNO FROM SP )
AND
NOT EXISTS ( SELECT SNO FROM SP
             EXCEPT
             SELECT SNO FROM S )
```

Именование столбцов в SQL

В реляционной модели (а) любой атрибут любого отношения имеет имя (то есть анонимные атрибуты не допускаются) и (б) эти имена должны быть уникальны в пределах одного отношения (то есть дубликаты имен запрещены). В SQL такие правила тоже действуют, но не всегда. Точнее, они проверяются для таблиц, являющихся значениями табличных переменных, – определенных с помощью CREATE TABLE или CREATE VIEW – но не для таблиц, которые получаются в результате вычисления некоторого табличного выражения[1]. **Настоятельная рекомендация:** Применяйте спецификацию AS, чтобы назначить подходящие имена столбцам, которые в противном случае (а) вообще не имели бы имен или (б) имели бы неуникальные имена. Вот несколько примеров:

```
SELECT DISTINCT SNAME , 'Supplier' AS TAG
FROM   S

SELECT DISTINCT SNAME , 2 * STATUS AS DOUBLE_STATUS
FROM   S

SELECT MAX ( WEIGHT ) AS MBW
FROM   P
WHERE  COLOR = 'Blue'

CREATE VIEW SDS AS
       SELECT DISTINCT SNAME , 2 * STATUS AS DOUBLE_STATUS
       FROM   S ;

SELECT DISTINCT S.CITY AS SCITY , P.CITY AS PCITY
FROM   S , SP , P
WHERE  S.SNO = SP.SNO
AND    SP.PNO = P.PNO

SELECT TEMP.*
FROM ( S JOIN P ON S.CITY > P.CITY ) AS TEMP
     ( SNO , SNAME , STATUS , SCITY ,
       PNO , PNAME , COLOR , WEIGHT , PCITY )
```

Разумеется, на эту рекомендацию можно не обращать внимания, если не возникает необходимости ссылаться на анонимные столбцы или столбцы с неуникальными именами. Например, третий из приведен-

[1] Неоспоримо, что в этом случае SQL не проверяет имена столбцов на дубликаты. Однако нельзя сказать, что он вообще не проверяет наличие анонимных столбцов; если столбцу явно не присвоено имя, то предполагается, что реализация выберет для него имя, уникальное в пределах объемлющей таблицы, но в остальном поведение зависит от реализации. На практике, однако, между «зависит от реализации» и «не определено» разница невелика (см. главу 12). Поэтому можно, не слишком отклоняясь от истины, назвать такие столбцы анонимными.

ных выше примеров можно было бы в некоторых случаях (скажем, во фразе WHERE или HAVING) безопасно сократить до:

```
SELECT MAX ( WEIGHT )
FROM   P
WHERE  COLOR = 'Blue'
```

Отметим еще, что этой рекомендации вообще невозможно следовать в случае таблиц, заданных выражениями VALUES.

Важное замечание: Есть много ситуаций, когда операторы реляционной алгебры полагаются на правильное именование атрибутов. Например, в главе 6 мы увидим, что оператор UNION требует, чтобы его операнды имели одинаковые заголовки (а, стало быть, и имена атрибутов); результат также будет иметь такой заголовок. Одно из достоинств такого подхода заключается как раз в том, что он позволяет избежать сложностей, связанных с зависимостью от порядковой позиции! Поэтому, если вы хотите использовать SQL реляционно, то должны применять это правило ко всем аналогам реляционных операторов в SQL. **Настоятельная рекомендация**: В качестве предварительного условия для навязывания такой дисциплины старайтесь давать одинаковые имена столбцам, представляющим «одну и ту же информацию». (Вот почему в базе данных о поставщиках и деталях оба столбца, содержащих номер поставщика, названы SNO, а не, скажем, SNO в одной таблице и SNUM в другой.) Наоборот, если два столбца представляют семантически различную информацию, лучше давать им разные имена.

Единственный случай, когда этой рекомендации невозможно следовать, – это когда в одной таблице есть два столбца, представляющих семантически одинаковую информацию. Рассмотрим, к примеру, таблицу EMP, в которой имеются столбцы «номер служащего» и «номер начальника», причем номер начальника сам является номером какого-то другого служащего. Эти столбцы по необходимости должны называться по-разному, например ENO и MNO. Поэтому иногда приходится прибегать к переименованию столбцов, как в следующем примере:

```
( SELECT ENO , MNO FROM EMP ) AS TEMP1
  NATURAL JOIN
( SELECT ENO AS MNO , ... FROM EMP ) AS TEMP2
/* где «...» столбцы EMP, отличные от ENO и MNO */
```

Если вы хотите использовать SQL в реляционном духе, то такое переименование имеет смысл производить и тогда, когда столбцы изначально были названы неудачно (то есть вам попалась база данных, спроектированная кем-то другим). В таком случае советую подумать о следующей стратегии:

- Для каждой базовой таблицы определите представление с идентичной структурой, отличающееся разве что тем, что некоторые столбцы переименованы.

- Убедитесь, что определенные таким образом представления подчиняются описанному выше правилу именования столбцов.
- Работайте с представлениями, а не с базовыми таблицами (отмечу, что такие представления, безусловно, допускают обновление).

К сожалению, невозможно полностью игнорировать тот факт, что в SQL столбцы имеют порядковые номера. (Разумеется, именно поэтому SQL может мириться с анонимными столбцами и столбцами с одинаковыми именами.) Отметим, в частности, что столбцы в SQL имеют порядковые номера даже тогда, когда в этом нет никакой необходимости (то есть когда у всех столбцов имеются приемлемые имена); это наблюдение в равной мере относится к столбцам базовых таблиц и представлений. **Настоятельная рекомендация**: Не пишите на SQL код, зависящий от порядка столбцов. В качестве примеров ситуаций, в которых SQL придает существенное значение упорядочению, назовем следующие:

- SELECT *
- JOIN, UNION, INTERSECT, EXCEPT – особенно, если в последних трех случаях отсутствует ключевое слово CORRESPONDING (см. главу 6)
- В списке имен столбцов, если он задан, вслед за определением переменной кортежа (range variable) (см. главу 12)
- В списке имен столбцов, если он задан, в CREATE VIEW (см. главу 9)
- INSERT, если не задан список имен столбцов
- Сравнения с модификаторами ALL и ANY, если степени операндов больше 1 (см. главу 11)
- Выражения VALUES

Заключительные замечания

В этой главе я дал точные определения фундаментальным понятиям кортежа и отношения. Как я уже говорил, поначалу эти определения могут немного обескуражить, но надеюсь, что, прочитав первые две главы, вы смогли в них разобраться. Я также объяснил, что такое типы кортежа и отношения, селекторы и сравнения, и рассмотрел ряд важных следствий, вытекающих из этих определений (в частности, вкратце описал важные отношения TABLE_DUM и TABLE_DEE). Кроме того, я рассказал об аналогах всех этих понятий в SQL в тех случаях, когда такие аналоги существуют. В заключение я хотел бы подчеркнуть важность рекомендаций (в разделе, непосредственно предшествующем этому) касательно именования столбцов в SQL. В последующих главах эти рекомендации будут активно применяться.

Упражнения

Упражнение 3.1. Дайте точные определения терминам *атрибут*, *тело*, *кардинальность*, *степень*, *заголовок*, *отношение*, *тип отношения* и *кортеж*.

Упражнение 3.2. Сформулируйте настолько точно, насколько сможете, что означают выражения: (а) два кортежа равны, (б) два отношения равны.

Упражнение 3.3. Напишите на языке **Tutorial D** вызовы селекторов кортежей для типичного кортежа из (а) переменной-отношения «детали», (б) переменной-отношения «поставки». Также покажите аналоги этих вызовов селекторов на SQL, если таковые существуют.

Упражнение 3.4. Напишите на языке **Tutorial D** вызов селектора отношения. Также покажите аналог этого вызова селектора на SQL, если таковой существует.

Упражнение 3.5. (*По существу, это повторение упражнения 1.8 из главы 1, но теперь вы знаете достаточно, чтобы дать более полный ответ.*) Существует много различий между отношением и таблицей. Перечислите столько, сколько сможете.

Упражнение 3.6. Атрибуты кортежа могут иметь произвольный тип (ну почти произвольный; можете ли вы назвать какие-нибудь исключения?). Приведите пример (а) кортежа, атрибутом которого является кортеж; (б) кортежа, атрибутом которого является отношение (RVA-атрибут).

Упражнение 3.7. Приведите пример отношения с (а) одним RVA-атрибутом, (б) двумя RVA-атрибутами. Также приведите еще два отношения, которые представляют ту же самую информацию, не используя RVA-атрибутов. Также приведите пример отношения с RVA-атрибутом такого, что не существует отношения, которое представляет ту же самую информацию, не используя RVA-атрибута.

Упражнение 3.8. Объясните своими словами смысл отношений TABLE_DUM and TABLE_DEE. Почему они не поддерживаются в SQL?

Упражнение 3.9. TABLE_DEE означает TRUE, а TABLE_DUM – FALSE. Означает ли это, что можно обойтись без обычного типа данных BOOLEAN? Кроме того, DEE и DUM – отношения, а не переменные-отношения. Как вы думаете, имеет ли смысл определять переменную-отношение степени 0?

Упражнение 3.10. В чем заключается логическое различие (и существует ли оно) – в отличие от очевидного семантического различия – между следующими двумя выражениями SQL:

```
VALUES ( 1, 2 ), ( 2, 1 ), ( 1, 1 ), ( 1, 2 )

VALUES ( ( 1, 2 ), ( 2, 1 ), ( 1, 1 ), ( 1, 2 ) )
```

Упражнение 3.11. Каков точный смысл следующего выражения SQL?

```
SELECT SNO
FROM   S
WHERE  ( NOT ( ( STATUS , SNO ) <= ( 20 , 'S4' ) ) ) IS NOT FALSE
```

Упражнение 3.12. Объясните своими словами, что понимается под многомерностью отношений.

Упражнение 3.13. Перечислите сколько сможете ситуаций, в которых SQL придает большое значение упорядочению столбцов слева направо.

Упражнение 3.14. Напишите на языке SQL аналог выражения IS_NOT_EMPTY(*rx*) языка **Tutorial D**.

Упражнение 3.15. Сформулируйте своими словами максимально точно описанную в тексте главы дисциплину именования столбцов в SQL.

Упражнение 3.16. Дисциплина именования столбцов, о которой упоминается в предыдущем упражнении, опирается на использование спецификаций AS. Но такие спецификации могут встречаться в нескольких разных контекстах. Более того, иногда синтаксис имеет вид «*X* AS . . .», а иногда «. . . AS *X*», причем само ключевое слово AS в одних случаях обязательно, а в других – нет[1]. Перечислите все контексты, в которых может встречаться слово AS, указав точную форму: «*X* AS . . . » или «. . . AS *X*» и уточнив, обязательно ли ключевое слово.

[1] Фактически по этой причине я всегда явно включаю ключевое слово, даже если оно необязательно. Трудно запомнить, когда ключевые слова в SQL должны присутствовать, а когда их можно опустить. Да и в любом случае довольно странно, особенно в случае AS, говорить о «фразе AS» или о «спецификации AS», когда никакого AS нет.

4

Нет дубликатам, нет null-значениям

В предыдущей главе я писал следующее (приблизительно):

- Отношения никогда не содержат кортежей-дубликатов, поскольку тело отношения – это множество (кортежей), а математические множества не содержат дубликатов.
- Отношения никогда не содержат null-значений, поскольку тело отношения – это множество кортежей, а кортежи не могут содержать null-значений.

Я также отметил, что, поскольку это обширные темы, то было бы лучше посвятить им отдельную главу. Вот эта глава. *Примечание*: По определению, эти вопросы имеют касательство к SQL, а не к реляционной теории, поэтому ниже я буду употреблять терминологию, принятую в SQL, а не в реляционной модели (по крайней мере, в большинстве случаев). Как вы вскоре увидите, я также применяю упрощенную, хотя и не нуждающуюся в особых пояснениях нотацию для строк.

Чем плохи дубликаты?

Существует много практических аргументов в пользу того, что строки-дубликаты (для краткости просто «дубликаты») следует запретить. Здесь я хочу остановиться только на одном – на мой взгляд, самом убедительном.[1] Однако он опирается на некоторые понятия, которые еще не обсуждались, поэтому я вынужден сделать несколько допущений:

---

[1] Один из рецензентов был уверен в том, что имеется даже более убедительный практический аргумент (на самом деле, самый практический из всех возможных) – дубликаты просто не находят отражения в реальности, а база данных, в которой дубликаты допускаются, спроектирована неправильно и не может быть тем, что в главе 1 я назвал «верной моделью реальности». Мне эта позиция очень близка. Но эта книга посвящена не проектированию

1. Я предполагаю, что вам известно о том, что все СУБД включают так называемый *оптимизатор*, в задачу которого входит отыскание наилучшего способа реализации пользовательских запросов (здесь под словом «наилучший» понимается *наилучший по производительности*).
2. Я предполагаю, что вам также известно о том, что оптимизаторы иногда выполняют так называемое *переписывание запроса*. Это процедура заключается в трансформации одного реляционного выражения *exp1* (представляющего, к примеру, запрос пользователя) в другое выражение *exp2* таким образом, что вычисление обоих выражений гарантированно дает одинаковый результат, но характеристики производительности *exp2* лучше (по крайней мере, мы рассчитываем на это). *Примечание*: Однако имейте в виду, что термин *переписывание запроса* применяется в некоторых продуктах в более ограниченном смысле.

Вот теперь я могу изложить свой аргумент. В его основе лежит тот факт, что некоторые трансформации выражений, а стало быть, и оптимизации, которые были бы допустимы, если бы SQL был по-настоящему реляционной системой, оказываются некорректны при наличии дубликатов. В качестве примера рассмотрим (нереляционную) базу данных, изображенную на рис. 4.1. Попутно отметим, что таблицы в этой базе данных не имеют ключей (на рисунке нет двойного подчеркивания). И кстати: если вы думаете, что такая база нереалистична – и особенно если считаете, что уже по одной этой причине последующие аргументы вас не убедят, – прочтите, пожалуйста, замечания по поводу этого примера в начале следующего раздела

P

PNO	PNAME
P1	Screw
P1	Screw
P1	Screw
P2	Screw

SP

SNO	PNO
S1	P1
S1	P1
S1	P2

Рис. 4.1. *Нереляционная база данных с дубликатами*

Прежде чем двигаться дальше, я хотел бы задать один вопрос: «Что могло бы означать наличие в таблице P трех строк (P1,Screw), а не двух, четырех или, скажем, семнадцати?» Что-то ведь это должно означать, иначе зачем вообще понадобились дубликаты? Как-то Тед Кодд сказал:

баз данных, и в любом случае вопрос о дубликатах связан не только с проектированием. Поэтому я попытаюсь показать, какие проблемы возникают из-за дубликатов, вне зависимости от того, вызваны они неудачным проектированием или иными причинами. Детальный анализ вопроса в целом, включая и аспекты, относящиеся к проектированию, можно найти в статье «Double Trouble, Double Trouble» (см. приложение D).

«Если некое утверждение является истинным, то оно не станет более истинным, если повторить его дважды»[1].

Поэтому я должен допустить, что какой-то смысл у этого дублирования есть, хотя он явно и не выражен. (Отмечу в скобках, что именно поэтому дубликаты идут вразрез с одной из исходных целей реляционной модели – принципом *явности*, в соответствии с которым данные должны быть выражены максимально явно, поскольку предполагается, что речь идет о базе данных, используемой совместно. Наличие дубликатов наводит на мысль о том, что какая-то часть семантики скрыта.) Другими словами, если предположить, что у дубликатов есть какой-то смысл, то на основе того факта, что в таблице P присутствуют именно три строки (P1,Screw), а не две, не четыре и не семнадцать, должны приниматься какие-то бизнес-решения. Ибо если это не так, то (повторюсь) зачем эти дубликаты понадобились?

Теперь рассмотрим такой запрос: получить номера тех деталей, которые либо являются болтами (screw), либо поставляются поставщиком S1, либо то и другое вместе. Ниже приведены некоторые возможные формулировки этого запроса на SQL вместе с результатами:

1.
```
SELECT P.PNO
FROM   P
WHERE  P.PNAME = 'Screw'
OR     P.PNO IN
     ( SELECT SP.PNO
       FROM   SP
       WHERE  SP.SNO = 'S1' )
```

Результат: P1 * 3, P2 * 1.

2.
```
SELECT SP.PNO
FROM   SP
WHERE  SP.SNO = 'S1'
OR     SP.PNO IN
     ( SELECT P.PNO
       FROM   P
       WHERE  P.PNAME = 'Screw' )
```

Результат: P1 * 2, P2 * 1.

3.
```
SELECT P.PNO
FROM   P, SP
WHERE  ( SP.SNO = 'S1' AND
         SP.PNO = P.PNO )
OR     P.PNAME = 'Screw'
```

[1] Как-то я процитировал это высказывание на одном семинаре, и один из слушателей сказал: «Это можно и повторить!», на что я ответил: «Да, между логикой и риторикой существует логическое различие».

Результат: P1 * 9, P2 * 4.

4.
```
SELECT SP.PNO
FROM  P, SP
WHERE  ( SP.SNO = 'S1' AND
         SP.PNO = P.PNO )
OR     P.PNAME = 'Screw'
```

Результат: P1 * 8, P2 * 4.

5.
```
SELECT P.PNO
FROM   P
WHERE  P.PNAME = 'Screw'
UNION  ALL
SELECT SP.PNO
FROM   SP
WHERE  SP.SNO = 'S1'
```

Результат: P1 * 5, P2 * 2.

6.
```
SELECT DISTINCT P.PNO
FROM   P
WHERE  P.PNAME = 'Screw'
UNION  ALL
SELECT SP.PNO
FROM   SP
WHERE  SP.SNO = 'S1'
```

Результат: P1 * 3, P2 * 2.

7.
```
SELECT P.PNO
FROM   P
WHERE  P.PNAME = 'Screw'
UNION  ALL
SELECT DISTINCT SP.PNO
FROM   SP
WHERE  SP.SNO = 'S1'
```

Результат: P1 * 4, P2 * 2.

8.
```
SELECT P.PNO
FROM   P
WHERE  P.PNAME = 'Screw'
OR     P.PNO IN
     ( SELECT SP.PNO
       FROM   SP
       WHERE  SP.SNO = 'S1' )
```

Результат: P1 * 3, P2 * 1.

9.
```
SELECT DISTINCT SP.PNO
FROM   SP
WHERE  SP.SNO = 'S1'
OR     SP.PNO IN
     ( SELECT P.PNO
```

```
        FROM   P
        WHERE  P.PNAME = 'Screw' )
```

Результат: P1 * 1, P2 * 1.

10.
```
SELECT   P.PNO
FROM     P
GROUP    BY P.PNO, P.PNAME
HAVING   P.PNAME = 'Screw'
OR       P.PNO IN
       ( SELECT SP.PNO
         FROM SP
         WHERE SP.SNO = 'S1' )
```

Результат: P1 * 1, P2 * 1.

11.
```
SELECT P.PNO
FROM   P, SP
GROUP  BY P.PNO, P.PNAME, SP.SNO, SP.PNO
HAVING ( SP.SNO = 'S1' AND
         SP.PNO = P.PNO )
OR     P.PNAME = 'Screw'
```

Результат: P1 * 2, P2 * 2.

12.
```
SELECT P.PNO
FROM   P
WHERE  P.PNAME = 'Screw'
UNION
SELECT SP.PNO
FROM   SP
WHERE  SP.SNO = 'S1'
```

Результат: P1 * 1, P2 * 1.

(На самом деле, некоторые из приведенных выше формулировок – какие? – вызывают подозрение, потому что в них предполагается, что любой болт поставляется хотя бы одним поставщиком. Но для последующего рассуждения этот факт не играет роли.)

Прежде всего, мы видим, что двенадцать разных формулировок дают девять различных результатов – различных именно в плане *объема дублирования*. (Кстати, я не утверждаю, что эти двенадцать формулировок и девять результатов включают в себя все возможные варианты; в действительности, вообще-то это не так.) Таким образом, если пользователя действительно интересуют дубликаты, то он должен очень тщательно подходить к формулировке запроса, если хочет получить желаемый результат.

Далее, аналогичные замечания относятся и к самой системе: поскольку различие в формулировке может приводить к различным результатам, оптимизатору необходимо проявлять крайнюю осторожность в деле трансформации выражений. Например, оптимизатор не впра-

ве трансформировать формулировку 1 в формулировку 3 или наоборот, даже если захотел бы. Иначе говоря, строки-дубликаты становятся серьезным *препятствием на пути оптимизации*. Вот некоторые следствия из этого факта.

- Код самого оптимизатора становится труднее писать и сопровождать, и более вероятно появление в нем ошибок – в совокупности это удорожает продукт, делает его менее надежным и увеличивает сроки выхода на рынок.
- Производительность системы оказывается ниже, чем могла бы быть.
- В решение проблем производительности вовлекаются пользователи. Точнее, им приходится тратить больше времени и сил на то, чтобы сформулировать запрос так, чтобы добиться оптимальной производительности, – а это как раз то, что реляционная модель призвана устранить.

То, что наличие дубликатов препятствует оптимизации, особенно обидно, если принять во внимание, что в большинстве случае пользователям *безразлично*, сколько дубликатов окажется в результате. Другими словами:

- Различные формулировки запроса могут порождать различные результаты.
- Но с точки зрения пользователя эти различия, скорее всего, несущественны.
- *Однако* оптимизатор об этом не знает и потому вынужден без всякой необходимости воздерживаться от трансформаций, которые могли бы принести эффект.

Исходя из подобных примеров у меня возникает искушение сделать вывод о том, что во всех случаях следует исключать дубликаты из результатов запроса – например, с помощью ключевого слова DISTINCT, – что позволит навсегда забыть о проблеме (и, если вы будете следовать этой рекомендации, то вообще не останется причин для заведения в базе дубликатов!). Однако в следующем разделе я еще кое-что скажу об этой рекомендации.

Дубликаты: новые проблемы

Можно было бы еще очень долго говорить о дубликатах и их зловредности, но я ограничусь лишь тремя замечаниями. Прежде всего, можно с полным на то основанием возразить, что на практике, по крайней мере, в базовых таблицах дубликатов обычно не бывает, и потому приведенный выше пример интуитивно кажется неубедительным. В общем-то, правильно; но проблема в том, что SQL может *порождать* дубликаты в результатах запроса. Более того, разные формулировки одного и того же запроса могут порождать результаты с разным объемом дублирования, даже если в исходных таблицах никаких дублика-

тов нет. Вот, например, две формулировки запроса «получить номера тех поставщиков, которые поставляют хотя бы одну деталь» (отметим, что в исходных таблицах дубликатов нет):

```
SELECT SNO                    |    SELECT SNO
FROM   S                      |    FROM   S NATURAL JOIN SP
WHERE  SNO IN                 |
     ( SELECT SNO             |
       FROM   SP )            |
```

По меньшей мере одно из этих выражений в общем случае порождает результат, содержащий дубликаты. (*Упражнение*: какие результаты дают эти выражения на наших тестовых данных?) Поэтому если вам не нравится считать таблицы на рис. 4.1 базовыми, не надо; считайте, что они получены в результате выполнения показанных выше запросов, при этом все остальные рассуждения остаются в силе.

Далее, существует еще один психологический довод против дубликатов, который лично мне кажется весьма убедительным (спасибо Джонатану Геннику (Jonathan Gennick), который его предложил): если, следуя идее о многомерности отношений (см. главу 3), представить себе таблицу, как график в n-мерном пространстве, то дубликаты ничего не добавляют, – просто одна и та же точка наносится дважды.

И последний аргумент. Предположим, что таблица T допускает дубликаты. Тогда невозможно отличить «истинные» дубликаты от тех, что возникли в результате ошибки при вводе данных! Например, что произойдет, если человек, отвечающий за ввод данных, случайно – то есть по ошибке – введет в T одну и ту же строку дважды? (Кстати, SQL легко позволяет сделать такую ошибку.) Существует ли способ удалить «вторую» строку, оставив первую? Заметим, что удалить желательно именно «вторую» строку, потому что она вообще не должна была присутствовать.

Как избежать дубликатов в SQL

В реляционной модели дубликаты запрещены, поэтому если мы хотим использовать SQL в реляционном духе, то должны принять меры к тому, чтобы дубликаты не появлялись. Прежде всего, если в каждой базовой таблице имеется хотя бы один ключ (см. главу 5), то в базовых таблицах полных дубликатов не окажется в принципе. Однако, как отмечалось в предыдущем разделе, некоторые SQL-выражения все равно могут порождать результаты, содержащие дубликаты. Вот перечень некоторых случаев, когда это может происходить:

- SELECT ALL
- UNION ALL
- VALUES (т. е. вызовы конструкторов табличных значений)

О VALUES см. главу 3. По поводу ALL отметим для начала, что это ключевое слово (и альтернативное ему DISTINCT) может встречаться:

- Во фразе SELECT, сразу после ключевого слова SELECT.
- В операциях объединения, пересечения и разности, сразу после соответствующего ключевого слова (UNION, INTERSECT и EXCEPT соответственно).
- Внутри скобок при вызове «функции множества», например SUM, непосредственно перед выражением-аргументом.

Примечание

Слово DISTINCT подразумевается по умолчанию для UNION, INTERSECT и EXCEPT; во всех остальных случаях по умолчанию подразумевается слово ALL.

Случай «функции множества» особый; вы должны задать ALL, хотя бы неявно, если хотите, чтобы функция учитывала дубликаты, а обычно так и есть (см. главу 7). Но в остальных случаях необходимо позаботиться об исключении дубликатов, и это нужно делать всегда, по крайней мере в принципе, если вы собираетесь использовать SQL реляционно. Таким образом, к этим случаям применимы очевидные рекомендации: всегда указывайте слово DISTINCT, и лучше явно, и никогда не указывайте ALL. Тогда о строках-дубликатах можно будет забыть.

Однако на практике все не так просто. Почему? Полагаю, что лучше всего будет привести цитату из моей предыдущей книги:

> В этом месте рукописи я добавил, что если вы считаете, что каждый раз задавать слово DISTINCT утомительно, то жалуйтесь не мне, а разработчикам SQL. Однако рецензенты почти единодушно ужаснулись моему предложению повсеместно вставлять DISTINCT. Один из них написал: «Те, кто хорошо знает SQL, были бы шокированы, если бы SELECT DISTINCT подразумевалось писать по умолчанию». На это я со всем почтением хочу возразить, что (а) те, кто «был бы шокирован», по-видимому, знают конкретную реализацию, а не SQL, и (б) шок, вероятно, объясняется тем, что эта реализация плохо справляется с оптимизацией излишних DISTINCT.[1] Если я пишу запрос SELECT DISTINCT SNO FROM S ..., то DISTINCT можно безопасно игнорировать. Если я пишу EXISTS (SELECT DISTINCT ...) или IN (SELECT DISTINCT ...), то DISTINCT тоже можно игнорировать. И если я пишу SELECT DISTINCT SNO FROM SP ... GROUP BY SNO, то и тогда DISTINCT можно игнорировать. Как и в случае SELECT DISTINCT ... UNION

[1] Это означает, что для выполнения SELECT DISTINCT может потребоваться больше времени, чем для SELECT ALL, пусть даже DISTINCT в конкретном случае – «пустая операция». Что ж, такое возможно, не буду долго распространяться на эту тему, замечу лишь, что обычно реализация не способна устранить излишние DISTINCT на этапе оптимизации, потому что не понимает, как работает механизм *вывода ключей* (то есть не может определить, какие ключи будут применимы к результату произвольного табличного выражения).

> SELECT DISTINCT ... И так далее. Так почему я, как пользователь, должен тратить свое время и силы на то, чтобы разбираться, когда некоторый DISTINCT отрицательно скажется на производительности, а когда его можно спокойно опустить? И при этом помнить все детали разномастных правил SQL о том, когда дубликаты автоматически удаляются, а когда нет?
>
> Я мог бы продолжать и дальше. Однако решил – вопреки собственному суждению, но в интересах сохранения добрых отношений (со своими рецензентами, я имею в виду) – не следовать своему же совету нигде в этой книге, кроме тех мест, где требование устранить дубликаты логически необходимо. Кстати, решить, когда это действительно так, не всегда было просто. Но зато теперь я могу присоединить свой голос к хору тех, кто недоволен поставщиками.

Итак, **рекомендация** (как это ни печально) сводится вот к чему: во-первых, убедитесь, что вы знаете, когда SQL устраняет дубликаты, не испрашивая вашего согласия. Во-вторых, в тех случаях, когда об этом нужно явно просить, проверьте, изменится ли что-нибудь, если вы не попросите. В-третьих, если это что-то может изменить, указывайте DISTINCT (но, как однажды заметил Хью Дарвен, относитесь к этому с опаской). И ни при каких обстоятельствах не указывайте ALL!

Чем плохи null-значения?

Первый абзац раздела «Чем плохи дубликаты?» можно было бы с равным успехом вставить и сюда, с одним небольшим отличием, поэтому я повторю его почти буквально. Существует много практических аргументов в пользу того, что null-значения следует запретить. Здесь я хочу остановиться только на одном – на мой взгляд, самом убедительном. Однако он опирается на некоторые понятия, которые еще не обсуждались, поэтому я вынужден сделать несколько допущений:

1. Я предполагаю, что вам известно о том, что результатом любого сравнения, в котором хотя бы одна из сравниваемых величин равна null, является значение истинности UNKNOWN, а не TRUE и не FALSE. Обоснованием такого решения служит тот факт, что null интерпретируется как *неизвестное* значение, но если значение A неизвестно, то неизвестно также, будет ли $A > B$, каким бы ни было значение B. Отмечу попутно, что именно из этого факта проистекает термин *трехзначная логика* (3VL): понятие null в том смысле, в котором оно трактуется в SQL, неизбежно приводит к логике, в которой есть три значения истинности вместо двух. (Напротив, реляционная модель построена на базе традиционной двузначной логики, 2VL.)
2. Я также предполагаю, что вы знакомы с таблицами истинности трехзначной логики для стандартных логических операторов, или *связок*, NOT, AND и OR (T = TRUE, F = FALSE, U = UNKNOWN):

p	NOT *p*
T	F
U	U
F	T

p	*q*	*p* AND *q*
T	T	T
T	U	U
T	F	F
U	T	U
U	U	U
U	F	F
F	T	F
F	U	F
F	F	F

p	*q*	*p* OR *q*
T	T	T
T	U	T
T	F	T
U	T	T
U	U	U
U	F	U
F	T	T
F	U	U
F	F	F

Отметим, в частности, что NOT возвращает UNKNOWN, если его операндом является UNKNOWN; AND возвращает UNKNOWN, если один операнд равен UNKNOWN, а второй – либо UNKNOWN, либо TRUE; OR возвращает UNKNOWN, если один операнд равен UNKNOWN, а второй – либо UNKNOWN, либо FALSE.

Теперь я готов изложить свой аргумент. Фундаментальное наблюдение заключается в том, что некоторые логические выражения – и, стало быть, некоторые запросы – порождают результаты, правильные с точки зрения трехзначной логики, но неправильные в реальном мире. В качестве примера рассмотрим (нереляционную) базу данных, показанную на рис. 4.2, в которой для детали P1 атрибут CITY «равен» null. Обратите внимание на пустое место там, где должно было бы находиться значение CITY, – оно обозначает *отсутствие чего-либо*; концептуально в этой позиции *нет вообще ничего* – ни даже последовательности пробелов или пустой строки (то есть «кортеж» для детали P1 на самом деле кортежем не является, к этому моменту я еще вернусь в конце раздела).

S

SNO	CITY
S1	London

P

PNO	CITY
P1	

Рис. 4.2. *Нереляционная база данных с дубликатами*

Рассмотрим теперь следующий (признаю, искусственный) запрос к базе данных на рис. 4.2: «получить пары (SNO,PNO), в которых либо города поставщика и детали различаются, либо городом детали не является Париж (либо то и другое вместе)». Вот как выглядит формулировка этого запроса на SQL:

```
SELECT S.SNO, P.PNO
FROM   S, P
WHERE  S.CITY <> P.CITY
OR     P.CITY <> 'Paris'
```

Я хочу обратить ваше внимание на булево выражение во фразе WHERE:

```
( S.CITY <> P.CITY ) OR ( P.CITY <> 'Paris' )
```

(Скобки добавлены для ясности.) Для тех данных, что есть в нашей базе, это выражение сводится к UNKNOWN OR UNKNOWN, что равно UNKNOWN. Но SQL-запросы отбирают те данные, для которых выражение во фразе WHERE принимает значение TRUE, а не FALSE и не UNKNOWN; следовательно, в данном примере ничего не будет выбрано.

Однако же в реальности деталь P1 все же находится в каком-то городе[1]; иными словами, то, что для детали P1 атрибут CITY содержит null, означает, что там есть какое-то, пусть неизвестное, значение, скажем, *xyz*. Но *xyz* – это либо Париж, либо нет. Если это Париж, то выражение

```
( S.CITY <> P.CITY ) OR ( P.CITY <> 'Paris' )
```

принимает вид (для тех данных, что у нас есть)

```
( 'London' <> 'Paris' ) OR ( 'Paris' <> 'Paris' )
```

и его вычисление дает TRUE, поскольку первый член равен TRUE. Наоборот, если *xyz* – не Париж, то выражение принимает такой вид (опять-таки для имеющихся данных)

```
( 'London' <> xyz ) OR ( xyz <> 'Paris' )
```

и, стало быть, снова равно TRUE, так как второй член равен TRUE. Таким образом, в реальном мире это булево выражение всегда равно TRUE, и запрос должен был бы вернуть пару (S1,P1) *независимо от того, что стоит за null*. Иначе говоря, логически правильный результат (с точки зрения трехзначной логики) и результат, правильный в реальном мире, различаются!

В качестве другого примера рассмотрим следующий запрос к той же таблице P, которая показана на рис. 4.2 (я не стал приводить его первым, потому что он еще более искусственный, чем предыдущий, но в некотором смысле – еще более убедительный):

```
SELECT PNO
FROM   P
WHERE  CITY = CITY
```

Если говорить о реальном мире, то ответом, конечно же, будет множество номеров деталей, присутствующих в P (другими словами, множество, состоящее только из номера детали P1, если говорить о данных, представленных на рис. 4.2). Однако SQL не вернет ничего.

Подведем итоги. Если в базе данных имеются null-значения, то на некоторые вопросы вы будете получать неверные ответы. Хуже того, в общем случае невозможно узнать, ответы на какие вопросы верны, а на

[1] Должна находиться, потому что в противном случае эта деталь не удовлетворяла бы *предикату переменной-отношения*. Это важное положение, но я еще не готов объяснить или даже толком сформулировать его. Дальнейшее обсуждение см. в главе 5.

какие нет; подозрительными становятся все результаты. *Вы в принципе не можете доверять ответам, полученным для базы данных, содержащей null-значения.* На мой взгляд, такое положение дел абсолютно неприемлемо.

Отступление

Не могу противиться искушению добавить ко всему вышесказанному еще и то, что хотя SQL поддерживает трехзначную логику и даже поддерживает ключевое слово UNKNOWN, это ключевое слово – в отличие от TRUE и FALSE – не является значением типа BOOLEAN! (Это один из многочисленных изъянов в поддержке трехзначной логики в SQL; есть еще великое множество других, но они выходят за рамки настоящей книги.) Другими словами, тип BOOLEAN по-прежнему содержит всего два значения; «третье значение истинности» представлено – совершенно некорректно – с помощью null. Вот какие следствия вытекают из этого факта:

- Присваивание значения UNKNOWN переменной X типа BOOLEAN фактически превращает X в null.
- После такого присваивания сравнение X = UNKNOWN дает не TRUE, а null (то есть SQL то ли действительно «верит», то ли «притворяется, что верит», будто находится в неведении относительно того, что X равно UNKNOWN).
- На самом деле сравнение X = UNKNOWN всегда дает null (читай UNKNOWN) вне зависимости от значения X, потому что логически оно эквивалентно сравнению «X = NULL» (хотя это и некорректный синтаксис).

Чтобы в полной мере осознать всю серьезность подобных изъянов, поразмыслите об аналогии с числовым типом, в котором используется null вместо нуля для представления нуля.

Как и о дубликатах, я мог бы еще много чего сказать по поводу null-значений, но хотел бы закончить обзором формальных возражений против них. Напомню, что, по определению, null не является значением. Отсюда следует, что:

- «Тип», который содержит null, не является типом (так как типы содержат значения).
- «Кортеж», который содержит null, не является кортежем (так как кортежи содержат значения).
- «Отношение», которое содержит null, не является отношением (так как отношения содержат кортежи, а кортеж не может содержать null).
- Фактически, null-значения нарушают самый фундаментальный из реляционных принципов, а именно *принцип информации* (см. приложение А).

Итак, получается, что если допустить существование null-значений, то мы вообще не можем говорить о реляционной модели (не знаю, о чем мы

ведем речь, но это точно не реляционная модель); все здание рушится, и *всякая определенность теряется*.

Как избежать null–значений в SQL

Реляционная модель запрещает null-значения; поэтому если мы хотим использовать SQL реляционно, то должны принять меры к тому, чтобы их не было. Прежде всего, для любого столбца во всех базовых таблицах следует задавать ограничение NOT NULL, явно или неявно (см. главу 5); тогда null-значения в базовых таблицах не появятся. К сожалению, однако, SQL-выражения все равно могут порождать таблицы, содержащие null. Вот несколько ситуаций, когда null-значения могут возникать.

- Все «функции множеств» в SQL, например SUM, возвращают null, если аргументом является пустое множество (за исключением COUNT и COUNT(*), которые корректно возвращают нуль).
- Если результатом скалярного подзапроса является пустая таблица, то эта таблица приводится к null.
- Если однострочный подзапрос дает пустую таблицу, то эта таблица приводится к строке, содержащей в каждом столбце null. *Примечание*: Строка из одних null-значений и null-строка – логически вовсе не одно и то же (кстати, вот еще одно логическое различие), и тем не менее SQL полагает, по крайней мере в некоторых случаях, что они неразличимы. Но попытка разобраться в таких тонкостях завела бы нас слишком далеко.
- Внешние соединения и соединения объединением (union join) изначально спроектированы так, что результат включает null-значения.
- Если в выражении CASE опущена фраза ELSE, предполагается, что фраза ELSE имеет вид ELSE NULL.
- Если $x = y$, то выражение NULLIF(x,y) возвращает null.
- Оба «ссылочных триггерных действия» ON DELETE SET NULL и ON UPDATE SET NULL порождают null-значения (по понятным причинам).

Настоятельные рекомендации:

- Задавайте ограничение NOT NULL, явное или неявное, для всех столбцов во всех базовых таблицах.
- Не используйте ключевое слово NULL ни в каких других контекстах (то есть нигде, кроме ограничения NOT NULL).
- Не используйте ключевое слово UNKNOWN вообще ни в каких контекстах.

- Не опускайте фразу ELSE в выражении CASE, если нет уверенности, что она никогда не получит управления.
- Не используйте функцию NULLIF.
- Не пользуйтесь внешними соединениями и не употребляйте ключевых слов OUTER, FULL, LEFT и RIGHT (кроме, быть может, случаев, описанных в разделе «Замечание о внешнем соединении» ниже).
- Не используйте соединение объединением.
- Не употребляйте ключевых слов PARTIAL и FULL в спецификаторе MATCH (они имеют смысл только при наличии null-значений). По тем же причинам не используйте спецификатор MATCH в ограничениях внешнего ключа и избегайте употребления IS DISTINCT FROM. (В отсутствие null-значений выражение *x* IS DISTINCT FROM *y* эквивалентно выражению $x <> y$.)
- Не используйте предикаты IS TRUE, IS NOT TRUE, IS FALSE и IS NOT FALSE. Причина в том, что если *bx* – булево выражение, то при наличии null-значений следующие эквиваленции ложны:

  ```
  bx IS TRUE       ≡  bx
  bx IS NOT TRUE   ≡  NOT bx
  bx IS FALSE      ≡  NOT bx
  bx IS NOT FALSE  ≡  bx
  ```

  Другими словами, IS TRUE и ему подобные при наличии null-значений только вводят в заблуждение.
- Наконец, используйте функцию COALESCE для любого скалярного выражения, которое в противном случае могло бы «принять значение null». (Приношу извинения за кавычки, но фраза «принимать значение null» внутренне противоречива.)

Последний пункт нуждается в дополнительных пояснениях. По сути дела, оператор COALESCE позволяет заменить null значением, отличным от null, «прямо в момент возникновения» (то есть до того, как появился шанс нанести заметный вред). Приведем его определение. Пусть *x*, *y*, ..., *z* – скалярные выражения. Тогда выражение COALESCE (*x*,*y*,...,*z*) возвращает null, если все его аргументы равны null, а в противном случае – значение первого аргумента, отличного от null. Конечно же, рекомендуется делать так, чтобы хотя бы один из аргументов *x*, *y*, ..., *z* был отличен от null. Например:

```
SELECT S.SNO , ( SELECT COALESCE ( SUM ( ALL QTY ) , 0 )
                 FROM   SP
                 WHERE  SP.SNO = S.SNO ) AS TOTQ
FROM   S
```

В этом примере, если результат вызова SUM «принимает значение null» – а так будет, в частности, для любого поставщика, для которого нет ни одной поставки, – то COALESCE заменит null на нуль. В при-

менении к нашим тестовым данным этот запрос, следовательно, вернет такой результат:

SNO	TOTQ
S1	1300
S2	700
S3	200
S4	900
S5	0

Замечание о внешнем соединении

Внешнее соединение специально задумано так, чтобы результат содержал в некоторых столбцах null-значения, поэтому, вообще говоря, его лучше избегать. Говоря реляционным языком, это что-то вроде брака поневоле: он навязывает таблицам некий род объединения – да-да, я имею в виду именно объединение, а не соединение, – даже тогда, когда рассматриваемые таблицы не удовлетворяют обычным условиям, разрешающим объединение (см. главу 6). Делается это, по существу, путем дополнения одной или обеих таблиц null-значениями еще до выполнения объединения, в результате чего они начинают удовлетворять стандартным требованиям, например:

```
SELECT SNO , PNO
FROM   SP
UNION
SELECT SNO , 'nil' AS PNO
FROM   S
WHERE  SNO NOT IN ( SELECT SNO FROM SP )
```

Результат получается такой (обратите особое внимание на последнюю строчку):

SNO	PNO
S1	P1
S1	P2
S1	P3
S1	P4
S1	P5
S1	P6
S2	P1
S2	P2
S3	P2
S4	P2
S4	P4
S4	P5
S5	nil

Тот же результат можно было бы получить с помощью SQL-оператора внешнего соединения в сочетании с COALESCE:

```
SELECT SNO , COALESCE ( PNO , 'nil' ) AS PNO
FROM ( S NATURAL LEFT OUTER JOIN SP ) AS TEMP
```

Заключительные замечания

Напоследок я хочу сделать еще несколько замечаний касательно null-значений и трехзначной логики. То и другое задумывалось как решение проблемы «отсутствующей информации», но полагаю, мне удалось убедить вас, что если они и могут считаться «решением», то крайне неудачным, приводящим к разрушительным последствиям. Однако прежде чем закрыть эту тему, я хотел бы привести аргумент, который часто приходится слышать в этой связи, и возразить на него. Аргумент звучит примерно так.

> Все приводимые вами примеры ситуаций, когда null-значения оказываются причиной неверных ответов, искусственны. В реальной практике таких запросов не бывает! И вообще, ваша критика по большей части академическая и теоретическая – бьюсь об заклад, что вы не сможете продемонстрировать реальную ситуацию, в которой null-значения порождают те проблемы, о которых вы так печетесь, и готов поспорить, что вы не сумеете доказать, что на практике такие ситуации действительно встречаются.

Понятно, что у меня есть несколько возражений против этого аргумента. Первое таково: откуда мы знаем, что null-значения *никогда не приводили* к серьезным практическим ошибкам? Мне кажется, что если бы было обнаружено, что какая-то серьезная реальная проблема – разлитие нефти, обрушение моста, неверный медицинский диагноз – произошла по вине null-значений, то нашлись бы веские причины (не технического свойства), по которым такая информация не вышла бы наружу. Все мы наслышаны о досадных сбоях, вызванных программными ошибками других видов, даже безо всяких null-значений; на мой взгляд, последние могут лишь повысить вероятности таких сбоев.

Во-вторых, предположим, кто-то (к примеру, я сам) объявил о том, что некий программный продукт или приложение содержит серьезную логическую ошибку из-за null-значений. Только представьте, с каким воодушевлением его притянут к суду.

В-третьих, и это самое важное, я полагаю, что те из нас, кто выступает с критикой null-значений, вообще не должны находиться в оборонительной позиции. Я полагаю, что мы должны услышать контраргументы противника. В конце концов, нельзя отрицать, что в некоторых случаях null-значения приводят к ошибкам. Поэтому не наше дело доказывать, что в число этих «некоторых случаев» попадают и реальные, встречающиеся на практике ситуации; пусть те, кто защищает null-значения, доказывают, что такого не бывает. Рискну добавить, что лично мне кажется, что доказать такую вещь будет очень трудно, если вообще возможно.

Конечно, если запретить null-значения, то проблему отсутствующей информации придется решать какими-то другими способами. К сожалению, эти способы слишком сложны, чтобы подробно обсуждать их

здесь. В простых случаях можно использовать имеющийся в SQL механизм значений по умолчанию (отличных от null). Но боюсь, что за всесторонним исследованием этой тематики, в том числе за объяснением того, как при желании все же можно получить ответ «не знаю» даже от базы данных без null-значений, вам придется обратиться к публикациям, перечисленным в приложении D.

Упражнения

Упражнение 4.1. «Дубликаты нужны в базах данных, потому что естественным образом встречаются в реальном мире. Например, все копейки – дубликаты друг друга». Как бы вы возразили против такого аргумента?

Упражнение 4.2. Пусть *r* – некое отношение, и пусть *bx* и *by* – булевы выражения. Тогда существует правило (оно используется в реляционных системах для оптимизации), согласно которому (*r* WHERE *bx*) UNION (*r* WHERE *by*) ≡ *r* WHERE *bx* OR *by* (где символом ≡ обозначается эквиваленция). Применимо ли это правило, если *r* – не отношение, а таблица, содержащая дубликаты?

Упражнение 4.3. Пусть *a*, *b* и *c* – множества. Тогда дистрибутивный закон пересечения относительно объединения (тоже используется в реляционных системах для оптимизации) утверждает, что *a* INTERSECT (*b* UNION *c*) ≡ (*a* INTERSECT *b*) UNION (*a* INTERSECT *c*). Выполняется ли этот закон, если *a*, *b* и *c* – не множества, а мультимножества?

Упражнение 4.4. В той части стандарта SQL, где описывается фраза FROM (в выражениях SELECT - FROM - WHERE), написано:

> Результатом <clause from> (фразы from) является ... декартово произведение таблиц, указанных в <table reference> (ссылках на таблицы) [в этой фразе <from clause>]. Декартово произведение, *CP*, представляет собой мультимножество всех строк *r*, таких, что *r* есть конкатенация строк, взятых из каждой из указанных таблиц...

Обратите внимание, что *CP* определено некорректно! – несмотря на то, что стандарт далее говорит: «кардинальность *CP* равна произведению кардинальностей указанных таблиц». Рассмотрим, к примеру, следующие таблицы T1 и T2:

T1

C1
0
0

T2

C2
1
2

Отметим, что любая из следующих таблиц удовлетворяет приведенному выше определению декартова произведения *CP* таблиц T1 и T2 (то есть и та, и другая могут быть взяты в качестве упоминаемого «мультимножества):

CP1

C1	C2
0	1
0	1
0	2
0	2

CP2

C1	C2
0	1
0	2
0	2
0	2

Можете ли вы исправить приведенную в стандарте формулировку?

Упражнение 4.5. Рассмотрим следующее определение курсора:

```
DECLARE X CURSOR FOR SELECT SNO , QTY FROM SP ;
```

Обратите внимание, что (а) курсор X допускает обновление, (б) таблица, видимая посредством курсора X, допускает дубликаты, но (в) базовая таблиц SP дубликатов не содержит. Предположим теперь, что выполняется операция DELETE ... WHERE CURRENT OF X. Тогда в общем случае невозможно сказать, какая конкретно строка таблицы SP была удалена. Как бы вы решили эту проблему?

Упражнение 4.6. Пожалуйста, напишите «гугол» (10^{100}) раз: никаких дубликатов не существует.

Упражнение 4.7. Как вы думаете, есть ли для null-значений естественное место в реальном мире?

Упражнение 4.8. Существует логическое различие между null и третьим значением истинности. Это утверждение истинно или ложно? (Быть может, следовало бы задать вопрос так: истинно, ложно или неизвестно?)

Упражнение 4.9. В тексте главы я привел таблицы истинности для одной одноместной (NOT) и двух двуместных (AND и OR) связок трехзначной логики, но есть много других связок (см. следующее упражнение). В частности, весьма полезна одноместная связка MAYBE со следующей таблицей истинности:

p	MAYBE *p*
T	F
U	T
F	F

Поддерживает ли такую связку SQL?

Упражнение 4.10. Продолжая предыдущее упражнение, ответьте, сколько всего связок имеется в двузначной логике (2VL)? А в трехзначной (3VL)? Какой вывод можно сделать из ответа на эти два вопроса?

Упражнение 4.11. Логика называется *функционально полной*, если она поддерживает прямо или косвенно все возможные связки. Функциональная полнота – чрезвычайно важное свойство; логика, не удовлетворяющая этому условию, была бы похожа на арифметику, в ко-

торой отсутствуют некоторые операции, например «+». Является ли логика 2VL функционально полной? А логика 3VL?

Упражнение 4.12. Пусть *bx* – булево выражение. Тогда *bx* OR NOT *bx* – тоже булево выражение, и в логике 2VL оно гарантированно дает значение TRUE (это пример того, что в логике называется *тавтологией*). Является ли это выражение тавтологией в 3VL? Если нет, то существует ли в 3VL аналогичная тавтология?

Упражнение 4.13. Пусть, как и в предыдущем упражнении, *bx* – булево выражение. Тогда *bx* AND NOT *bx* – тоже булево выражение, и в логике 2VL оно гарантированно дает значение FALSE (это пример того, что в логике называется *противоречием*). Является ли это выражение противоречием в 3VL? Если нет, то существует ли в 3VL аналогичное противоречие?

Упражнение 4.14. В 2VL *r* JOIN *r* равно *r*, а операторы INTERSECT и TIMES являются частными случаями JOIN (см. главу 6). Остаются ли эти наблюдения истинными в 3VL?

Упражнение 4.15. Следующее выражение является допустимым вызовом конструктора значения в SQL: ROW (1,NULL). Представляет ли эта строка null или не-null?

Упражнение 4.16. Пусть *bx* – булево SQL-выражение. Тогда NOT (*bx*) и (*bx*) IS NOT TRUE – тоже булевы SQL-выражения. Эквивалентны ли они?

Упражнение 4.17. Пусть *x* – SQL-выражение. Тогда x IS NOT NULL и NOT (x IS NULL) – булевы SQL-выражения. Эквивалентны ли они?

Упражнение 4.18. Пусть DEPT и EMP – SQL-таблицы, и пусть столбец DNO есть в обеих таблицах, а столбец ENO – только в EMP. Рассмотрим выражение DEPT.DNO = EMP.DNO AND EMP.DNO = 'D1' (это выражение могло бы быть, например, частью фразы WHERE). «Хороший» оптимизатор мог бы трансформировать это выражение в DEPT.DNO = EMP.DNO AND EMP.DNO = 'D1' AND DEPT.DNO = 'D1', исходя из того, что если $a = b$ и $b = c$, то $a = c$ (см. упражнение 6.13 в главе 6). Но корректна ли такая трансформация? Если нет, то почему? И какие отсюда следуют выводы?

Упражнение 4.19. Предположим, что в базе данных о поставщиках и деталях разрешены null-значения. Рассмотрим тогда следующий запрос к этой базе, который по причинам, выходящим за рамки данной книги, выражен не на SQL, а на некоторой не вполне правильной форме реляционного исчисления (см. главу 10):

```
S WHERE NOT EXISTS SP ( SP.SNO = S.SNO AND SP.PNO = 'P2' )
```

Что означает этот запрос? И является ли следующая запись его эквивалентной формулировкой?

```
S WHERE NOT ( S.SNO IN ( SP.SNO WHERE SP.PNO = 'P2' ) )
```

Упражнение 4.20. Пусть *k1* и *k2* – значения одного и того же типа. Что тогда означают следующие утверждения в SQL?

a. *k1* и *k2* «одинаковы» с точки зрения сравнения, например во фразе WHERE.

b. *k1* и *k2* «одинаковы» с точки зрения уникальности ключей.

c. *k1* и *k2* «одинаковы» с точки зрения исключения дубликатов.

Упражнение 4.21. В тексте главы я сказал, что оператор UNION ALL может порождать дубликаты. А что можно сказать об операторах INTERSECT ALL и EXCEPT ALL?

Упражнение 4.22. Являются ли рекомендации «Всегда используйте DISTINCT» и «Никогда не используйте ALL» дубликатами?

Упражнение 4.23. Если TABLE_DEE соответствует TRUE (*да*), TABLE_DUM – FALSE (*нет*), то что соответствует UNKNOWN (*может быть*)?

5

Базовые переменные-отношения, базовые таблицы

К этому моменту вы уже должны понимать, что между *отношениями-значениями* (для краткости просто *отношения*) и переменными-отношениями имеется существенное логическое различие. Теперь пришло время взглянуть на это различие поближе; точнее, мы рассмотрим те аспекты, которые характерны именно для переменных-отношений, но не для отношений. *Предостережение*: К сожалению, части последующего обсуждения, относящиеся к SQL, несколько путаны, поскольку в SQL эти понятия четко не различаются – один и тот термин *таблица* иногда обозначает табличное значение, а иногда – табличную переменную. Например, ключевое слово TABLE в предложении CREATE TABLE, очевидно, относится к табличной переменной, но когда мы говорим, например, что таблица S содержит пять строк, то ясно, что слова «таблица S» относятся к табличному значению (то есть к текущему значению табличной переменной с именем S). Поэтому остерегайтесь путаницы в этом вопросе.

Позвольте также напомнить следующее:

- Во-первых, переменная-отношение – это такая переменная, значениями которой являются отношения, и именно к переменным-отношениям, а не к отношениям, применяются операции INSERT, DELETE и UPDATE (точнее, операции реляционного присваивания, – напомним, что INSERT, DELETE и UPDATE – не более чем сокращенная запись некоторых реляционных присваиваний).
- Далее, если R – переменная-отношение, а r – отношение, которое нужно присвоить R, то R и r должны иметь одинаковый тип (отношения).
- Наконец, термины *заголовок*, *тело*, *атрибут*, *кортеж*, *кардинальность* и *степень*, формально определенные в главе 3 для отношений,

можно очевидным образом интерпретировать и для переменных-отношений.

В настоящей главе мы будем иметь дело с базовыми переменными-отношениями (базовыми таблицами в терминологии SQL); фактически не будет особого вреда, если далее вы будете предполагать, что все переменные-отношения являются базовыми переменными-отношениями, коль скоро явно не оговорено противное. В главе 9 рассматриваются аспекты, касающиеся только виртуальных переменных-отношений, или представлений.

Определения данных

В качестве основы для примеров я буду использовать следующие определения в базе данных о поставщиках и деталях (слева приведено определение на языке **Tutorial D**, а справа – на SQL, и такого представления я буду придерживаться в большинстве примеров в этой главе):

```
VAR S BASE RELATION          |    CREATE TABLE S
  { SNO    CHAR ,            |      ( SNO    VARCHAR(5)   NOT NULL ,
    SNAME  CHAR ,            |        SNAME  VARCHAR(25)  NOT NULL ,
    STATUS INTEGER ,         |        STATUS INTEGER      NOT NULL ,
    CITY   CHAR }            |        CITY   VARCHAR(20)  NOT NULL ,
  KEY { SNO } ;              |        UNIQUE ( SNO ) ) ;

VAR P BASE RELATION          |    CREATE TABLE P
  { PNO    CHAR ,            |      ( PNO    VARCHAR(6)   NOT NULL ,
    PNAME  CHAR              |        PNAME  VARCHAR(25)  NOT NULL ,
    COLOR  CHAR ,            |        COLOR  CHAR(10)     NOT NULL ,
    WEIGHT FIXED ,           |        WEIGHT NUMERIC(5,1) NOT NULL ,
    CITY   CHAR }            |        CITY   VARCHAR(20)  NOT NULL ,
  KEY { PNO } ;              |        UNIQUE ( PNO ) ) ;

VAR SP BASE RELATION         |    CREATE TABLE SP
  { SNO    CHAR ,            |      ( SNO    VARCHAR(5)   NOT NULL ,
    PNO    CHAR ,            |        PNO    VARCHAR(6)   NOT NULL ,
    QTY    INTEGER }         |        QTY    INTEGER      NOT NULL ,
  KEY { SNO , PNO }          |        UNIQUE ( SNO , PNO ) ,
  FOREIGN KEY { SNO }        |        FOREIGN KEY ( SNO )
          REFERENCES S       |           REFERENCES S ( SNO ) ,
  FOREIGN KEY { PNO }        |        FOREIGN KEY ( PNO )
          REFERENCES P ;     |           REFERENCES P ( PNO ) ) ;
```

Обновление – это операция над множеством

Сразу же хочу подчеркнуть, что вне зависимости от используемого синтаксиса, реляционное присваивание – это *операция над множествами*. (Фактически все операции в реляционной модели производятся над множествами, то есть принимают в качестве операндов отношения или

переменные-отношения целиком, а не отдельные кортежи). Таким образом, INSERT вставляет множество кортежей в конечную переменную-отношение, DELETE удаляет множество кортежей из переменной-отношения, а UPDATE обновляет множество кортежей в переменной-отношении. Конечно, мы часто говорим, например, об обновлении одного кортежа, но вы должны понимать, что:

- Такое словоупотребление означает лишь, что кардинальность обновляемого множества кортежей равна единице.
- Более того, обновление множества кортежей кардинальности 1 иногда невозможно в принципе.

Предположим, например, что на переменную-отношение S наложено ограничение целостности (см. главу 8), согласно которому поставщики S1 и S4 всегда должны находиться в одном городе. Тогда любая операция «UPDATE одного кортежа», которая попытается изменить город только для одного из этих двух поставщиков, неизбежно завершится неудачно. Нам придется изменять одновременно обоих поставщиков, например, так:

```
UPDATE S                          |    UPDATE S
WHERE SNO = 'S1'                  |    SET    CITY = 'New York'
   OR SNO = 'S4' :                |    WHERE  SNO = 'S1'
    { CITY := 'New York' } ;      |    OR     SNO = 'S4' ;
```

В данном примере обновляется именно множество, состоящее из двух кортежей.

Из сказанного выше вытекает, что в реляционной модели нет никакого соответствия «позиционным обновлениям» в SQL (то есть предложениям UPDATE или DELETE «WHERE CURRENT OF cursor»), так как это, по определению, операции над кортежами (точнее, над строками), а не над множествами. В современных SQL-продуктах они по большей части работают нормально, но лишь потому, что эти продукты не очень хорошо поддерживают ограничения целостности. Если они когда-нибудь будут улучшены в этом отношении, то «позиционные обновления» могут и перестать работать, то есть приложения, которые сегодня функционируют, завтра могут отказать – не слишком приятная ситуация, на мой взгляд. **Рекомендация**: Не пользуйтесь в SQL обновлениями на курсоре, если нет абсолютной уверенности, что проблема, подобная той, что описана выше, никогда не возникнет. (Я говорю это, полностью осознавая тот факт, что в настоящее время большая часть обновлений в SQL производится на курсоре.)

А теперь я должен кое в чем сознаться. Дело в том, что фраза «обновление кортежа» – или, точнее, множества кортежей – в любом случае очень неточна (если не сказать небрежна). Вспомните определения *значения* и *переменной* из главы 1. Если *V* – субъект обновления, то *V* должна быть переменной, по определению, но кортежи (как и отношения) являются значениями и не могут быть обновлены – тоже по определению.

Говоря об обновлении кортежа *t1* до *t2* в некоторой переменной-отношении *R*, мы имеем в виду замену кортежа *t1* в *R* другим кортежем *t2*. Но даже такое словоупотребление есть небрежность! – на самом деле мы заменяем отношение *r1*, являющееся начальным значением *R*, другим отношением *r2*. А что такое здесь отношение *r2*? Объясняю: пусть *s1* и *s2* – отношения, содержащие только кортеж *t1* и *t2* соответственно; тогда *r2* равно (*r1* MINUS *s1*) UNION *s2*. Другими словами, «обновление кортежа *t1* до *t2* в переменной-отношении *R*» можно рассматривать как две последовательные операции: сначала удаление *t1*, а затем вставку *t2*, – если (вопреки всему, что я сказал) вы позволите мне так неформально говорить об удалении и вставке отдельных кортежей.

Точно так же не имеет смысла говорить об «удалении атрибута *A* из кортежа *t*» – или из отношения *r*, или даже из переменной-отношения *R*. Конечно, на практике мы так говорим, потому что это удобно (и позволяет избежать многословия), но, как и в случае с дружественной к пользователю терминологией, о которой шла речь в главе 1, это приемлемо лишь при условии, что все мы понимаем, что это лишь приблизительное отображение истины, которое затемняет настоящую суть дела.

Проверка ограничений

Из того, что операторы обновления суть операции над множествами, следует среди прочего, что проверку ограничений целостности нельзя выполнять, пока обновление не будет завершено целиком (дополнительные соображения см. в главе 8). Иначе говоря, обновление множества нельзя рассматривать как последовательность обновлений индивидуальных кортежей (или строк в терминологии SQL). Я полагаю, что стандарт SQL согласуется с этим требованием, – а, может быть, и нет; в этой связи вызывают некоторое подозрение «триггеры на уровне строки» (см. следующий подраздел). Как бы то ни было, даже если стандарт согласуется с теорией, это еще не значит, что с ней согласуются и все продукты, поэтому всегда нужно быть настороже.

Триггерные действия

Из того, что операторы обновления суть операции над множествами, вытекает еще одно важное следствие: «ссылочные триггерные действия», например ON DELETE CASCADE (см. раздел «Еще о внешних ключах» ниже в этой главе), да и любые другие триггерные действия, тоже нельзя выполнять, пока обновление не будет завершено целиком. Однако здесь, к сожалению, SQL трактует обновление множества как последовательность обновлений строк, по крайней мере (как уже отмечалось) в поддержке триггеров на уровне строки, а, может быть, и еще где-то. **Рекомендация**: Старайтесь избегать операций, принципиально относящихся к отдельным строкам. Конечно, эта рекомендация не запрещает операции над множествами кардинальности 1, как показано в следующем примере:

```
UPDATE S WHERE SNO = 'S5' :     |    UPDATE S
     { CITY := 'New York' } ;   |    SET    CITY = 'New York'
                                |    WHERE  SNO = 'S5' ;
```

Заключительное замечание

Итог обсуждения в этом разделе сводится к тому, что операции обновления, а фактически – все операции в реляционной модели, *семантически всегда атомарны*, то есть либо выполняются целиком, либо не производят никакого эффекта (за исключением, быть может, возврата кода состояния или чего-то в этом роде). Таким образом, хотя иногда мы неформально описываем операцию над множествами как сокращенное обозначение последовательности операций над кортежами, важно понимать, что все такие описания – лишь приблизительное отображение истины.

Реляционное присваивание

В общем случае реляционное присваивание производится путем присваивания отношения-значения, которое обозначается некоторым реляционным выражением, переменной-отношению, обозначаемой своим именем. Например:

```
S := S WHERE NOT ( CITY = 'Athens' ) ;
```

В главе 1 мы видели, что такое присваивание логически эквивалентно следующему предложению DELETE языка **Tutorial D**:

```
DELETE S WHERE CITY = 'Athens' ;
```

Более общо, предложение DELETE

```
DELETE R WHERE bx ;
```

(где R – имя переменной-отношения, а bx – булево выражение) является сокращенной записью следующего реляционного присваивания:

```
R := R WHERE NOT ( bx ) ;
```

Аналогично предложение INSERT языка **Tutorial D**

```
INSERT R rx ;
```

(где R – снова имя переменной-отношения, а rx – реляционное выражение) является сокращенной записью для:

```
R := R D_UNION rx ;
```

Примечание

Здесь D_UNION обозначает дизъюнктное объединение. Дизъюнктное объединение похоже на обычное объединение с тем отличием, что его операнды-отношения не должны иметь общих кортежей (см. главу 6).

Наконец, предложение UPDATE языка **Tutorial D** тоже соответствует некоторому реляционному присваиванию, но его детали несколько сложнее, чем для INSERT и DELETE, поэтому я отложу их рассмотрение до главы 7.

Табличное присваивание в SQL

Предложения INSERT, DELETE и UPDATE в SQL являются точными аналогами соответствующих предложений **Tutorial D**, и больше о них сказать почти нечего, разве что привести два мелких замечания по поводу INSERT. В SQL источник операции INSERT задается табличным выражением (часто, хотя и необязательно, выражением VALUES – см. главу 3). Таким образом, INSERT в SQL действительно вставляет таблицу, а не строку, хотя эта таблица может содержать всего одну строку или даже не содержать ни одной строки. Также INSERT в SQL поддерживает возможность задавать после имени таблицы заключенный в скобки список имен столбцов. **Рекомендация**: Всегда пользуйтесь этой возможностью. Например, следующая форма записи предложения INSERT

```
INSERT INTO SP ( PNO , SNO , QTY ) VALUES ( 'P6' , 'S5' , 700 ) ;
```

предпочтительнее такой:

```
INSERT INTO SP VALUES ( 'S5' , 'P6' , 700 ) ;
```

потому что во втором варианте мы полагаемся на упорядочение слева направо столбцов в таблице SP, а в первом – нет. Вот еще один пример:

```
INSERT INTO SP ( SNO , PNO , QTY ) VALUES ( 'S3' , 'P1' , 500 ) ,
                                          ( 'S2' , 'P5' , 400 ) ;
```

К сожалению, в SQL нет прямого аналога реляционному присваиванию как таковому. Самое большее, на что он способен в плане обобщенного присваивания

```
R := rx ;
```

– это следующая последовательность предложений:

```
DELETE FROM T ;
INSERT INTO T ( ... ) tx ;
```

(здесь T и tx – SQL-аналоги R и rx соответственно). Отметим, в частности, что такая последовательность предложений может завершиться неудачно в случае, когда ее реляционный аналог (то есть реляционное присваивание) завершился бы успешно, – например, если на таблицу T наложено ограничение, согласно которому она не должна быть пустой.

Принцип присваивания

В заключение этого раздела я хотел бы привлечь ваше внимание к принципу, который, хотя и очень прост, имеет далеко идущие послед-

ствия. Это *принцип присваивания*, который гласит, что после присваивания значения v переменной V результатом сравнения $v = V$ должно быть TRUE. *Примечание*: *Принцип присваивания* – это фундаментальный принцип, справедливый не только для реляционной модели, но и для информатики вообще. Конечно, он применим, в частности, и к реляционному присваиванию, но (повторюсь) распространяется на присваивание любого вида. На самом деле, его можно в какой-то мере считать определением присваивания, и я уверен, что вы это осознаете. Я еще вернусь к этой теме в главе 8 при обсуждении так называемого *множественного* присваивания.

Еще о потенциальных ключах

Основную идею потенциальных ключей я объяснил в главе 1, но сейчас хочу уточнить это понятие. Сначала определение.

> **Определение**: Пусть K – подмножество заголовка переменной-отношения R. Тогда K называется *потенциальным ключом* (или просто *ключом*) R, если оно обладает следующими свойствами:
>
> 1. *Уникальность*: никакое из возможных значений R не содержит двух разных кортежей с одинаковым значением K.
> 2. *Неприводимость*: *никакое* собственное подмножество K не обладает свойством уникальности.
>
> Если K состоит из n атрибутов, то n называется *степенью* K.

Свойство уникальности не требует пояснений, а вот о свойстве неприводимости я должен сказать несколько слов. Рассмотрим переменную-отношение S и множество атрибутов {SNO,CITY} – назовем его *SK* – которое, безусловно, является подмножеством S, обладающим свойством уникальности (никакое отношение, которое могло бы являться значением переменной-отношения S, не содержит двух разных кортежей с одним и тем же значением *SK*). Но свойством неприводимости оно не обладает, поскольку можно отбросить атрибут CITY, а оставшееся множество {SNO} все равно будет обладать свойством уникальности. Поэтому мы не рассматриваем множество *SK* как ключ, потому что оно «слишком велико». Напротив, множество {SNO} неприводимо и является ключом.

Почему мы хотим, чтобы ключи были неприводимы? Одна из важных причин такова: если бы мы задали «ключ», который не является неприводимым, то СУБД не смогла бы обеспечить выполнение нужного ограничения уникальности. Предположим, к примеру, что мы сказали СУБД (солгали!), что {SNO,CITY} – ключ. Тогда СУБД не смогла бы гарантировать, что номера поставщиков «глобально» уникальны, а обеспечила бы лишь более слабое ограничение «локальной» уникальности номеров, то есть уникальности в пределах одного города. По этой – но

не только по этой – причине мы требуем, чтобы ключи не содержали атрибутов, которые не нужны для уникальной идентификации. **Рекомендация:** В SQL никогда не обманывайте систему, определяя в качестве ключа комбинацию столбцов, которая заведомо не является неприводимой.

Все переменные-отношения, которые встречались нам до сих пор, имели только один ключ. А теперь для разнообразия приведем несколько примеров (для простоты только на языке **Tutorial D**) переменных-отношений с двумя и более ключами. Обратите внимание на перекрывающиеся ключи во втором и третьем примерах.

```
VAR TAX_BRACKET BASE RELATION
  { LOW MONEY , HIGH MONEY , PERCENTAGE INTEGER }
  KEY { LOW }
  KEY { HIGH }
  KEY { PERCENTAGE } ;

VAR ROSTER BASE RELATION
  { DAY DAY_OF_WEEK , TIME TIME_OF_DAY , GATE GATE , PILOT NAME }
  KEY { DAY , TIME , GATE }
  KEY { DAY , TIME , PILOT } ;

VAR MARRIAGE BASE RELATION
  { SPOUSE_A NAME , SPOUSE_B NAME , DATE_OF_MARRIAGE DATE }
    /* в предположении, что полигамии нет и ни одна пара не */
    /* сочетается между собой браком более одного раза ... */
  KEY { SPOUSE_A , DATE_OF_MARRIAGE }
  KEY { DATE_OF_MARRIAGE , SPOUSE_B }
  KEY { SPOUSE_B , SPOUSE_A } ;
```

Кстати, вы, возможно, заметили здесь мелкую несообразность. По идее, ключ должен представлять собой множество атрибутов, а атрибут – это пара имя-атрибута/имя-типа. И тем не менее в синтаксисе KEY в языке **Tutorial D** задаются только имена атрибутов, а не пары. Но такой синтаксис годится, потому что имена атрибутов уникальны в пределах заголовка, следовательно, соответствующие им имена типов заданы неявно. Аналогичные замечания применимы ко многим отрывкам на языке **Tutorial D**, и я больше не буду их повторять, предлагая отнести сказанное в этом абзаце ко всем остальным случаям.

Я хочу закончить этот раздел несколькими замечаниями на разные темы. Во-первых, отметим, что понятия ключа применяется к переменным-отношениям, а не к отношениям.[1] Почему? Потому что сказать,

[1] С другой стороны, можно сказать, что отношение удовлетворяет или не удовлетворяет некоторому ограничению ключа. Можно даже пойти дальше и сказать, пусть это и не совсем строго, что отношение, которое удовлетворяет заданному ограничению ключа, фактически «имеет» этот ключ, хотя такая манера выражаться может привести к путанице, и я бы ее не рекомендовал.

что нечто является ключом, – все равно, что сказать, что действует некое ограничение целостности – конкретно, ограничение уникальности, – а ограничения целостности применяются к переменным, а не к значениям. (По определению, ограничения целостности налагают ограничения на обновления, а обновления применяются к переменным, а не к значениям. Продолжение обсуждения см. в главе 8.)

Во-вторых, в случае базовых переменных-отношений принято, как отмечалось в главе 1, назначать некий ключ *первичным* (а все остальные ключи для рассматриваемой переменной-отношения тогда называются *альтернативными*). Но выбирать ли какой-то ключ в качестве первичного и, если да, то какой именно, – вопрос прежде всего психологический, лежащий за рамками самой реляционной модели. На практике большинство базовых переменных-отношений, пожалуй, должны иметь первичный ключ, но, повторюсь, это правило, если его можно назвать правилом, к реляционной теории касательства не имеет.

В-третьих, если R – переменная-отношение, то R имеет и даже обязана иметь по меньшей мере один ключ. Причина состоит в том, что любое возможное значение R есть отношение и, следовательно, по определению не содержит кортежей-дубликатов; поэтому уж по крайней мере комбинация всех атрибутов R – то есть всего заголовка, – безусловно, обладает свойством уникальности. Таким образом, либо сама эта комбинация также обладает свойством неприводимости, либо существует его собственное подмножество, обладающее этим свойством. В любом случае существует нечто, одновременно уникальное и неприводимое. *Примечание*: Эти замечания необязательно применимы к SQL-таблицам, поскольку таблицы в SQL могут содержать строки-дубликаты, а потому могут вовсе не иметь ключа. **Настоятельная рекомендация**: По крайней мере для базовых таблиц задавайте спецификаторы PRIMARY KEY и/или UNIQUE, чтобы у любой такой таблицы гарантированно был хотя бы один ключ.

В-четвертых, отмечу, что значения ключа – это *кортежи* (строки в SQL), а не скаляры. Например, в случае переменной-отношения S, у которой есть единственный ключ {SNO}, значением этого ключа для некоторого конкретного кортежа – скажем, для поставщика S1 – будет

```
TUPLE { SNO 'S1' }
```

(Вспомните главу 3, где говорилось о том, что любое подмножество кортежа само является кортежем.) Конечно, на практике мы обычно неформально говорим, что значением ключа в этом примере является просто S1 – или, точнее, 'S1', – но фактически это не так.

Кстати, теперь должно быть ясно, что ключи, как и многое другое в реляционной модели, в сильнейшей мере зависят от понятия *равенства кортежей*. Действительно, чтобы гарантировать соблюдение ограничения целостности, мы должны уметь определять, равны ли два ключа, а это как раз и есть вопрос о равенстве кортежей – даже тогда, когда,

как в случае переменной-отношения S, соответствующие кортежи имеют степень 1 и «выглядят» как простые скалярные значения.

В-пятых, пусть *SK* – подмножество заголовка переменной-отношения *R*, обладающее свойством уникальности, но не обязательно свойством неприводимости. Тогда *SK* – *суперключ R* (а суперключ, не являющийся ключом, называется *собственным* суперключом). Например, {SNO} и {SNO,CITY} – суперключи переменной-отношения S, причем последний – собственный. Отметим, что заголовок любой переменной-отношения *R* по определению является суперключом *R*.

И последнее замечание касается понятия *функциональной зависимости*. Сейчас я не хочу вдаваться в детали этого понятия, поскольку вернусь к нему в главе 8 и в приложении В, но, надо полагать, вы и так с ним знакомы. Я лишь хотел бы привлечь ваше внимание к следующему. Пусть *SK* – суперключ (а, быть может, и ключ) переменной-отношения *R*, и пусть *A* – произвольное подмножество заголовка *R*. Тогда *R* обязательно удовлетворяет функциональной зависимости

$$SK \rightarrow A$$

Поясню. В общем случае функциональная зависимость $SK \rightarrow A$ означает, что если для двух кортежей *R* значение *SK* одинаково, то и значение *A* для них тоже одинаково. Но если у двух кортежей одинаковое значение *SK*, где *SK* – суперключ, то, по определению, это *должен быть* один и тот же кортеж! – поэтому и значение *A* для них тоже одинаково. Иными словами, если говорить неформально, то всегда существует «стрелочка функциональной зависимости», направленная от любого суперключа (а значит, и от любого ключа) к любого другому подмножеству заголовка переменной-отношения. Повторю, что еще вернусь к этой теме в главе 8 и приложении В.

Еще о внешних ключах

Напомню (см. главу 1), что внешний ключ – это множество атрибутов одной переменной-отношения, значения которых должны совпадать со значениями некоторого потенциального ключа другой (или той же самой) переменной-отношения. Так, в базе данных о поставщиках и деталях {SNO} и {PNO} – внешние ключи SP, значения которых должны совпадать со значениями потенциального ключа {SNO} в S и значениями потенциального ключа {PNO} в P соответственно. (Говоря *должны совпадать*, я имею в виду, что если, например, переменная-отношение SP содержит кортеж, в котором SNO имеет значение S1, то переменная-отношение S также должна содержать кортеж, в котором SNO имеет значение S1, ибо в противном случае в SP присутствовала бы поставка, произведенная несуществующим поставщиком, и база данных не была бы «верной моделью реальности».) А теперь приведу более точное определение (еще раз обращаю внимание на зависимость от понятия равенства кортежей):

> **Определение:** Пусть *R1* и *R2* – переменные-отношения, не обязательно различные, и пусть *K* – ключ *R1*. Пусть *FK* – подмножество заголовка *R2* такое, что существует, возможно пустая, последовательность переименований атрибутов *R1*, которая переводит *K* в *K′*, где *K′* и *FK* содержат в точности одинаковые атрибуты. Пусть далее *R2* и *R1* в любой момент времени удовлетворяют ограничению, согласно которому для любого кортежа *t2* в *R2* значение *FK* совпадает со значением *K′* для некоторого (необязательно единственного) кортежа *t1* в *R1* в тот же момент времени. Тогда *FK* называет внешним ключом (той же степени, что и *K*), вышеупомянутое ограничение – *ограничением ссылочной целостности*, а *R2* и *R1* – соответственно *ссылающейся переменной-отношением* и *переменной-отношением, на которую указывает ссылка*, для этого ограничения.

Попутно отмечу, что в первоначальном варианте реляционной модели требовалось, чтобы внешние ключи соответствовали не какому-то ключу, а именно первичному ключу переменной-отношения, на которую указывает ссылка. Но так как мы не настаиваем на существовании первичных ключей, то, разумеется, не можем и не будем требовать, чтобы внешние ключи соответствовали только первичным (и SQL с такой позицией согласен).

Еще раз повторю, что в базе данных о поставщиках и деталях {SNO} и {PNO} – внешние ключи SP, ссылающиеся на единственный потенциальный ключ – на самом деле являющийся также и первичным, – S и P соответственно. Рассмотрим более сложный пример:

```
VAR EMP BASE RELATION            |  CREATE TABLE EMP
  { ENO CHAR ,                   |    ( ENO VARCHAR(6) NOT NULL ,
    MNO CHAR ,                   |      MNO VARCHAR(6) NOT NULL ,
    ... }                        |       ..... ,
  KEY { ENO }                    |      UNIQUE ( ENO ) ,
  FOREIGN KEY { MNO }            |      FOREIGN KEY ( MNO )
     REFERENCES EMP { ENO }      |      REFERENCES EMP ( ENO ) ) ;
        RENAME ( ENO AS MNO ) ;  |
```

Как видите, имеется существенное различие между этими двумя спецификациями FOREIGN KEY. Сначала я остановлюсь на варианте, написанном на языке **Tutorial D**[1]. Атрибут MNO – это номер служащего, приписанный начальнику служащего, обозначенного атрибутом ENO (например, в кортеже EMP для служащего E3 значение MNO могло бы

[1] Точности ради я должен пояснить, что на момент написания этой книги в языке **Tutorial D** не было явной поддержки внешних ключей. Однако предложения добавить такую поддержку активно рассматриваются (см. статью «*Inclusion Dependencies and Foreign Keys*», упомянутую в приложении D), и мне удобно считать, что они уже приняты.

быть равно номеру служащего E2, и в результате мы имели бы ссылку на кортеж EMP, представляющий служащего E2). Поэтому «ссылающаяся переменная-отношение» (в определении она названа *R2*) и «переменная-отношение, на которую указывает ссылка» (в определении она названа *R1*) в этом примере совпадают. Но ближе к теме: значения внешних ключей, как и значения потенциальных ключей, суть *кортежи*; поэтому мы должны произвести в спецификации внешнего ключа переименование, чтобы сравнение кортежей на равенство было хотя бы синтаксически корректно. (Какое сравнение кортежей на равенство? *Ответ*: То, что неявно выполняется в процессе проверки ограничения внешнего ключа; напомним, что сравнивать на равенство можно только кортежи одинакового типа, а «одинаковость типа» означает, что атрибуты должны быть одинаковы и, в частности, иметь одинаковые имена.) Вот почему в спецификации на языке **Tutorial D** целью является не просто EMP, а EMP{ENO} RENAME (ENO AS MNO). *Примечание*: Оператор RENAME подробно описан в следующей главе, а пока я надеюсь, что его назначение вам и так понятно.

Теперь обратимся к SQL. Памятуя о приведенном выше определении внешнего ключа, отметим, что в SQL ключ *K* и соответствующий ему внешний ключ *FK* представляют собой последовательности, а не множества столбцов. (Иными словами, наличие упорядочения слева направо важно.) Пусть эти столбцы в той последовательности, которая задана в спецификации FOREIGN KEY, будут *B1*, *B2*, ..., *Bn* (для *FK*) и *A1*, *A2*, ..., *An* (для K).[1] Тогда столбцы *Bi* и *Ai* ($1 \leq i \leq n$) должны иметь одинаковые типы, но могут иметь разные имена. Вот почему в SQL спецификация

```
FOREIGN KEY ( MNO ) REFERENCES EMP ( ENO )
```

корректна даже без переименования.

Рекомендация: Несмотря на это последнее замечание, старайтесь по возможности называть столбцы внешнего ключа точно так же, как столбцы соответствующего ключа (вопрос об именовании столбцов обсуждался в главе 3). Однако бывают ситуации – ровно две, если быть точным, – когда этой рекомендации невозможно следовать на все 100 процентов:

- Когда в некоторой таблице *T* имеется внешний ключ, ссылающийся на ключ самой таблицы *T* (как, например, в таблице EMP).
- Когда в некоторой таблице *T2* есть два разных внешних ключа, ссылающихся на один и тот же ключ *K* таблицы *T1*.

---

[1] Столбцы *A1*, *A2*, ..., *An* должны встречаться в какой-то спецификации UNIQUE или PRIMARY KEY для целевой таблицы, но не обязательно в том же порядке, что в спецификации FOREIGN KEY. Однако и сами эти столбцы, и окружающие их скобки в спецификации FOREIGN KEY можно опустить, но в этом случае они должны встречаться в спецификации PRIMARY KEY, а не UNIQUE для целевой таблицы, и, разумеется, должны быть заданы в ней в надлежащем порядке.

Но даже при таких условиях нужно стремиться соблюсти хотя бы дух приведенной выше рекомендации. Например, во втором случае можно сделать так, чтобы один из внешних ключей включал столбцы с теми же именами, что *K*, пусть даже для другого это не так (и не может быть так). См. упражнение 5.15 в конце главы.

Ссылочные действия

Как вы, наверное, знаете, SQL поддерживает не только внешние ключи, но также *ссылочные действия*, например CASCADE. Такие действия можно задавать во фразах ON DELETE и ON UPDATE. Например, предложение CREATE TABLE для таблицы поставок могло бы включать такую спецификацию:

```
FOREIGN KEY ( SNO ) REFERENCES S ( SNO ) ON DELETE CASCADE
```

В этом случае попытка удалить некоторого поставщика привела бы к каскадному удалению всех связанных с ним поставок.

Отметим, что ссылочные действия могут оказаться полезны на практике, однако частью реляционной модели они не являются. Но это не обязательно считать проблемой! Реляционная модель – это фундамент всей отрасли баз данных, но *и только*. Поэтому не видно, почему на этом фундаменте не разместить дополнительные средства, – при условии, конечно, что они не нарушают предписаний модели (наверное, стоит еще добавить, что они также должны отвечать духу модели и быть действительно полезны). Немного разовью эту мысль.

- Самый очевидный пример дает теория типов. В главе 2 мы видели, что «типы ортогональны таблицам», но видели также и то, что полная и строгая поддержка типов в реляционных системах – быть может, даже с поддержкой наследования – весьма желательна, и это еще мягко сказано.
- *Триггерные процедуры*. Строго говоря, триггерная процедура – это действие (*триггерное действие*), которое выполняется при возникновении заданного события (*триггерного события*), но этот термин часто неформально употребляется и для обозначения самого триггерного события. Ссылочные триггерные действия, например ON DELETE CASCADE, – это просто прагматически важный пример более общей конструкции, в которой действием является DELETE (фактически, «процедура», определенная в данной случае декларативно), а триггерным событием – ON DELETE.[1] Реляционная модель не предписывает никаких триггерных процедур, но и не запрещает их, – а запрещала бы, если бы они нарушали теоретико-множественную природу модели или *принцип присваивания*, что, кста-

[1] Если вам интересно, в терминологии SQL ON DELETE CASCADE называется «ссылочным триггерным действием» (referential triggered action), а само слово CASCADE – «ссылочным действием» (referential action).

ти, на практике весьма вероятно. *Примечание*: Сочетание триггерного события и соответствующего триггерного действия часто называют *триггером*. **Рекомендация**: Как отмечалось выше, избегайте имеющихся в SQL триггеров на уровне строк и вообще не пользуйтесь триггерами, если это нарушает *принцип присваивания*.

- В качестве третьего примера скажу, что реляционная модель почти ничего не говорит о восстановлении и управлении параллельным доступом, но это не означает, что реляционные системы могут не предоставлять такие средства. (На самом деле, можно возразить, что реляционная модель все-таки упоминает о таких материях неявно, так как полагает, что СУБД реализует обновления корректно и без потери данных; тем не менее ничего конкретного она не предписывает.)

И еще одно последнее замечание. Я завел разговор о внешних ключах, потому что их прагматическая ценность несомненна, а также потому, что они определены в оригинальной модели. Но хочу подчеркнуть, что ничего фундаментального в них нет, это просто сокращенная запись для некоторого ограничения целостности, которое часто встречается на практике, как мы увидим в главе 8. (На самом деле почти то же самое можно сказать и о потенциальных ключах, однако в этом случае практическая польза от наличия такой сокращенной записи неизмеримо больше.)

Переменные-отношения и предикаты

Вот мы и подошли к теме, которую в каком-то смысле можно назвать важнейшей в этой главе. Суть ее в том, что существует другой взгляд на переменные-отношения. Я имею в виду, что многие считают переменные-отношения просто файлами в традиционном для информатики смысле – быть может, довольно абстрактные файлы, но все-таки файлы. Но можно взглянуть на них и иначе, и, как мне кажется, это позволит гораздо глубже понять, что происходит на самом деле.

Рассмотрим переменную-отношение S, представляющую поставщиков. Предполагается, что она, как и все переменные-отношения, описывает какую-то часть реального мира. Но я мог бы выразиться и точнее: заголовок этой переменной-отношения представляет некоторый *предикат*, то есть обобщенное утверждение о какой-то части реального мира (обобщенное, потому что оно *параметризовано*, как я вскоре объясню). Предикат выглядит следующим образом:

Поставщик SNO связан контрактом, он называется SNAME, имеет статус STATUS и находится в городе CITY.

Этот предикат являет собой *подразумеваемую интерпретацию*, или смысловое содержание переменной-отношения S, называемое также *интенцией*.

Вообще говоря, предикат можно считать *функцией, возвращающей значение истинности*. Как и у любой другой функции, у него есть набор

параметров, а при вызове он возвращает результат, который может принимать одно из двух значений: TRUE или FALSE. Например, параметрами вышеупомянутого предиката являются SNO, SNAME, STATUS и CITY (они соответствуют атрибутам переменной-отношения), и обозначают они значения соответствующих типов (CHAR, CHAR, INTEGER и CHAR). При вызове этой функции – в логике это называется *порождением предиката* – мы подставляем вместо параметров аргументы. Предположим, что мы подставили соответственно аргументы S1, Smith, 20 и London. Тогда получается следующее предложение:

Поставщик S1 связан контрактом, он называется Smith, имеет статус 20 и находится в городе London.

Фактически это предложение представляет собой *высказывание*, так в формальной логике называют утверждение, которое может быть или истинным, или ложным и не содержит никаких условий. Вот два примера высказываний:

1. «Банду разводного ключа» (The Monkey Wrench Gang) написал Эдвард Эбби.
2. «Банду разводного ключа» написал Вильям Шекспир.

Первое истинно, второе ложно. Не делайте распространенную ошибку, считая, что всякое высказывание должно быть истинным! Однако те, которые я сформулирую ниже, действительно *предполагаются* истинными.

- Прежде всего, с каждой переменной-отношением ассоциирован предикат, который называется *предикатом переменной-отношения*. (Показанный выше предикат является предикатом переменной-отношения S.)
- Пусть с переменной-отношением R ассоциирован предикат P. Тогда каждый кортеж t, входящий в R в данный момент времени, можно рассматривать как представление некоторого высказывания p, которое получается путем вызова (или *порождения*) P в этот момент со значениями атрибутов t в качестве аргументов.
- И (*очень важно!*) мы принимаем соглашение о том, что каждое полученное таким образом высказывание p истинно.

Так, взяв в качестве примера нашу переменную-отношение S, мы предполагаем, что каждое из следующих высказываний в данный момент времени истинно:

Поставщик S1 связан контрактом, он называется Smith, имеет статус 20 и находится в городе London.

Поставщик S2 связан контрактом, он называется Jones, имеет статус 10 и находится в городе Paris.

Поставщик S3 связан контрактом, он называется Blake, имеет статус 30 и находится в городе Paris.

И так далее. Но мы можем пойти еще дальше: если некоторый кортеж теоретически мог бы встретиться в некоторой переменной-отношении в какой-то момент времени, но фактически не встречается, то мы предполагаем, что соответствующее высказывание в данный момент ложно. Например, кортеж

```
TUPLE { SNO 'S6' , SNAME 'Lopez' , STATUS 30 , CITY 'Madrid' }
```

вполне годится на роль поставщика, но в данный момент отсутствует в переменной-отношении S, поэтому мы вправе *предположить*, что следующее высказывание в данный момент времени ложно:

> *Поставщик S6 связан контрактом, он называется Lopez, имеет статус 30 и находится в городе Madrid.*

Подведем итог. В любой момент времени данная переменная-отношение *R* содержит *те и только те* кортежи, которым соответствуют истинные высказывания (порождения предиката переменной-отношения *R* возвращают TRUE) в этот момент; по крайней мере, именно это мы всегда подразумеваем на практике. Иначе говоря, на практике мы принимаем так называемое *допущение замкнутости мира* (развитие этой темы см. в приложении А).

Еще немного терминологии. Пусть снова *P* – предикат переменной-отношения, или интенция переменной-отношения *R*, и пусть отношение *r* – это значение *R* в некоторый момент времени. Тогда *r* – тело *r*, если быть точным, – составляет *экстенцию P* в тот же момент времени. Отметим, однако, что экстенция для данной переменной-отношения меняется со временем, а интенция – нет.

И последние два терминологических замечания:

- Возможно, вам уже знаком термин *предикат*, поскольку в SQL он часто употребляется для обозначения того, что в этой книге называется булевым выражением (то есть в SQL встречаются словосочетания «предикат IN», «предикат EXISTS» и т. д.). Нельзя сказать, что такое словоупотребление совсем уж некорректно, но SQL узурпировал очень общий термин – весьма важный в реляционных контекстах – и придал ему довольно узкое значение, поэтому лично я предпочитаю этой практике не следовать.
- Если уж зашла речь об узурпировании общих терминов и наделении их специализированной семантикой, то есть еще один потенциальный источник путаницы. Он связан с термином *statement*. В логике этот термин употребляется в смысле, очень близком к естественному языку[1]. Напротив, в языках программирования ему придается более ограниченный и специализированный смысл. Так называется конструкция, приводящая к выполнению некоторого действия, на-

[1] В контексте логики слово *statement* переводится как *утверждение*, *суждение*, в языках программирования – как *предложение*, хотя встречаются также переводы *команда*, *инструкция*. – *Прим. перев.*

пример к определению либо модификации переменной или к изменению потока управления. Боюсь, что в настоящей книге этот термин используется в обоих смыслах, и надеюсь, что конкретное значение ясно из контекста. *Caveat lector.*

Отношения и типы

В главе 2 среди прочего мы обсуждали типы и отношения. Однако тогда я еще не мог объяснить самое важное логическое различие между этими понятиями. Теперь могу – и объясню.

Я показал, что в любой момент времени базу данных можно рассматривать как совокупность истинных высказываний, таких, например, как высказывание *Поставщик S1 связан контрактом, он называется Smith, имеет статус 20 и находится в городе London*. Точнее, я показал, что значения аргументов подобного высказывания (в данном случае S1, Smith, 20 и London) – это в точности значения атрибутов, взятых из соответствующего кортежа, причем каждый такой атрибут имеет значение ассоциированного с ним типа. Отсюда следует, что:

Типы – это множества предметов, о которых мы можем рассуждать; отношения – это (истинные) суждения о таких предметах.

Другими словами, типы дают нам словарь (предметы, о которых можно рассуждать), а отношения – возможность высказывать суждения об этих предметах. (Возможно, вам поможет такая аналогия: *между типами и отношениями такая же связь, как между существительными и грамматическими предложениями*.) Например, если для простоты ограничиться только поставщиками, то мы увидим, что:

- Предметы, о которых мы можем рассуждать, – это символьные строки и целые числа, и ничего более. (В реальной базе данных наш словарь обычно гораздо шире, особенно если участвуют еще и типы, определенные пользователем).
- Возможны суждения вида «поставщик с номером поставщика, обозначенным некоторой символьной строкой, связан контрактом, имеет имя, обозначенное другой символьной строкой, имеет статус, обозначенный целым числом, и находится в городе, обозначенном третьей символьной строкой» – и никакие другие. (Никакие, кроме *логически вытекающих* из тех суждений, которые мы можем сформулировать явно. Например, уже зная, что можно явно сказать о поставщике S1, мы можем также сказать, что *Поставщик S1 связан контрактом, называется Smith, имеет статус 20 и находится в некотором городе*, причем город оставлен не заданным. И если вы полагаете, что сказанное мной очень напоминает и, пожалуй, даже имеет глубинную связь с реляционной проекцией... что ж, вы абсолютно правы. См. раздел «В чем смысл реляционных выражений?» в главе 6.)

Из всего сказанного выше вытекают по меньшей мере три важных следствия. Конкретно, чтобы «описывать какую-то часть реального мира» (так я выразился в предыдущем разделе):

1. Необходимы и типы, и отношения – без типов нам не о чем было бы рассуждать, а без отношений мы ничего не смогли бы сказать.
2. Типы и отношения достаточны и необходимы – логически говоря, нам больше ничего не нужно. (Конечно, нам необходимы переменные-отношения, чтобы отразить изменчивость реального мира во времени, но для представления ситуации в любой конкретный момент времени они ни к чему.)

Отступление

Когда я говорю, что типы и отношения необходимы и достаточны, то, разумеется, имею в виду только логический уровень. Понятно, что на физическом уровне нужны и другие конструкции (к примеру, указатели), но это просто потому, что на этом уровне ставятся другие цели. Физический уровень намеренно выведен за рамки реляционной модели.

3. Типы и отношения – не одно и то же. Берегитесь тех, кто пытается уверить вас, будто это не так! И ведь есть такие продукты, которые как раз и «делают вид», что тип – это просто частный случай отношения (хотя, конечно, выражают эту мысль другими словами), но я надеюсь, что любой продукт, основанный на подобной логической ошибке, обречен на провал. (Фактически, по крайней мере один из продуктов, которые я имею в виду, уже прогорел.) Впрочем, продукты, о которых я говорю, не являются реляционными; обычно это системы, поддерживающие «объекты» в объектно-ориентированном смысле или пытающиеся тем или иным способом «поженить» объекты и SQL-таблицы. Детальное их рассмотрение выходит за рамки этой книги.

Изложу несколько более формальный взгляд на все вышесказанное. Как мы видели, базу данных можно представлять себе как совокупность истинных высказываний. В действительности база данных в сочетании с операторами, которые применяются к представленным в этой базе высказываниям (точнее, к множествам таких высказываний), – это *логическая система*. Под «логической системой» я понимаю здесь формальную систему – такую, как, скажем, евклидова геометрия, – в которой имеются *аксиомы* («исходные истины») и *правила вывода*, позволяющие доказывать *теоремы* («производные истины») на основе аксиом. Величайшим озарением Кодда, когда в 1969 году он придумал реляционную модель, было осознание того факта, что база данных (вопреки названию) – это не просто совокупность данных, а совокупность *фактов*, или истинных высказываний. Эти высказывания – заданные, то есть представленные кортежами в базовых переменных-отношениях, – являются аксиомами логической системы. А правила вывода – это те

правила, с помощью которых можно из имеющихся высказываний выводить новые; иными словами, правила говорят, как применять операторы реляционной алгебры. Таким образом, вычисляя некоторое реляционное выражение (в частности, отвечая на запрос), система на самом деле выводит новые истины из заданных, то есть по существу доказывает теорему!

Осознав это, мы приходим к заключению, что для поиска подходов к «проблеме баз данных» в нашем распоряжении оказывается весь аппарат формальной логики. Другими словами, следующие вопросы:

- Как база данных должна выглядеть с точки зрения пользователя?
- Как должны выглядеть ограничения целостности?
- Как должен выглядеть язык запросов?
- Как лучше всего реализовать запросы?
- Более общо, как лучше всего вычислять выражения?
- Как следует представлять результаты пользователю?
- Как вообще следует проектировать базу данных?

(и им подобные) становятся, по сути дела, вопросами, к которым можно применить методы формальной логики и получить логически безупречные ответы.

Не стоит и говорить, что реляционная модель непосредственно поддерживает только что изложенную интерпретацию, и именно поэтому, на мой взгляд, она несокрушима, как скала, и ей суждена долгая жизнь. И именно поэтому, опять-таки на мой взгляд, прочие «модели данных» просто находятся в другой весовой категории. Я совершенно серьезно задаю вопрос, можно ли вообще эти «модели данных» называть моделями в том же смысле, в каком таковой является реляционная модель. Ведь большинство из них в той или иной степени ситуативны, а не возведены, как реляционная модель, на незыблемом фундаменте теории множеств и логики предикатов. Я еще вернусь к этим вопросам в приложении А.

Упражнения

Упражнение 5.1. Иногда предлагают рассматривать переменную-отношение как обычный файл, в котором кортежи играют роль записей, а атрибуты – роль полей. Обсудите такой подход.

Упражнение 5.2. Объясните своими словами, почему фразы типа «Эта операция UPDATE обновляет статус всех поставщиков в Лондоне» не очень точны. Дайте настолько точную переформулировку этой фразы, насколько сможете.

Упражнение 5.3. Почему операции «позиционного обновления» в SQL – неудачная идея?

Упражнение 5.4. Пусть SS – базовая таблица с такими же столбцами, как S. Рассмотрим следующие SQL-предложения INSERT:

```
INSERT INTO SS ( SNO , SNAME , STATUS , CITY )
     ( SELECT SNO , SNAME , STATUS , CITY
       FROM   S
       WHERE  SNO = 'S6' ) ;

INSERT INTO SS ( SNO , SNAME , STATUS , CITY ) VALUES
     ( SELECT SNO , SNAME , STATUS , CITY
       FROM   S
       WHERE  SNO = 'S6' ) ;
```

Являются ли они логически эквивалентными? Если нет, то в чем их различие?

Упражнение 5.5. (*Это, по существу, повтор упражнения 2.21 из главы 2, но теперь вы можете дать более полный ответ.*) Сформулируйте *принцип присваивания*. Можете ли вы назвать ситуации, в которых SQL нарушает этот принцип? Можете ли вы сказать, каковы негативные последствия такого нарушения?

Упражнение 5.6. Напишите определения базовых таблиц SQL, соответствующих переменным-отношениям TAX_BRACKET, ROSTER и MARRIAGE из раздела «Еще о потенциальных ключах».

Упражнение 5.7. Почему бессмысленно говорить о том, что отношение имеет ключ?

Упражнение 5.8. В тексте главы я назвал одну причину, по которой неприводимость ключа – здравая идея. Видите ли вы еще какие-нибудь?

Упражнение 5.9. «Значения ключа – не скаляры, а кортежи». Прокомментируйте это замечание.

Упражнение 5.10. Пусть степень переменной-отношения R равна n. Какое максимальное количество ключей может иметь R?

Упражнение 5.11. В чем различие между ключом и суперключом? И коль скоро существует понятие суперключа, то имеет ли, по вашему мнению, смысл такая вещь, как подключ?

Упражнение 5.12. Переменная-отношение EMP из раздела «Еще о внешних ключах» – это пример так называемой *самоссылающейся* переменной-отношения. Придумайте какие-нибудь тестовые данные для такой переменной-отношения. Верно ли, что такие примеры с неизбежностью влекут за собой требование о поддержке null-значений? (*Ответ*: нет, но они показывают, насколько соблазнительной может быть эта мысль.) Что можно сделать в этом примере, если null-значения запрещены?

Упражнение 5.13. Почему в SQL нет никакого аналога опции переименования в спецификации внешних ключей в языке **Tutorial D**?

Упражнение 5.14. Можете ли вы придумать ситуацию, в которой каждая из двух переменных-отношений *R1* и *R2* имеет внешний ключ, ссылающийся на другую? Если да, то каковы последствия?

Упражнение 5.15. В хорошо известном приложении *разузлования* речь идет о переменной-отношении (назовем ее PP), показывающей для каждого компонента (узла), из каких компонентов (узлов или простых деталей) он состоит, с указанием количества каждой детали (например, «деталь P1 состоит из 4 деталей P2»). Дайте определения PP на языках **Tutorial D** и SQL. Как вы думаете, какие ссылочные действия могут иметь смысл в этом примере?

Упражнение 5.16. Изучите имеющуюся в вашем распоряжении SQL-систему. Какие ссылочные действия она поддерживает? Какие, на ваш взгляд, полезны? Можете ли вы назвать другие действия, которые не поддерживаются системой, но могли бы оказаться полезными?

Упражнение 5.17. Определите термины *высказывание* и *предикат*. Приведите примеры.

Упражнение 5.18. Назовите предикаты для переменных-отношений P и SP из базы данных о поставщиках и деталях.

Упражнение 5.19. Как вы понимаете термины *интенция* и *экстенция*?

Упражнение 5.20. Пусть *DB* – какая-нибудь знакомая вам база данных, и пусть *R* – произвольная переменная-отношение в *DB*. Что есть предикат для *R*? *Примечание*: Смысл этого упражнения в том, чтобы вы применили некоторые из обсуждавшихся в этой главе идей к своим данным и постарались начать думать о данных в таких терминах. Очевидно, у этого упражнения нет единственного правильного ответа.

Упражнение 5.21. Объясните своими словами *допущение замкнутости мира*. Возможно ли *допущение открытости мира*?

Упражнение 5.22. Ключ – это множество атрибутов, а пустое множество вполне допустимо; таким образом, мы можем определить *пустой* ключ как ключ с пустым множеством атрибутов. Какие следствия отсюда вытекают? Можете ли вы придумать применения такому ключу?

Упражнение 5.23. У предиката имеется множество параметров, а пустое множество вполне допустимо; таким образом, возможен предикат с пустым множеством параметров. Какие следствия отсюда вытекают?

Упражнение 5.24. Что такое предикат для переменной-отношения степени 0? (Имеет ли этот вопрос смысл? Обоснуйте свой ответ.)

Упражнение 5.25. Значением любой переменной-отношения является отношение. Верно ли обратное? Иными словами, всякое ли отношение является значением некоторой переменной-отношения?

6

SQL и реляционная алгебра I: оригинальные операторы

Это первая из двух глав, посвященных операторам реляционной алгебры; в ней детально обсуждаются оригинальные операторы (то есть те, что упоминались в главе 1) и попутно рассматриваются некоторые важные вопросы, например о важности правильного именования атрибутов (или столбцов). Также объясняется, как все это отражается на нашей главной цели – реляционном использовании SQL.

Предварительные сведения

Я хочу начать с повторения изложенного в главе 1. Во-первых, напомню, что любой алгебраический оператор принимает на входе по меньшей мере одно отношение и на выходе порождает другое отношение. Во-вторых, напомню и о том, что как исходные данные, так и результат являются отношениями, и именно это свойство, называемое *замкнутостью* алгебры, позволяет записывать вложенные реляционные выражения. В-третьих, в главе 1 я дал краткое описание операторов, которые называю «оригинальными» (ограничение, проекция, произведение, пересечение, объединение, разность и соединение), теперь же я могу определить эти и другие операторы гораздо более строго. Но сначала необходимо сделать несколько замечаний общего характера:

1. Операторы алгебры являются *обобщенными*: они применимы к *любым отношениям*. Например, не нужен один оператор соединения для того, чтобы соединить служащих с отделами, и другой – для соединения поставщиков с поставками. (Кстати, как вы думаете, применимо ли аналогичное замечание к объектным системам?)
2. Операторы предназначены *только для чтения*: они «читают» свои операнды и возвращают значение, но ничего не обновляют. Иными

словами, они применяются к отношениям, а не к переменным-отношениям.

3. Обратите внимание, что предыдущее замечание не означает, что реляционные выражения не могут ссылаться на переменные-отношения. Например, если *R1* и *R2* имена переменных-отношений, то *R1 UNION R2*, конечно, является допустимым реляционным выражением в языке **Tutorial D** (при условии, что эти переменные-отношения имеют один и тот же тип). Однако в этом выражении *R1* и *R2* обозначают не переменные-отношения как таковые, а отношения, являющиеся текущими значениями этих переменных-отношений в данный момент времени. Иначе говоря, мы, безусловно, можем указывать имя переменной-отношения для обозначения *отношения-операнда*, а такая ссылка на переменную-отношение сама составляет допустимое реляционное выражение[1], но в принципе с равным успехом мы могли бы на месте этого операнда написать подходящее отношение-литерал.

   Следующая аналогия поможет прояснить последнюю мысль. Предположим, что N – переменная типа INTEGER, имеющая в момент времени *t* значение 3. Тогда N + 2 – допустимое выражение, которое в момент времени *t* означает в точности 3 + 2, не более и не менее.

4. Наконец, коль скоро операторы алгебры способны только читать, значит, INSERT, DELETE и UPDATE (а также реляционное присваивание), несомненно, являющиеся реляционными операторами, не являются частью алгебры как таковой, хотя, к сожалению, в литературе часто можно встретить противоположное утверждение.

Необходимо также сказать несколько слов о структуре языка **Tutorial D**, поскольку поддержка алгебры в нем устроена совершенно иначе, чем в SQL. Дело в том, что в операциях типа UNION или JOIN, где необходимо устанавливать соответствие между атрибутами операндов, **Tutorial D** решает задачу, требуя, чтобы рассматриваемые атрибуты формально представляли собой один и тот же атрибут (то есть имели одинаковые имена и типы). Например, вот как выглядит в **Tutorial D** выражение для соединения деталей и поставщиков по городу:

```
P JOIN S
```

Здесь операция соединения, по определению, выполняется по городу, так как CITY – единственный атрибут, общий для P и S. А вот как та же самая операция записывается в SQL (обратите особое внимание на последнюю строчку, в которое соответствие между атрибутами – точнее, столбцами – задано явно):

```
SELECT P.PNO , P.PNAME , P.COLOR , P.WEIGHT , P.CITY /* или S.CITY */ ,
       S.SNO , S.SNAME , S.STATUS
```

[1] Но в SQL это необязательно так! Например, если *T1* и *T2* – имена SQL-таблиц, то, как правило, мы не можем написать нечто вроде *T1 UNION T2*, а должны вместо этого писать SELECT * FROM T1 UNION SELECT * FROM *T2*.

```
FROM   P , S
WHERE  P.CITY = S.CITY
```

Вообще-то, этот пример можно записать на SQL многими способами. Ниже приводятся еще три. Как видите, второй и третий по духу несколько ближе к **Tutorial D** (обратите внимание, в частности, на то, что результат соединения в этих двух вариантах содержит столбец с именем просто CITY, а не P.CITY или S.CITY):

```
SELECT P.PNO , P.PNAME , P.COLOR , P.WEIGHT , P.CITY /* или S.CITY */ ,
       S.SNO , S.SNAME , S.STATUS
FROM   P JOIN S
ON     P.CITY = S.CITY

SELECT P.PNO , P.PNAME , P.COLOR , P.WEIGHT , CITY ,
       S.SNO , S.SNAME , S.STATUS
FROM   P JOIN S
USING  ( CITY )

SELECT P.PNO , P.PNAME , P.COLOR , P.WEIGHT , CITY ,
       S.SNO , S.SNAME , S.STATUS
FROM   P NATURAL JOIN S
```

Однако я выбрал ту формулировку, что выбрал, отчасти потому, что только она поддерживалась в оригинальном варианте SQL, а отчасти, и это более важно, потому что она позволяет мне отметить ряд дополнительных различий между SQL и алгеброй в том виде, в каком она реализована в **Tutorial D**.

- SQL допускает, а иногда даже требует, квалифицированных имен с точками, **Tutorial D** – нет. *Примечание*: О квалифицированных именах в SQL я еще буду говорить в главе 12.
- **Tutorial D** иногда требует переименовывать атрибуты, чтобы избежать конфликта имен или несоответствия. SQL обычно в этом не нуждается (хотя и поддерживает некий аналог оператора RENAME, который используется в **Tutorial D** для этой цели, как мы увидим в следующем разделе).
- Отчасти как следствие предыдущего пункта, **Tutorial D** не нуждается в понятии «корреляционного имени», существующем в SQL; по существу, оно заменено идеей о том, что атрибуты иногда необходимо переименовывать, о чем шла речь выше. *Примечание*: Корреляционные имена в SQL будут подробно рассматриваться в главе 12.
- Хотя в примере выше этого не видно, SQL иногда устанавливает соответствие столбцов, исходя из того, в каком порядке (слева направо) они упоминаются. **Tutorial D** так никогда не делает.
- Помимо явной или неявной поддержки некоторых средств реляционной алгебры, SQL также явно поддерживает некоторые средства реляционного исчисления (один пример – корреляционные имена,

другой – оператор EXISTS). **Tutorial D** реляционное исчисление не поддерживает. Одним из результатов такого различия является тот факт, что SQL – довольно избыточный язык, то есть часто предлагает много разных способов сформулировать один и тот же запрос, и это может иметь серьезные негативные последствия для оптимизатора. (Когда-то я написал на эту тему статью под названием «Пятьдесят способов задать вопрос» – см. приложение D, – в которой показал, что для выражения даже такого простого запроса, как «получить имена поставщиков, которые поставляют деталь P2», в SQL есть больше 50 способов.)

- В SQL большинство запросов должно строиться по следующему шаблону: SELECT - FROM - WHERE. В **Tutorial D** аналогичного требования нет. *Примечание*: В следующей главе у меня еще будет что сказать по этому поводу.

Далее я буду приводить примеры как на **Tutorial D**, так и на SQL.

Еще о замкнутости

Повторю, что результатом любой реляционной операции является отношение. И наоборот, любой оператор, результатом которого не является отношение, по определению не реляционный. Например, оператор, порождающий упорядоченный результат, реляционным не является (см. обсуждение ORDER BY в следующей главе). А конкретно в SQL то же самое справедливо для любого оператора, который порождает результат, содержащий строки-дубликаты, или столбцы, упорядоченные слева направо, или null-значения, или анонимные столбцы, или столбцы с повторяющимися именами. Замкнутость – важнейшее свойство! Как я уже говорил, именно замкнутость позволяет записывать вложенные выражения в реляционной модели, и (как мы увидим ниже) она важна также для трансформации выражений и, следовательно, для оптимизации. **Настоятельная рекомендация**: Не пользуйтесь операциями, которые нарушают свойство замкнутости, если хотите, чтобы результат можно было подвергнуть дальнейшей реляционной обработке.

Отступление

Несмотря на замечания в предыдущем абзаце, некоторые авторы не без оснований считают *реляционное включение* ($\subseteq$) реляционной операцией – точнее, частью реляционной алгебры, – хотя ее результатом является значение истинности, а не отношение. Однако этот вопрос не настолько важен, чтобы ломать здесь копья.

Надеюсь, всем ясно, что мои слова о том, что результат любой алгебраической операции – отношение, следует понимать концептуально; я не хочу сказать, что система должна материализовать результат операции целиком. Рассмотрим, к примеру, следующее выражение (ограничение со-

единения; слева, как обычно, версия на **Tutorial D**, справа – на SQL, причем в SQL-версии я сознательно показал квалифицированные имена):

```
( P JOIN S )            |    SELECT P.* , S.SNO , S.SNAME , S.STATUS
WHERE PNAME > SNAME     |    FROM   P , S
                        |    WHERE  P.CITY = S.CITY
                        |    AND    P.PNAME > S.SNAME
```

Ясно, что как только очередной кортеж соединения сформирован, система в состоянии сразу же проверить, выполняется ли для него ограничение PNAME > SNAME (P.PNAME > S.SNAME в SQL-версии), то есть должен ли он входить в окончательный результат или его следует отбросить. Следовательно, промежуточный результат операции соединения можно было бы вообще не материализовывать в виде настоящего отношения. (На практике все системы стремятся во что бы то ни стало избежать материализации промежуточных результатов, по очевидным соображениям производительности.)

В данном примере затрагивается еще один важный момент. Рассмотрим булево выражение PNAME > SNAME в версии на **Tutorial D**. Концептуально это выражение применяется к результату операции P JOIN S, следовательно, имена атрибутов PNAME и SNAME в нем относятся к атрибутам этого результата, а *не* к одноименным атрибутам переменных-отношений P и S. Но откуда мы знаем, что результат соединения содержит такие атрибуты? Каков заголовок этого результата? И вообще, откуда нам известно, какой заголовок имеет результат *произвольной* алгебраической операции? Очевидно, необходим какой-то набор правил – конкретно, *правил вывода типа отношения*, – который позволял бы по известным заголовкам (и, стало быть, типам) исходных для операции отношений выводить заголовок (а, значит, и тип) результата операции. И в реляционной модели такой набор правил есть. Для случая соединения эти правила говорят, что результатом операции P JOIN S является отношение такого типа:

```
RELATION { PNO CHAR , PNAME CHAR , COLOR CHAR , WEIGHT FIXED ,
           CITY CHAR , SNO CHAR , SNAME CHAR , STATUS INTEGER }
```

Фактически, для соединения заголовок результата представляет собой объединение заголовков исходных отношений, где под *объединением* я понимаю обычное теоретико-множественное объединение, а не специальное реляционное объединение, которое будет обсуждаться ниже в этой главе. Иначе говоря, результат обладает всеми атрибутами исходных отношений, но общие атрибуты – в нашем примере только CITY – встречаются один раз, а не дважды. Разумеется, эти атрибуты не упорядочены слева направо, поэтому я с тем же успехом мог бы сказать, что результат операции P JOIN S имеет такой тип:

```
RELATION { SNO CHAR , PNO CHAR , SNAME CHAR , PNAME CHAR ,
           CITY CHAR , STATUS INTEGER , WEIGHT FIXED , COLOR CHAR }
```

Отмечу, что какие-то правила вывода типа, безусловно, необходимы для поддержки свойства замкнутости в полном объеме. Это свойство означает, что результатом любой операции является отношение, а у отношений есть тело и заголовок; значит, любой результат должен иметь корректный реляционный заголовок и корректное реляционное тело.

Кстати, оператор RENAME, упоминавшийся во введении к этой главе, в значительной мере необходим именно из-за правил вывода типа; он позволяет выполнять, к примеру, соединение даже тогда, когда отношения-операнды не удовлетворяют требованиям к именованию атрибутов для данной операции (говоря не слишком строго). Приведу определение.

> **Определение**: Пусть r – отношение, и пусть A – атрибут r. Тогда результатом *переименования* r RENAME (A AS B) является отношение, для которого (а) заголовок совпадает с заголовком r, с тем отличием, что атрибут A переименован в B, и (б) тело совпадает с телом r (с тем отличием, что все ссылки на A в теле заменены ссылками на B, хотя сейчас мы можем не обращать внимания на эту тонкость).

Например:

```
S RENAME ( CITY AS SCITY )     |    SELECT SNO , SNAME , STATUS ,
                               |           S.CITY AS SCITY
                               |    FROM   S
```

Для наших обычных тестовых данных получается такой результат:

SNO	SNAME	STATUS	SCITY
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

Примечание

В этой главе я обычно не показываю результаты явно, разве что в случае, когда думаю, что рассматриваемый оператор может быть вам незнаком.

Важное замечание: В приведенном выше примере переменная-отношение S в базе данных *не* изменяется! Оператор RENAME отличается от SQL-предложения ALTER TABLE тем, что обращение к RENAME – это просто выражение (точно такое же, как, например, P JOIN S или N + 2), и, как любое выражение, оно лишь служит для обозначения некоторого значения. Более того, поскольку это *выражение*, а не предложение, или «команда», оно может быть вложено в другие выражения. Ниже мы встретим много примеров такого вложения.

Так как же SQL справляется с задачей вывода типа результата? Ответ: не очень хорошо. Во-первых, в главе 3 мы видели, что в SQL вообще нет понятия «тип отношения» (есть понятие «тип строки»). Во-вторых, SQL может порождать безымянные столбцы (как, например, в запросе SELECT DISTINCT 2 * WEIGHT FROM P). В-третьих, он может также порождать результаты с повторяющимися именами столбцов (как, например, в запросе SELECT DISTINCT P.CITY, S.CITY FROM P, S).

Настоятельная рекомендация: Придерживайтесь дисциплины именования столбцов, описанной в главе 3, если хотите, чтобы SQL максимально полно следовал сформулированным там же реляционным правилам. На всякий случай напомню, что эта дисциплина предполагает использование спецификации AS для присвоения правильных имен столбцам, которые в противном случае (а) не имели бы имени вовсе или (б) имели бы неуникальные имена. Игнорировать ее допускается разве что в случае, когда на столбцы без имени или с неуникальными именами нет никаких других ссылок. В примерах SQL в этой и следующей главах (да, собственно, во всей книге) я буду придерживаться этого правила.

Я еще не закончил разговор о примере в начале этого раздела. Давайте взглянем на него еще раз:

```
( P JOIN S )            |    SELECT P.* , S.SNO , S.SNAME , S.STATUS
WHERE PNAME > SNAME     |    FROM   P , S
                        |    WHERE  P.CITY = S.CITY
                        |    AND    P.PNAME > S.SNAME
```

Как видите, аналогом выражения PNAME > SNAME **Tutorial D** в языке SQL служит P.PNAME > S.SNAME, что по некотором размышлении выглядит довольно странно, так как это выражение по идее должно применяться к результату фразы FROM (см. раздел «Вычисление SQL-выражений» ниже), а переменные-отношения P и S уж точно не являются частью этого результата! На самом деле, довольно трудно объяснить, как нечто вроде P.PNAME во фразах WHERE и SELECT (а, быть может, и в других местах выражения) может иметь хоть какой-то смысл в терминах результата фразы FROM. Стандарт SQL дает такое объяснение, но уловки, на которые ему пришлось для этого пойти, куда сложнее правил вывода типа в **Tutorial D**, – настолько, что я даже не буду пытаться объяснить их здесь, а просто констатирую, что при необходимости это можно объяснить. Эту вольность я оправдываю тем, что вы предположительно уже знакомы с SQL. Правда, так и тянет спросить, а задумывались ли вы когда-нибудь над этим вопросом, ... но я не стану.

Теперь я готов продолжить описание некоторых алгебраических операторов. Отмечу, что ни в этой, ни в следующей главе я не пытаюсь дать исчерпывающую информацию. Я не собираюсь рассматривать «всем известные операторы», и даже те, которые я буду рассматривать, не собираюсь описывать во всей полноте и общности. Фактически в большин-

стве случаев я буду лишь давать приемлемое, но не вполне формальное, определение и приводить простые примеры.

Ограничение

> **Определение**: Пусть *r* – отношение, и пусть *bx* – булево выражение, в котором все атрибуты, на которые есть ссылки, являются атрибутами *r*, а ссылок на переменные-отношения нет вообще. Тогда *bx* называется *условием ограничения*, а *ограничением r* по *bx*, *r* WHERE *bx*, называется такое отношение, что (а) заголовок совпадает с заголовком *r* и (б) тело состоит из всех кортежей *r*, для которых значение *bx* равно TRUE.

Например:

```
P WHERE WEIGHT < 17.5          |    SELECT *
                               |    FROM   P
                               |    WHERE  WEIGHT < 17.5
```

Пусть *r* – отношение. Тогда ограничение *r* WHERE TRUE (или, более общо, любое ограничение вида *r* WHERE *bx*, где *bx* – произвольное выражение, например 1 = 1, тождественно равное TRUE) возвращает просто *r*. Такое ограничение называется *тождественным*.

Примечание

В языке **Tutorial D** поддерживаются выражения вида *r* WHERE *bx*, но эти выражения не обязаны содержать только такие простые ограничения, как приведенное выше, потому что булево выражение *bx* может быть более общим, чем условие ограничения. Аналогичное замечание относится и к SQL. Примеры будут приведены в последующих главах.

Попутно отмечу, что операция ограничения иногда называется *выборкой* (select), но я предпочитаю не употреблять этот термин из-за возможной путаницы со словом SELECT в SQL. В языке SQL SELECT, а точнее фраза SELECT в выражении SELECT – это вообще не ограничение, а некая комбинация операторов UNGROUP, EXTEND, RENAME и «проекции» (слово «проекция» заключено в кавычки, потому что по умолчанию эта операция не устраняет дубликаты). *Примечание*: Операторы UNGROUP и EXTEND описываются в следующей главе.

Проекция

> **Определение**: Пусть *r* – отношение и пусть *A*, *B*, ..., *C* – атрибуты *r*. Тогда *проекцией r* на эти атрибуты *r{A,B, ... ,C}* называется отношение, для которого (а) заголовок равен

> {*A*,*B*,…,*C*} и (b) тело состоит из множества всех кортежей *x*, таких, что в *r* существует кортеж *t*, для которого значение *A* равно значению *A* в *x*, значение *B* равно значению *B* в *x*, …, и значение *C* равно значению *C* в *x*.

Например:

```
P { COLOR , CITY }            |    SELECT DISTINCT COLOR , CITY
                              |    FROM   P
```

Еще раз повторю, что результат – это отношение, поэтому, как говорят, «дубликаты устраняются», то есть в формулировке на языке SQL слово DISTINCT обязательно.[1] Заголовок результата состоит из атрибутов – или столбцов – COLOR и CITY (в SQL именно в таком порядке).

Пусть *r* – отношение. Тогда проекция *r*{*H*}, где {*H*} – множество всех атрибутов (иными словами, заголовок) *r*, возвращает просто *r*. Такая проекция называется *тождественной*.

Кстати, язык **Tutorial D** позволяет также определять проекцию в терминах удаляемых, а не оставляемых атрибутов. Так, следующие выражения в **Tutorial D** эквивалентны:

```
P { COLOR , CITY }   и   P { ALL BUT PNO , PNAME , WEIGHT }
```

Эта возможность иногда позволяет существенно сократить длину вводимого текста (представьте себе проекцию отношения степени 100 на 99 атрибутов).[2] Это замечание относится ко всем операторам **Tutorial D**, для которых оно имеет смысл.

В реальном синтаксисе удобно приписать оператору проекции высокий приоритет. Так, в **Tutorial D** выражение

```
S JOIN P { PNO }
```

означает

```
S JOIN ( P { PNO } )
```

а не

```
( S JOIN P ) { PNO }
```

Упражнение: Продемонстрируйте различие между этими двумя интерпретациями на примере наших тестовых данных.

---

[1] Не могу не отметить, что термин «устранение дубликатов», который употребляется практически повсеместно (и не только в контексте SQL), не совсем точен, – правильнее было бы говорить об устранении дублирования.

[2] В отличие от отношения, существование переменной-отношения такой большой степени маловероятно, поскольку она, скорее всего, нарушала бы принципы нормализации (см. Приложение В). Но и сказать, что такие переменные-отношения вообще не встречаются, тоже нельзя.

Соединение

Прежде чем переходить собственно к определению соединения, полезно будет ввести понятие «соединяемости». Отношения *r1* и *r2* называются *соединяемыми* тогда и только тогда, когда атрибуты с одинаковыми именами имеют одинаковые типы, или, эквивалентно, когда теоретико-множественное объединение их заголовков само является допустимым заголовком. В следующей главе мы увидим, что это понятие относится не только к соединению как таковому, но и к другим операциям. Так или иначе, вооружившись этим понятием, я могу теперь определить операцию соединения.

> **Определение.** Пусть *r1* и *r2* – соединяемые отношения. Тогда их *естественным соединением* (или для краткости просто *соединением*) *r1* JOIN *r2* называется отношение, для которого (а) заголовком является теоретико-множественное объединение заголовков *r1* и *r2* и (б) телом является множество всех кортежей *t*, таких, что *t* есть теоретико-множественное объединение некоторого кортежа из *r1* и некоторого кортежа из *r2*.

Следующий пример повторяет пример из раздела «Предварительные сведения» с тем отличием, что я опустил явные квалификаторы имен в SQL-версии там, где их наличие необязательно:

```
P JOIN S                |    SELECT PNO , PNAME , COLOR , WEIGHT ,
                        |           P.CITY , SNO , SNAME , STATUS
                        |    FROM   P , S
                        |    WHERE  P.CITY = S.CITY
```

Напомню, однако, что SQL также позволяет записывать соединение и по-другому, в виде, чуть более близком к **Tutorial D** (и на этот раз я сознательно вместо длинного списка ссылок на столбцы во фразе SELECT употребил просто «*»):

```
SELECT *
FROM   P NATURAL JOIN S
```

В этой формулировке результирующий заголовок состоит из атрибутов, или столбцов CITY, PNO, PNAME, COLOR, WEIGHT, SNO, SNAME и STATUS (в SQL, но не в **Tutorial D**, они следуют именно в таком порядке).

В связи с операцией естественного соединения уместно будет сделать несколько замечаний. Прежде всего, отметим, что его частным случаем является пересечение (то есть *r1* INTERSECT *r2* – частный случай *r1* JOIN *r2* в терминах **Tutorial D**). Точнее, этот случай возникает, когда отношения *r1* и *r2* не просто соединяемые, а имеют в точности один и тот же тип. Впрочем, к операции INTERSECT я еще вернусь в этой главе.

Далее, произведение тоже является частным случаем соединения (то есть *r1* TIMES *r2* – частный случай *r1* JOIN *r2* в терминах **Tutorial D**). Точнее, этот случай возникает, когда отношения *r1* и *r2* не имеют общих имен атрибутов. Почему? Да потому что в этом случае (а) множество общих атрибутов пусто, (б) любой кортеж с пустым множеством атрибутов имеет одно и то же значение (а именно, 0-кортеж) и, значит, (в) любой кортеж в *r1* соединяется с любым кортежем в *r2*, то есть мы получаем именно то, что называется произведением. Но для полноты картины я все же приведу определение.

> **Определение**. *Декартовым произведением* (или для краткости просто *произведением*) отношений *r1* и *r2*, *r1* TIMES *r2*, где *r1* и *r2* не имеют атрибутов с одинаковыми именами, называется отношение, для которого (а) заголовком является теоретико-множественное объединение заголовков *r1* и *r2* и (б) телом является множество всех кортежей *t*, таких, что *t* есть теоретико-множественное объединение некоторого кортежа из *r1* и некоторого кортежа из *r2*.

Например:

```
( P RENAME ( CITY AS PCITY ) )    |    SELECT PNO , PNAME , COLOR ,
  TIMES  /* или JOIN */          |       WEIGHT , P.CITY AS PCITY,
( S RENAME ( CITY AS SCITY ) )    |       SNO , SNAME , STATUS ,
                                  |       S.CITY AS SCITY
                                  |    FROM   P , S
```

Обратите внимание, что по крайней мере один из двух атрибутов CITY в этом примере необходимо переименовать. Результирующий заголовок содержит атрибуты, или столбцы, PNO, PNAME, COLOR, WEIGHT, PCITY, SNO, SNAME, STATUS и SCITY (в SQL именно в таком порядке).

И наконец, соединение – принципиально двуместный оператор, однако возможно и полезно определить *n*-местный вариант этого оператора (и в языке **Tutorial D** это сделано), что позволило бы записывать выражения вида

```
JOIN { r1 , r2 , ... , rn }
```

для соединения любого количества отношений *r1*, *r2*, ..., *rn*.[1] Например, соединение деталей и поставщиков можно было записать и в таком виде:

```
JOIN { P , S }
```

Более того, такой синтаксис можно использовать для «соединения», в котором участвует всего одно отношение или даже вовсе нет отноше-

[1] Для полноты отмечу, что **Tutorial D** поддерживает также *n*-местные варианты операторов INTERSECT и TIMES.

ний! Соединение одного отношения, JOIN{*r*}, – по определению, просто само *r*; этот случай, наверное, не представляет практического интереса (?). Однако, как это удивительно, соединение без указания отношений, JOIN{}, – весьма важная операция, результатом которой является TABLE_DEE! (Напомню, что TABLE_DEE – единственное отношение, не имеющее атрибутов и состоящее всего из одного кортежа.) Почему же результатом является TABLE_DEE? Рассмотрим следующие доводы.

- В обычной арифметике 0 называется *нейтральным элементом* относительно операции «+», то есть для любого числа *x* выражения x + 0 и 0 + x равны *x*. Отсюда следует, что *сумма пустого множества чисел равна 0*. (Чтобы понять, почему это так, представьте себе программу, которая вычисляет сумму чисел: она присваивает сумме начальное значение 0, а затем в цикле перебирает все числа. Ну и что произойдет, если *n* = 0?)
- Аналогично 1 является нейтральным элементом относительно операции «*», то есть для любого числа *x* выражения *x * 1* и *1 * x* равны *x*. Следовательно, произведение пустого множества чисел равно 1.
- В реляционной алгебре *TABLE_DEE – нейтральный элемент относительно операции JOIN*, то есть соединение любого отношения *r* с TABLE_DEE равно самому отношению *r* (см. ниже). Следовательно, соединение пустого множества отношений равно TABLE_DEE.

Не тревожьтесь, если сейчас вам трудно переварить эту идею. Но если вы впоследствии станете перечитывать этот раздел, попытайтесь убедить себя в том, что *r* JOIN TABLE_DEE и TABLE_DEE JOIN *r* действительно совпадают с *r*. Может быть, будет проще, если заметить, что рассматриваемые соединения на самом деле являются декартовыми произведениями (правильно?).

Явные операторы JOIN в SQL

В SQL ключевое слово JOIN применяется для обозначения различных операций соединения (хотя их всегда можно записать и без него). Слегка упрощая, можно сказать, что имеются следующие возможности, которые я пронумеровал для удобства ссылок впоследствии (здесь *t1* и *t2* – таблицы, *bx* – булево выражение, а *C1*, *C2*, ..., *Cn* – столбцы, общие для *t1* и *t2*):

1. `t1 NATURAL JOIN t2`
2. `t1 JOIN t2 ON bx`
3. `t1 JOIN t2 USING ( C1 , C2 , ... , Cn )`
4. `t1 CROSS JOIN t2`

Я ненадолго задержусь на этих четырех случаях, потому что между ними есть тонкие различия, которые трудно запомнить.

1. Случай 1 как раз и был рассмотрен выше.
2. Случай 2 логически эквивалентен такой формулировке:

```
( SELECT * FROM t1 , t2 WHERE bx )
```

3. Случай 3 логически эквивалентен случаю 2, в котором *bx* принимает вид

```
t1.C1 = t2.C1 AND t1.C2 = t2.C2 AND ... AND t1.Cn = t2.Cn
```

   с тем отличием, что *C1*, *C2*, ..., *Cn* встречаются в результате только один раз, а не дважды, а порядок столбцов в заголовке результата в общем случае иной: сначала идут столбцы *C1*, *C2*, ..., *Cn* (в этом порядке), затем остальные столбцы *t1* в том порядке, в котором они встречаются в *t1*, а затем остальные столбцы *t2* в том порядке, в котором они встречаются в *t2*. (Теперь начинаете понимать, сколько мороки с этим упорядочением слева направо?)

4. Наконец, случай 4 логически эквивалентен такой формулировке:

```
( SELECT * FROM t1 , t2 )
```

Рекомендации:

1. При записи соединений предпочитайте случай 1 (NATURAL JOIN) всем остальным (но убедитесь, что столбцы с одинаковыми именами имеют одинаковые типы). Отмечу, что запись с NATURAL JOIN зачастую будет самой краткой, если следовать и другим рекомендациям, предлагаемым в этой книге.
2. Избегайте случая 2 (JOIN ON), поскольку он гарантированно порождает результат с повторяющимися именами столбцов (кроме случая, когда таблицы *t1* и *t2* изначально не имеют общих столбцов). Но если вам позарез нужно воспользоваться именно случаем 2, – а так бывает, например, когда требуется сформулировать запрос с оператором «больше»[1], – то не забывайте о переименовании, например:

```
SELECT TEMP.*
FROM ( S JOIN P ON S.CITY > P.CITY ) AS TEMP
     ( SNO , SNAME , STATUS , SCITY ,
       PNO , PNAME , COLOR , WEIGHT , PCITY )
```

3. В случае 3 требуется, чтобы столбцы с одинаковыми именами имели одинаковые типы.
4. В случае 4 следите за тем, чтобы не было общих имен столбцов.

Во всех четырех случаях операнды *t1* и *t2* задаются с помощью конструкции, которая в SQL называется *ссылкой на таблицу*. Пусть *tr* – такая ссылка. Если табличное выражение в *tr* представляет собой подзапрос, то *tr* должна также включать фразу AS, даже если «корреляци-

[1] Соединение с оператором «больше» – частный случай так называемого θ-соединения, о котором мы будем говорить ниже в этой главе.

онное имя», определяемое этой фразой, больше нигде не упоминается (дополнительную информацию см. в главе 12). Например:

```
( SELECT SNO , CITY FROM S ) AS TEMP1
  NATURAL JOIN
( SELECT PNO , CITY FROM P ) AS TEMP2
```

И последнее. Имейте в виду, что явное обращение к JOIN не может встречаться в SQL ни (а) в качестве «независимого» табличного выражения (то есть находящегося на самом внешнем уровне вложенности), ни (б) в качестве табличного выражения в скобках, которое составляет подзапрос.

Объединение, пересечение и разность

Операции объединения, пересечения и разности (UNION, INTERSECT и MINUS в **Tutorial D**; UNION, INTERSECT и EXCEPT в SQL) устроены одинаково. Я начну с объединения.

Объединение

> **Определение**: Пусть отношения *r1* и *r2* имеют одинаковый тип; тогда их *объединением r1* UNION *r2* называется отношение такого же типа, тело которого состоит из всех кортежей, встречающихся в *r1*, в *r2* или в обоих.

Например (во всех примерах из этого раздела я предполагаю, что в отношении «детали» есть дополнительный атрибут STATUS типа INTEGER):

```
P { STATUS , CITY } UNION      |    SELECT STATUS , CITY
S { CITY , STATUS }            |    FROM   P
                               |    UNION  CORRESPONDING
                               |    SELECT CITY , STATUS
                               |    FROM   S
```

Как и в случае проекции, имеет смысл явно подчеркнуть, что из объединения «устраняются дубликаты». Отметим, что для достижения этого эффекта в SQL необязательно указывать DISTINCT; хотя UNION предлагает те же два варианта, что и SELECT (DISTINCT и ALL), по умолчанию в данном случае подразумевается DISTINCT, а не ALL (в главе 4 было сказано, что в случае SELECT дело обстоит «с точностью до наоборот»). Результирующий заголовок содержит атрибуты, или столбцы, STATUS и CITY (в SQL именно в таком порядке). Отмечу, что спецификатор CORRESPONDING в SQL позволяет игнорировать тот факт, что в таблицах-операндах столбцы могут быть расположены в другом порядке.

Рекомендации:

- Следите за тем, чтобы соответственные столбцы имели одинаковые имена и типы.

- Всюду, где возможно, включайте спецификатор CORRESPONDING.[1] Если это невозможно (например, потому что продукт не поддерживает этот спецификатор), то хотя бы располагайте столбцы строго друг под другом, как показано ниже:

```
SELECT STATUS , CITY FROM P
UNION
SELECT STATUS , CITY FROM S  /* обратите внимание на изменение */
                             /* порядка */
```

- Включайте опцию «BY (список имен столбцов)» в спецификатор CORRESPONDING, только если она не изменяет семантики выражения (например, в данном примере включение BY (STATUS,CITY) ничего не изменило бы);
- Никогда не указывайте ALL. *Примечание*: Обычная причина для задания ALL в операторе UNION состоит не в том, что пользователь хочет видеть строки-дубликаты. Напротив, он точно знает, что никаких дубликатов во входных данных нет, то есть объединение дизъюнктное, и просто пытается избавить систему от лишней работы по устранению дубликатов, которых заведомо не существует. Другим словами, все упирается в производительность. См. обсуждение этой тематики в разделе «Как избежать дубликатов в SQL» главы 4.

Язык **Tutorial D** поддерживает также операцию «дизъюнктного объединения» (D_UNION) – вариант объединения, при котором требуется, чтобы операнды не содержали одинаковых кортежей. Например:

```
S { CITY } D_UNION P { CITY }
```

На наших тестовых данных это выражение привело бы к ошибке во время выполнения, поскольку «города поставщиков» и «города деталей» не дизъюнктны. В SQL нет прямого аналога оператору D_UNION.

Кроме того, **Tutorial D** поддерживает *n*-местные варианты операторов UNION и D_UNION. Детали я здесь опущу.

Пересечение

> **Определение**: Пусть отношения *r1* и *r2* имеют одинаковый тип; тогда их *пересечением r1* INTERSECT *r2* называется отношение такого же типа, тело которого состоит из всех кортежей, встречающихся одновременно в *r1* и *r2*.

Например:

```
P { STATUS , CITY } INTERSECT    |    SELECT    STATUS , CITY
S { CITY , STATUS }              |    FROM      P
                                 |    INTERSECT CORRESPONDING
```

[1] В предыдущих главах я опускал CORRESPONDING, потому что там это лишь отвлекало бы от основной темы.

```
|    SELECT    CITY , STATUS
|    FROM      S
```

Все комментарии и рекомендации, приведенные в разделе «Объединение», можно дословно повторить для пересечения. *Примечание*: Как мы уже видели, пересечение – всего лишь частный случай соединения. Но и **Tutorial D**, и SQL поддерживают эту операцию, пусть даже по чисто психологическим причинам. В сноске выше упоминалось, что **Tutorial D** поддерживает также *n*-местный вариант, но детали я опущу.

Разность

> **Определение**: Пусть отношения *r1* и *r2* имеют одинаковый тип; тогда их *разностью r1* MINUS *r2* называется отношение такого же типа, тело которого состоит из всех кортежей, встречающихся в *r1*, но не встречающихся в *r2*.

Например:

```
P { STATUS , CITY } MINUS      |    SELECT STATUS , CITY
S { CITY , STATUS }            |    FROM   P
                               |    EXCEPT CORRESPONDING
                               |    SELECT CITY , STATUS
                               |    FROM   S
```

Все комментарии и рекомендации, приведенные в разделе «Объединение», можно дословно повторить для разности.

Какие операторы являются примитивными?

Вот и все операторы, которые я хотел рассмотреть в этой главе. Однако выше я оговорился, что не все они примитивные – некоторые можно определить через другие. Один из возможных примитивных наборов включает ограничение, проекцию, соединение (или произведение), объединение и разность. *Примечание*: Возможно, вас удивило отсутствие в этом списке переименования. Но на самом деле переименование не является примитивом, хотя на данный момент еще рано объяснять, почему (см. упражнение 7.3 в главе 7). Тем не менее этот пример показывает, что между «примитивным» и «полезным» есть разница! Я определенно не хотел бы остаться без такого полезного оператора переименования, пусть даже он не является примитивным.

Пошаговое конструирование выражений

Рассмотрим следующее выражение языка **Tutorial D** (соответствующее запросу «Получить пары номеров поставщиков, находящихся в одном городе»):

```
( ( ( S RENAME ( SNO AS SA ) ) { SA , CITY } JOIN
```

```
( S RENAME ( SNO AS SB ) ) { SB , CITY } )
    WHERE SA < SB ) { SA , SB }
```

Результат содержит два атрибута, SA и SB (отметим, что было бы достаточно одного переименования, я сделал два просто для симметрии). У условия SA < SB двоякая цель:

- Исключить пары номеров поставщиков вида (*a*,*a*);
- Гарантировать, что результат не будет содержать одновременно пары (*a*,*b*) и (*b*,*a*).

А теперь я приведу другую формулировку этого запроса, чтобы показать, как имеющаяся в языке **Tutorial D** конструкция WITH позволяет упрощать выражения, которые без нее могли бы оказаться довольно сложными:

```
WITH ( S RENAME ( SNO AS SA ) ) { SA , CITY } AS R1 ,
     ( S RENAME ( SNO AS SB ) ) { SB , CITY } AS R2 ,
     R1 JOIN R2 AS R3 ,
     R3 WHERE SA < SB AS R4 :
     R4 { SA, SB }
```

Как видно из примера, фраза WITH в **Tutorial D** состоит из ключевого слова WITH, за которым следует список спецификаций вида *expression* AS *name*, а после списка ставится двоеточие. Для каждой спецификации «*expression* AS *name*» вычисляется выражение *expression*, и результат присваивается (концептуально) временной переменной с именем *name*. Отметим, что любая спецификация в списке может ссылаться на имена, введенные ранее встретившимися спецификациями из того же списка. Отметим также, что WITH не является оператором реляционной алгебры, это просто способ упростить запись сложных выражений (особенно с общими подвыражениями). Я часто буду пользоваться этой конструкций в последующих главах.

SQL также поддерживает конструкцию WITH со следующими отличиями:

- Операнды записываются в противоположном порядке: WITH *name* AS *expression*, ..., *name* AS *expression*.
- Двоеточие-разделитель не используется.
- В **Tutorial D** WITH может использоваться с выражениями любого вида, а в SQL – только с табличными выражениями.

Кроме того, в SQL за частью *name* спецификации «*name* AS *expression*» может следовать необязательный список имен столбцов в скобках (как в определении переменной кортежа (range variable), см. главу 12), но если вы будете придерживаться прочих рекомендаций из этой книги, то этот список вам никогда не понадобится.

Вот как выглядит этот пример на SQL:

```
WITH T1 AS ( SELECT SNO AS SA , CITY
             FROM   S ) ,
     T2 AS ( SELECT SNO AS SB , CITY
             FROM   S ) ,
     T3 AS ( SELECT *
             FROM   T1 NATURAL JOIN T2 ) ,
     T4 AS ( SELECT *
             FROM   T3
             WHERE  SA < SB )
     SELECT SA , SB
     FROM   T4
```

В чем смысл реляционных выражений?

Вспомните, о чем мы говорили в предыдущей главе: каждой переменной-отношению соответствует некоторый *предикат переменной-отношения*, который и составляет смысл этой переменной-отношения. Например, предикат переменной-отношения S звучит так:

> *Поставщик SNO связан контрактом, называется SNAME, имеет статус STATUS и находится в городе CITY.*

Но я не упомянул, что это понятие естественным образом обобщается на произвольные реляционные выражения. Рассмотрим, к примеру, проекцию поставщиков на все атрибуты, кроме CITY:

```
S { SNO , SNAME , STATUS }
```

Это выражение обозначает отношение, содержащее все кортежи вида

```
TUPLE { SNO sno , SNAME sn , STATUS st }
```

такие, что в текущий момент времени в переменной-отношении S существует кортеж вида

```
TUPLE { SNO sno , SNAME sn , STATUS st , CITY sc }
```

для некоторого значения *sc* атрибута CITY. Иначе говоря, результат представляет экстенцию следующего предиката в текущий момент времени (см. главу 5, если вы забыли, что такое экстенция предиката):

> *Существует такой город CITY, что поставщик SNO связан контрактом, называется SNAME, имеет статус STATUS и находится в городе CITY.*

Этот предикат, следовательно, и представляет смысл реляционного выражения (проекции) S{SNO,SNAME,STATUS}. Отметим, что у него всего три параметра, и соответствующее отношение имеет всего три атрибута; CITY – это не параметр предиката, а то, что в логике называют «связанной переменной» из-за того, что она «квантифицирована» фразой *Существует такой город* (о связанных переменных и кванто-

рах см. главу 10).[1] *Примечание*: быть может, мысль о том, что предикат имеет всего три, а не четыре параметра, станет более понятной, если заметить, что этот предикат логически эквивалентен такому:

> *Поставщик SNO связан контрактом, называется SNAME, имеет статус STATUS и находится в некотором городе (а в каком, мы не знаем).*

Аналогичное замечание справедливо для любого реляционного выражения. Точнее: с любым реляционным выражением *rx* ассоциирован некий смысл, или предикат, причем предикат для *rx* можно определить, зная предикаты всех переменных-отношений, встречающихся в выражении, и семантику примененных к ним реляционных операций. В качестве упражнения предлагаю вам вернуться к некоторым реляционным (или записанным на SQL) выражениям в этой главе и определить, как выглядит соответствующий предикат в каждом случае.

Вычисление табличных выражений в SQL

Помимо естественного соединения, Кодд с самого начала определил оператор θ-соединения, где θ может обозначать любой скалярный оператор сравнения («=», «≠», «<» и т. д.). На самом деле θ-соединение не является примитивом, оно определяется как ограничение произведения. Вот, например, соединение поставщиков и деталей с оператором «не равно» (здесь θ – это «≠»):

```
( ( S RENAME ( CITY AS SCITY ) )  |   SELECT SNO , SNAME, STATUS,
 TIMES                            |          S.CITY AS SCITY, PNO,
 ( P RENAME ( CITY AS PCITY ) ) ) |          PNAME, COLOR, WEIGHT,
WHERE SCITY ≠ PCITY               |          P.CITY AS PCITY
                                  |   FROM   S , P
                                  |   WHERE  S.CITY <> P.CITY
```

Я хотел бы заострить внимание на SQL-формулировке. Мы можем считать, что в SQL выражение реализуется тремя шагами.

1. Выполняется фраза FROM и формируется произведение таблиц S и P. *Примечание*: Если бы мы делали это реляционно, то должны были бы переименовать атрибуты CITY до вычисления произведения. SQL позволяет выполнять переименование после, потому что столбцы таблиц упорядочены слева направо, поэтому два столбца CITY можно различить по позиции. Но для простоты не будем обращать на эту деталь внимания.

---

[1] Один из рецензентов спросил, почему атрибут CITY вообще упоминается в этом предикате, коль скоро он не является частью результата проекции. Это важный вопрос! Короткий ответ звучит так: потому что результат получен отбрасыванием имени атрибута CITY. А более развернутый можно найти в моей книге «Logic and Databases: The Roots of Relational Theory» (Trafford, 2007), стр. 387–391.

2. Далее выполняется фраза WHERE и формируется ограничение этого произведения, из которого исключены строки, где значения обоих городов одинаковы. *Примечание*: Если бы под θ понимался оператор «=», а не «≠» (в SQL «<>»), то этот шаг выглядел бы так: ограничить произведение, *оставив* только строки, в которых оба города равны, то есть получилось бы так называемое *эквисоединение* поставщиков и деталей по городу. Иначе говоря, эквисоединение – это θ-соединение, в котором вместо θ подставлено «=». *Упражнение*: В чем разница между эквисоединением и естественным соединением?
3. Наконец, выполняется фраза SELECT и формируется проекция ограничения на столбцы, указанные во фразе SELECT. (В данном примере SELECT производит переименование некоторого вида, а в общем случае, как я отмечал выше, SELECT поддерживает и дополнительную функциональность, но для простоты я эти детали опускаю.)

Таким образом, по крайней мере в первом приближении фраза FROM соответствует произведению, фраза WHERE – ограничению, а фраза SELECT – проекции, и, значит, выражение SELECT - FROM - WHERE в целом описывает проекцию ограничения произведения. Следовательно, я только что дал нестрогое, но достаточно формальное определение семантики выражений SELECT - FROM – WHERE в SQL. Можно также сказать, что я описал *концептуальный алгоритм* вычисления таких выражений. Правда, ниоткуда не следует, что в конкретной реализации необходимо использовать именно этот алгоритм; напротив, алгоритм может быть любым, лишь бы он гарантированно давал тот же результат, что концептуальный. И как правило, имеются основательные причины, обычно связанные с производительностью, для применения иных алгоритмов – например, можно вычислять фразы в другом порядке или еще как-то переписывать исходный запрос. Однако автор реализации имеет право на такие вольности, *только если может доказать, что используемый алгоритм логически эквивалентен концептуальному*. На самом деле, один из аспектов работы оптимизатора как раз и заключается в том, чтобы отыскать алгоритм, который гарантированно эквивалентен концептуальному, но работает быстрее... что подводит нас к следующему разделу.

Трансформация выражений

В этом разделе я хочу более пристально взглянуть на то, чем занимается оптимизатор. Точнее, я хочу рассказать, что скрывается за трансформацией одного реляционного выражения в другое, логически ему эквивалентное. *Примечание*: При обсуждении дубликатов в главе 4 я уже упоминал, что выполнение таких трансформаций – одна из задач оптимизатора; на самом деле трансформации составляют содержание одной из двух основных идей, лежащих в основе реляционной оптимизации (вторая, выходящая за рамки этой книги, касается использова-

ния «статистики базы данных» для выполнения так называемой оптимизации по стоимости).

Начну с тривиального примера. Рассмотрим следующее выражение языка **Tutorial D** (словами этот запрос можно выразить так: «Получить поставщиков, которые поставляют деталь P2, вместе с количеством поставленных деталей», а аналог на SQL я для простоты опустил):

```
( ( S JOIN SP ) WHERE PNO = 'P2' ) { ALL BUT PNO }
```

Предположим, что имеется 100 поставщиков и 1 000 000 поставок, из которых 500 относятся к детали P2. Если это выражение вычислять «в лоб», то есть безо всякой оптимизации, то последовательность событий будет такой:

1. *Соединить S и SP.* На этом шаге нужно прочитать 100 кортежей поставщиков; для каждого из 100 поставщиков прочитать 1 000 000 кортежей поставок; построить промежуточный результат, содержащий 1 000 000 кортежей, и записать полученные 1 000 000 кортежей на диск. (Для простоты я предполагаю, что кортежи физически хранятся именно в таком виде и что в качестве приемлемого показателя производительности можно взять «количество операций чтения и записи кортежей». Оба предположения не слишком реалистичны, но на мое рассуждение этот факт большого влияния не оказывает.)
2. *Ограничить результат, полученный на шаге 1.* На этом шаге придется прочитать 1 000 000 кортежей, но результат будет содержать всего 500 кортежей, которые, как я предполагаю, можно сохранить в оперативной памяти. (На шаге 1 я предполагал, реалистично или нет, что 1 000 000 кортежей промежуточного результата нельзя разместить в оперативной памяти.)
3. *Спроецировать результат, полученный на шаге 2.* На этом шаге не требуется ни читать, ни записывать кортежи, так что мы можем его игнорировать.

Следующая процедура эквивалентна описанной выше в том смысле, что порождает тот же самый конечный результат, но, очевидно, гораздо эффективнее.

1. *Ограничить SP только кортежами для детали P2.* На этом шаге требуется прочитать 1 000 000 кортежей поставок, но результат будет содержать всего 500 кортежей, которые можно сохранить в оперативной памяти.
2. *Соединить S с результатом, полученным на шаге 1.* На этом шаге требуется прочитать 100 кортежей поставщиков (только один раз, а не по одному разу для каждой поставки P2, так как все поставки P2 хранятся в памяти). Результат содержит 500 кортежей (тоже в памяти).
3. *Спроецировать результат, полученный на шаге 2.* Этот шаг опять-таки можно игнорировать.

В первой из описанных процедур требуется 102 000 000 операций чтения и записи кортежей, а во второй – всего 1 000 100. Стало быть, вторая процедура более чем в 100 раз быстрее первой. Ясно, что реализация, в которой используется вторая процедура, предпочтительнее! Ее смысл заключается в трансформации исходного выражения

```
( S JOIN SP ) WHERE PNO = 'P2'
```

(я игнорирую завершающую проекцию, поскольку она к рассуждению не относится) в такое:

```
S JOIN ( SP WHERE PNO = 'P2' )
```

Оба выражения логически эквивалентны, но, как мы видели, сильно отличаются с точки зрения производительности. Мы хотели бы, чтобы система умела трансформировать первое выражение во второе еще до вычисления, и она, конечно, умеет это делать. А все потому, что в реляционной алгебре, которая является высокоуровневым формальным механизмом, действуют различные *правила трансформации*; например, одно из них гласит, что операцию соединения, за которой следует ограничение, можно трансформировать в операцию ограничения, за которой следует соединение (именно этим правилом я и воспользовался в примере выше). Хороший оптимизатор все эти правила знает и применяет, поскольку в идеале производительность запроса не должна зависеть от синтаксической формы его записи. *Примечание*: На самом деле возможность трансформации одних выражений в другие – прямое следствие того, что не все алгебраические операторы примитивны (например, выражение, содержащее пересечение, можно трансформировать в выражение, содержащее соединение), но этим, как я надеюсь, понятно, дело отнюдь не исчерпывается.

Правил трансформации много, и здесь не место для их исчерпывающего рассмотрения. Я лишь хочу остановиться на нескольких наиболее важных случаях и подчеркнуть ключевые моменты. Во-первых, правило, упомянутое в предыдущем абзаце, – частный случай более общего правила, называемого *дистрибутивным* законом. В общем случае одноместный оператор f называется *дистрибутивным* относительно двуместного оператора g, если $f(g(a,b)) = g(f(a),f(b))$ для любых a и b. Например, в обычной арифметике операция извлечения неотрицательного квадратного корня SQRT дистрибутивна относительно умножения, так как

```
SQRT ( a * b ) = SQRT ( a ) * SQRT ( b )
```

для любых a и b (берем SQRT в качестве f и «*» в качестве g). Поэтому оптимизатор числовых выражений всегда может заменить одно из этих выражений другим при выполнении трансформации выражений. В качестве контрпримера отметим, что операция SQRT не является дистрибутивной относительно сложения, поскольку в общем случае квадратный корень из суммы $a + b$ не равен сумме квадратных корней из a и b.

В реляционной алгебре ограничение дистрибутивно относительно пересечения, объединения и разности. Оно дистрибутивно также относительно соединения при условии, что условие ограничения не сложнее, чем конъюнкция (AND) двух различных условий, по одному для каждого из двух операндов. В примере выше это требование удовлетворялось – на самом деле условие ограничения было совсем простым и применялось лишь к одному операнду, – поэтому мы имели право воспользоваться дистрибутивным законом для замены исходного выражения более эффективным. В результате нам удалось «выполнить раннее ограничение». Раннее ограничение – почти всегда хорошо, поскольку позволяет уменьшить количество кортежей, просматриваемых следующей операцией, и, возможно, количество кортежей в результате этой операции.

Упомянем еще некоторые частные случаи дистрибутивного закона, на этот раз касающиеся проекции. Во-первых, проекция дистрибутивна относительно объединения, но не относительно пересечения и разности. Во-вторых, она дистрибутивна относительно соединения при условии, что все атрибуты, по которым производится соединение, включены в проекцию. Эти законы можно использовать для «раннего проецирования», которое также полезно по тем же соображениям, что и ограничение.

Есть еще два общих закона: *коммутативности* и *ассоциативности*.

- Двуместный оператор *g* называется *коммутативным*, если *g(a,b) = g(b,a)* для любых *a* и *b*. Например, в обычной арифметике сложение и умножение коммутативны, а вычитание и деление – нет. В реляционной алгебре коммутативными являются операции пересечения, объединения и соединения, но не разности.[1] Так, например, если запрос содержит соединение двух отношений *r1* и *r2*, то коммутативный закон утверждает, что неважно, какое из них является «внешним», а какое – «внутренним». Поэтому при вычислении соединения система вправе взять в качестве внешнего отношения то, которое меньше (к примеру).
- Двуместный оператор *g* называется *ассоциативным*, если *g(a,g(b,c)) = g(g(a,b),c)* для любых *a*, *b*, *c*. В арифметике сложение и умножение ассоциативны, а умножение и деление – нет. В реляционной алгебре ассоциативными являются операции пересечения, объединения и соединения, но не разности. Так, например, если запрос содержит со-

[1] Строго говоря, аналоги этих операторов в SQL не коммутативны, поскольку (и это не единственная причина) порядок столбцов результата зависит от того, какой операнд задан первым; рекомендуемые мной правила работы с этим операторами направлены, в частности, на то, чтобы уйти от подобных проблем. Более общо, возможность возникновения таких проблем – одна из многих причин, по которым не рекомендуется писать на SQL код, зависящий от порядка столбцов.

единение трех отношений *r1*, *r2* и *r3*, то сочетание ассоциативного и коммутативного закона позволяет соединять эти отношения попарно в любом порядке. Поэтому система вправе решить, какая из возможных последовательностей наиболее эффективна.

Попутно отметим, что все эти трансформации можно выполнять безотносительно к тому, как организовано физическое хранение данных и какие имеются пути доступа (индексы и т. п.). Другими словами, трансформации описывают такие оптимизации, которые гарантированно корректны без учета физической структуры базы данных.

Зависимость от имен атрибутов

Существует один вопрос, который, возможно, не дает вам покоя, но еще не обсуждался в этой главе. Операторы реляционной алгебры, по крайней мере, те, что описаны в этой книге, существенно зависят от именования атрибутов. Например, в языке **Tutorial D** выражение *R1* JOIN *R2*, где просто для определенности предположим, что *R1* и *R2* – базовые переменные-отношения, по определению выполняет соединение по атрибутам *R1* и *R2* с одинаковыми именами. Но часто возникает вопрос: не слишком ли ненадежен такой подход? Например, что случится, если впоследствии мы добавим в переменную-отношение *R2* новый атрибут с таким же именем, как у существующего атрибута *R1*?

Для начала проясню один момент. Да, действительно, операторы зависят, и существенно, от правильного именования атрибутов. Однако они требуют, чтобы одноименные атрибуты имели также одинаковые типы (то есть, формально говоря, это был один и тот же атрибут). Таким образом, если бы в выражении *R1* JOIN *R2* каждая из переменных-отношений *R1* и *R2* содержала атрибут *A*, но эти атрибуты *A* были бы разных типов, то должна была бы возникнуть ошибка – надеюсь, на этапе компиляции.[1] Отметим, что это требование (однотипности одноименных атрибутов) не налагает серьезных ограничений на функциональность благодаря наличию оператора RENAME.

Теперь вернемся к существу вопроса. Здесь существует распространенное заблуждение, и я с удовольствием воспользуюсь случаем, чтобы развеять его. В современных SQL-системах доступ из прикладной программы к базе данных обычно производится либо с помощью интерфейса уровня вызовов, либо с помощью встроенного, но концептуально независимого подъязыка доступа к данным («встроенный SQL»). Но по существу встроенный SQL – это все тот же интерфейс уровня вызовов, только прикрытый синтаксической глазурью, поэтому с точки зрения СУБД,

[1] На самом деле в SQL такая ошибка может и не возникнуть, потому что SQL допускает приведение типов; но в языке **Tutorial D** приведения запрещены, поэтому для него это наблюдение, безусловно, справедливо.

да и с точки зрения включающего языка оба пути ведут в одно и то же место. Другими словами, СУБД и включающий язык в большинстве современных систем слабо связаны между собой. В результате утрачиваются многие преимущества хорошо спроектированного и структурированного языка программирования. Приведу цитату:[1] «Большая часть программных ошибок в приложениях баз данных должна была бы проявляться как *ошибки типизации*, если бы определение базы данных входило составной частью в структуру типов программы».

Истоки того, что в современных системах определение базы данных не является «частью структуры типов программы», следует искать в фундаментальном недопонимании, которое существовало в кругах исследователей СУБД в начале 1960-х годов. В то время считалось, что для достижения независимости от данных (точнее, *логической* независимости от данных, см. главу 9) необходимо вынести определения объектов баз данных из программы, так, чтобы хотя бы в принципе определение можно было впоследствии изменить, не меняя программу. Но такая точка зрения, по крайней мере отчасти, неправильна. В действительности и тогда, и теперь необходимо иметь *два разных определения* – одно внутри программы, а другое вне ее. Внутреннее определение должно было бы представлять взгляд программиста на базу данных (и обеспечивать на этапе компиляции необходимые проверки запросов и т. д.), а внешнее – «истинную» базу данных. Тогда, если впоследствии возникает необходимость изменить определение «истинной» базы данных, то независимость от данных сохраняется за счет изменения соответствия между обоими определениями.

Вот как только что описанный механизм мог бы выглядеть в SQL. Для начала введем представление *открытой таблицы* (public table), которая соответствует взгляду приложения на некоторую часть базы данных. Например:

```
CREATE PUBLIC TABLE X            /* гипотетический синтаксис! */
  ( SNO   VARCHAR(5)  NOT NULL ,
    SNAME VARCHAR(25) NOT NULL ,
    CITY  VARCHAR(20) NOT NULL ,
    UNIQUE ( SNO ) ) ;

CREATE PUBLIC TABLE Y            /* гипотетический синтаксис! */
  ( SNO   VARCHAR(5)  NOT NULL ,
    PNO   VARCHAR(6)  NOT NULL ,
    UNIQUE ( SNO , PNO ) ) ,
    FOREIGN KEY ( SNO ) REFERENCES X ( SNO ) ) ;
```

[1] Из доклада Atsushi Ohori, Peter Buneman, Val Breazu-Tannen «Database Programming in Machiavelli – A Polymorphic Language with Static Type Inference», Proc. ACM SIGMOD International Conference on Management of Data, Portland, Ore. (Июнь 1989).

В этих определениях, по существу, утверждается, что «приложение верит», будто таблицы в базе данных о поставщиках и деталях называются X и Y, с указанными столбцами и ключами. На самом деле это не так, но в базе данных существуют таблицы S и SP (в них имеются столбцы и ключи, заданные для X и Y соответственно, и еще по одному дополнительному столбцу), и мы можем определить следующие соответствия:

```
X ::= SELECT SNO , SNAME , CITY FROM S ; /* гипотетический синтаксис!*/

Y ::= SELECT SNO , PNO FROM SP ;         /* гипотетический синтаксис!*/
```

Эти соответствия определены вне приложения (символ «::=» здесь означает «определено как»). Очевидно, что соединение здесь выполняется по общему столбцу SNO. И если впоследствии в таблицу SP будет добавлен, например, столбец SNAME, то нам нужно будет лишь изменить соответствие, – а в данном примере вообще не потребуется никаких изменений! – и все будет работать, как раньше. Другими словами, независимость от данных налицо.

К сожалению, нынешние продукты на основе SQL работают иначе. Так, например, SQL-выражение S NATURAL JOIN SP, как это ни грустно, подвержено той самой «ненадежности», с которой все и началось (как и более простое выражение SELECT * FROM S, если уж на то пошло). Однако проблему можно сгладить, если придерживаться стратегии, предложенной, когда мы обсуждали именование столбцов в главе 3. Для удобства я повторю ее еще раз:

- Для каждой базовой таблицы определите представление с идентичной структурой, отличающееся разве что тем, что некоторые столбцы переименованы.
- Убедитесь, что определенные таким образом представления подчиняются правилу именования столбцов, описанному в той же главе 3.
- Работайте с представлениями, а не с базовыми таблицами.

Если впоследствии таблицы изменятся, вам придется лишь внести соответствующие изменения в определения представлений.

Упражнения

Упражнение 6.1. Что неправильно (или все правильно?) в следующих SQL-выражениях (как с точки зрения реляционной теории, так и независимо от нее):

a. `SELECT * FROM S , SP`

b. `SELECT SNO , CITY FROM S`

c. `SELECT SNO , PNO , 2 * QTY FROM SP`

d. `SELECT S.SNO FROM S , SP`

e. `SELECT S.SNO , S.CITY FROM S NATURAL JOIN P`

f. `SELECT CITY FROM S UNION SELECT CITY FROM P`

g. `SELECT S.* FROM S NATURAL JOIN SP`

Упражнение 6.2. Свойство замкнутости важно для реляционной модели по тем же причинам, по которым замкнутость относительно операций с числами важна в обычной арифметике. Но в арифметике есть один случай, когда замкнутость в некотором смысле нарушается, – деление на нуль. Существует ли аналогичная ситуация в реляционной алгебре?

Упражнение 6.3. Если взять базу данных о поставщиках и деталях, то каким будет значение выражения JOIN{S,SP,P} в языке **Tutorial D**? А как выглядит соответствующий предикат? И как вы записали бы это соединение на языке SQL?

Упражнение 6.4. Как вы думаете, почему оператор проекции назван именно так?

Упражнение 6.5. Если взять базу данных о поставщиках и деталях, то какие значения обозначают следующие выражения **Tutorial D**? В каждом случае дайте (а) аналог на SQL и (б) неформальную интерпретацию выражения (то есть соответствующий ему предикат) на естественном языке.

a. `( S JOIN ( SP WHERE PNO = 'P2' ) ) { CITY }`

b. `( P { PNO } MINUS ( SP WHERE SNO = 'S2' ) { PNO } ) JOIN P`

c. `S { CITY } MINUS P { CITY }`

d. `( S { SNO, CITY } JOIN P { PNO, CITY } ) { ALL BUT CITY }`

e.
```
JOIN { ( S RENAME ( CITY AS SC ) ) { SC } ,
       ( P RENAME ( CITY AS PC ) ) { PC } }
```

Упражнение 6.6. Объединение, пересечение, произведение и соединение – коммутативные и ассоциативные операции. Проверьте эти утверждения. Справедливы ли они для SQL?

Упражнение 6.7. Определение какого из описанных в этой главе операторов не зависит от понятия равенства кортежей?

Упражнение 6.8. В SQL фраза FROM *t1*, *t2*, ..., *tn* (где каждое *ti* обозначает таблицу) возвращает произведение своих аргументов. Но что будет в случае $n = 1$? Что такое произведение всего одной таблицы? И, кстати, что понимать под произведением *t1* и *t2*, если и *t1*, и *t2* содержат строки-дубликаты?

Упражнение 6.9. Напишите на **Tutorial D** и/или SQL выражения для следующих запросов к базе данных о поставщиках и деталях:

a. Получить все поставки.

b. Получить номера тех поставщиков, которые поставляют деталь P1.

c. Получить поставщиков, статус которых находится в диапазоне от 15 до 25 включительно.

d. Получить номера деталей, поставляемых поставщиками, которые находятся в Лондоне.

e. Получить номера деталей, не поставляемых ни одним поставщиком, находящимся в Лондоне.

f. Получить все пары номеров деталей такие, что существует поставщик, поставляющий обе детали в паре.

g. Получить номера поставщиков, статус которых ниже статуса поставщика S1.

h. Получить номера деталей, поставляемых всеми поставщиками, которые находятся в Лондоне.

i. Получить пары (SNO,PNO), такие, что поставщик SNO не поставляет деталь PNO.

j. Получить поставщиков, которые поставляют как минимум все детали, поставляемые поставщиком S2.

Упражнение 6.10. Докажите следующие утверждения (уточнив их там, где необходимо):

a. Последовательность ограничений данного отношения можно трансформировать в единственное ограничение.

b. Последовательность проекций данного отношения можно трансформировать в единственную проекцию.

c. Ограничение проекции можно трансформировать в проекцию ограничения.

Упражнение 6.11. Объединение называют *идемпотентным* оператором, потому что *r* UNION *r* тождественно равно *r* для любого *r*. (Верно ли это в SQL?) Свойство идемпотентности может оказаться полезным для трансформации выражений. Какие еще реляционные операторы идемпотентны?

Упражнение 6.12. Пусть *r* – отношение. Что означает выражение *r*{} в языке **Tutorial D** (то есть какой ему соответствует предикат)? Что оно возвращает? А что означает в **Tutorial D** выражение *r*{ALL BUT}, и что оно возвращает?

Упражнение 6.13. Булево выражение

```
x > y AND y > 3
```

(которое могло бы быть частью запроса) эквивалентно (и, следовательно, может быть трансформировано) следующему:

```
x > y AND y > 3 AND x > 3
```

Эквивалентность основывается на том факте, что оператор сравнения «>» *транзитивен*. Отметим, что эту трансформацию заведомо

имеет смысл выполнять, если x и y взяты из разных отношений, потому что тогда система может произвести дополнительное ограничение ($x > 3$) перед тем, как приступать к соединению с оператором «больше», которое предполагается условием $x > y$. В тексте главы мы видели, что раннее ограничение – это хорошо; когда система может *вывести* дополнительные «ранние» ограничения, это тоже неплохо. Знаете ли вы какие-нибудь продукты на основе SQL, в которых такая оптимизация действительно выполняется?

Упражнение 6.14. Рассмотрим следующее выражение языка **Tutorial D**:

```
WITH ( P WHERE COLOR = 'Purple' ) AS PP ,
     ( SP RENAME ( SNO AS X ) ) AS T :
S WHERE ( T WHERE X = SNO ) { PNO } ⊇ PP { PNO }
```

Что оно означает? Покажите, какой результат получается для наших тестовых данных. Согласуется ли этот результат с вашим пониманием смысла выражения? Обоснуйте свой ответ.

Упражнение 6.15. В SQL не существует прямого аналога оператора D_UNION. Как лучше всего эмулировать этот оператор в SQL?

Упражнение 6.16. Что вы понимаете под термином *соединяемые*? Как можно было бы обобщить это понятие на случай произвольного количества отношений (а не только двух – этот случай обсуждался в тексте главы)?

Упражнение 6.17. Какое именно свойство открывает возможность определить n-местные варианты операторов JOIN и UNION (и D_UNION)? Есть ли у него аналог в SQL? Почему не имеет смысла n-местный вариант оператора MINUS?

Упражнение 6.18. Ранее я уже говорил, что TABLE_DEE означает TRUE, а TABLE_DUM – FALSE. Обоснуйте и/или развейте эти утверждения.

7

SQL и реляционная алгебра II: дополнительные операции

Как я уже неоднократно говорил, оператором реляционной алгебры называется такой оператор, который принимает на входе одно или несколько отношений и порождает на выходе новое отношение. Однако в главе 1 отмечалось, что можно определить произвольное количество операторов, отвечающих такой простой характеристике. В главе 6 были описаны оригинальные операторы (соединение, проекция и т. д.), а в этой главе мы рассмотрим некоторые из многочисленных дополнительных операторов, которые были определены уже после изобретения реляционной модели. Мы также обсудим, как эти операторы лучше всего реализовать в SQL.

Полусоединение и полуразность

Соединение – один из наиболее известных реляционных операторов. Но на практике часто бывает, что в тех запросах, где вообще необходимо соединение, возникает потребность в обобщенной форме этого оператора, которая называется полусоединением (возможно, вы никогда не слышали о полусоединении, но это достаточно важный оператор).

> **Определение**: *Полусоединение* отношений *r1* и *r2* (в таком порядке), *r1* MATCHING *r2*, эквивалентно операции (*r1* JOIN *r2*){*A*,*B*,...,*C*}, где *A*, *B*, ..., *C* – все атрибуты *r1*.

Иначе говоря, *r1* MATCHING *r2* – это результат соединения *r1* и *r2*, спроецированный на атрибуты *r1*. Рассмотрим пример («Получить всех поставщиков, которые в настоящий момент поставляют хотя бы одну деталь»):

```
S MATCHING SP                |   SELECT S.* FROM S
                             |   WHERE  SNO IN
                             |        ( SELECT SNO FROM SP )
```

Заголовок результата такой же, как у S. Отметим, что выражения *r1* MATCHING *r2* и *r2* MATCHING *r1* в общем случае не эквивалентны. Отметим также, что можно было бы заменить ключевое слово IN в SQL словом MATCH; однако любопытно, что заменить NOT IN на NOT MATCH в операторе полуразности (см. ниже) не получится, потому что в SQL нет оператора «NOT MATCH».

Теперь обратимся к полуразности. Если полусоединение в каком-то отношении важнее соединения, то к полуразности это замечание относится даже в большей мере – на практике в большинстве запросов, где используется разность, на самом деле необходима полуразность.

> **Определение**: *Полуразность* между отношениями *r1* и *r2* (в таком порядке), *r1* NOT MATCHING *r2*, эквивалентна операции *r1* MINUS (*r1* MATCHING *r2*).

Вот пример («Получить всех поставщиков, который в настоящий момент не поставляют ни одной детали»):

```
S NOT MATCHING SP                |    SELECT S.* FROM S
                                 |    WHERE  SNO NOT IN
                                 |         ( SELECT SNO FROM SP )
```

Опять-таки заголовок результата такой же, как у S. *Примечание*: Если отношения *r1* и *r2* одного типа, то *r1* NOT MATCHING *r2* вырождается в *r1* MINUS *r2*; другими словами, в реляционной теории разность (MINUS) – это частный случай полуразности. Однако соединение не является частным случаем полусоединения, это совершенно разные операторы, хотя неформально можно сказать, что некоторые соединения являются полусоединениями, а некоторые полусоединения – соединениями. См. упражнение 7.19 в конце главы.

Расширение

Возможно, вы обратили внимание, что в описанной до сих пор алгебре нет традиционных средств вычислений. А в SQL они есть; например, никто не мешает написать запрос такого вида: SELECT A + B AS C ... . Однако, написав знак «+», мы сразу же вышли за рамки первоначально определенной алгебры. Поэтому, чтобы получить такого рода функциональность, нам необходимо что-то добавить в алгебру, и это «что-то» как раз и есть оператор EXTEND. Предположим, например, что веса деталей (в отношении P) выражены в фунтах, и что мы хотим получить их в граммах. Поскольку в фунте 454 грамма, то можно написать такой запрос:

```
EXTEND P                         |    SELECT P.* ,
   ADD ( WEIGHT * 454 AS GMWT )  |           WEIGHT * 454 AS GMWT
                                 |    FROM   P
```

Для наших тестовых данных получится такой результат:

PNO	PNAME	COLOR	WEIGHT	CITY	GMWT
P1	Nut	Red	12.0	London	5448.0
P2	Bolt	Green	17.0	Paris	7718.0
P3	Screw	Blue	17.0	Oslo	7718.0
P4	Screw	Red	14.0	London	6356.0
P5	Cam	Blue	12.0	Paris	5448.0
P6	Cog	Red	19.0	London	8626.0

Важно: Переменная-отношение P в базе данных не изменяется! EXTEND – не аналог SQL-предложения ALTER TABLE; выражение EXTEND – это всего лишь выражение, и, как всякое выражение, служит для обозначения некоторого значения.

Продолжая этот пример, рассмотрим запрос «Получить номер детали и вес в граммах для тех деталей, для которых вес в граммах больше 7000 граммов»:

```
( ( EXTEND P ADD                |    SELECT PNO ,
    ( WEIGHT * 454 AS GMWT ) ) |           WEIGHT * 454 AS GMWT
        WHERE GMWT > 7000.0 )  |    FROM   P
                { PNO , GMWT } |    WHERE  WEIGHT * 454 > 7000.0
```

Как видите, в SQL-версии выражение WEIGHT * 454 встречается дважды, и мы можем лишь надеяться, что реализация достаточно изощренна, чтобы понять, что это выражение нужно вычислять не два, а один раз для каждого кортежа (или строки). В версии же для языка **Tutorial D** выражение встречается только один раз.

Проблема, которую иллюстрирует этот пример, заключается в том, что конструкция SELECT - FROM - WHERE в SQL слишком жесткая. В действительности нам здесь необходимо – из формулировки на **Tutorial D** это совершенно очевидно – выполнить ограничение расширения; в терминах SQL нужно применить оператор WHERE к результату SELECT. Но шаблон SELECT - FROM - WHERE вынуждает применять фразу WHERE к результату фразы FROM, а не фразы SELECT. Выразим ту же мысль по-другому: во многих отношениях весь смысл алгебры (благодаря замкнутости) состоит в том, чтобы реляционные операции можно было произвольным образом сочетать и вкладывать друг в друга. Но шаблон SELECT - FROM - WHERE в SQL означает по сути дела, что выразить можно только те запросы, в которых сначала выполняется произведение, потом ограничение, а потом некоторая комбинация проекции и/или расширения, и/или разгруппирования, и/или переименования. Однако существует много запросов, не укладывающихся в такую схему.

Кстати, возможно, вам интересно, почему я не записал SQL-версию в таком виде:

```
SELECT PNO , WEIGHT * 454 AS GMWT
FROM   P
WHERE  GMWT > 7000.0
```

(изменена последняя строчка). Причина в том, что GMWT – имя столбца в *конечном результате*; в таблице P такого столбца нет, поэтому фраза WHERE не имеет смысла и вызовет ошибку на этапе компиляции.

На самом деле стандарт SQL допускает формулировку рассматриваемого запроса в виде, чуть более близком к **Tutorial D** (ниже для ясности я привожу полные имена):

```
SELECT TEMP.PNO , TEMP.GMWT
FROM ( SELECT P.PNO , ( P.WEIGHT * 454 ) AS GMWT
       FROM P ) AS TEMP
WHERE  TEMP.GMWT > 7000.0
```

Но не во всех SQL-системах разрешается употреблять вложенные подзапросы во фразе FROM. Отмечу также, что такая формулировка неизбежно ведет к необходимости ссылаться на некоторые переменные (в данном случае TEMP) еще до того, как они определены, а в реальных SQL-запросах, быть может, задолго до точки определения.

Я завершаю этот раздел формальным определением:

> **Определение:** Пусть *r* – отношение. *Расширением* EXTEND *r* ADD (*exp* AS X) называется отношение, для которого (а) заголовок совпадает с заголовком *r*, дополненным атрибутом *X*, и (б) тело состоит из множества всех кортежей *t*, таких что *t* есть кортеж *r*, дополненный значением атрибута *X*, которое получается вычислением выражения *exp* для этого кортежа *r*. В отношении *r* не должно быть атрибута с именем *X* и *exp* не должно ссылаться на *X*. Отметим, что кардинальность результата равна кардинальности *r*, а степень равна степени *r* плюс 1. Типом *X* в результате является тип *exp*.

Отношения-образы

Отношение-образ – это, неформально говоря, «образ» некоторого кортежа внутри некоторого отношения (обычно кортеж принадлежит какому-то другому отношению). Например, для базы данных о поставщиках и деталях следующее отношение является образом кортежа поставщика S4 (из отношения «поставщики») в отношении «поставки»:

PNO	QTY
P2	200
P4	300
P5	400

Ясно, что это отношение-образ можно получить с помощью такого выражения на языке **Tutorial D**:

```
( SP WHERE SNO = 'S4' ) { ALL BUT SNO }
```

Приведем формальное определение отношения-образа в общем случае.

> **Определение**: Пусть *r1* и *r2* – соединяемые отношения (то есть одноименные атрибуты имеют один и тот же тип); пусть *t1* – кортеж *r1*, а *t2* – кортеж *r2*, для которого значения общих с кортежем *t1* атрибутов такие же, как в кортеже *t1*. Пусть отношение *r3* – это ограничение *r2*, которое содержит все такие и только такие кортежи *t2*, и пусть *r4* – проекция *r3* на все атрибуты, кроме общих. Тогда *r4* называется *отношением-образом* (относительно *r2*), соответствующим *t1*.

Следующий пример иллюстрирует полезность отношений-образов:

```
S WHERE ( !!SP ) { PNO } = P { PNO }
```

Отметим попутно, что здесь булево выражение во фразе WHERE является реляционным сравнением.

Объяснение:

- Во-первых, в качестве *r1* и *r2* из определения выше выступают отношения «поставщики» и «поставки» соответственно (под «отношением “поставщики”» я понимаю текущее значение переменной-отношения S, и аналогично для «отношения “поставки”»).
- Далее, мы можем представить себе, что булево выражение во фразе WHERE вычисляется для каждого кортежа *t1* из *r1* (то есть для каждого кортежа в отношении «поставщики») по очереди.
- Рассмотрим один такой кортеж, скажем, для поставщика S*x*. Тогда для этого кортежа выражение !!SP обозначает соответствующее отношение-образ *r4* внутри *r2*; иными словами, это множество пар (PNO,QTY) отношения SP для деталей, поставляемых поставщиком S*x*. Выражение !!SP представляет собой *ссылку на отношение-образ*.
- Выражение (!!SP){PNO}, то есть проекция отношения-образа на {PNO}, обозначает, следовательно, множество номеров деталей для деталей, поставляемых поставщиком S*x*.
- Таким образом, все выражение в целом (то есть S WHERE ...) обозначает поставщиков из S, для которых это множество номеров деталей равно множеству всех номеров деталей в проекции P на {PNO}. Иначе говоря, оно представляет запрос «Получить поставщиков, которые поставляют все детали» (допуская некоторую вольность формулировки).

Примечание

Поскольку понятие отношения-образа определяется в терминах некоторого заданного кортежа (в формальном определении он назван *t1*), то понятно, что ссылка на отношение-образ может встречаться не во всех контекстах, где допустимы общие реляционные выражения, а только в некоторых, точнее в тех, где кортеж *t1* имеет смысл. Примером такого контекста могут служить фразы WHERE, как показывает приведенный выше пример. Другой пример мы увидим в разделе «Еще об отношениях-образах».

Отступление

В SQL нет прямой поддержки отношений-образов как таковых. Но для интересующихся покажу аналог приведенного выше выражения **Tutorial D** (я включил его только для справки и не собираюсь вдаваться в детали, замечу лишь, что, очевидно (?), есть много способов улучшить его):

```
SELECT *
FROM   S
WHERE  NOT EXISTS
     ( SELECT PNO
       FROM   SP
       WHERE  SP.SNO = S.SNO
       EXCEPT
       SELECT PNO
       FROM   P )
AND    NOT EXISTS
     ( SELECT PNO
       FROM   P
       EXCEPT
       SELECT PNO
       FROM   P
       WHERE  SP.SNO = S.SNO )
```

Но вернемся к отношениям-образам. Стоит отметить, что оператор «!!» можно определить в терминах MATCHING. Например, для рассмотренного выше примера выражение

```
S WHERE ( !!SP ) { PNO } = P { PNO }
```

логически эквивалентно такому выражению:

```
S WHERE ( SP MATCHING RELATION { TUPLE { SNO SNO } } ) { PNO } = P { PNO }
```

Объяснение: Снова рассмотрим некоторый кортеж S, скажем, для поставщика S*x*. Для этого кортежа выражение TUPLE{SNO SNO} – вызов селектора кортежа – обозначает кортеж, содержащий только значение атрибута SNO кортежа S*x* (первое вхождение SNO – имя атрибута, второе – значение атрибута с этим именем в кортеже S*x* переменной-отношения S). Таким образом, выражение

```
RELATION { TUPLE { SNO SNO } }
```

представляет собой вызов селектора отношения и обозначает отношение, содержащее только этот кортеж. Следовательно, выражение

```
SP MATCHING RELATION { TUPLE { SNO SNO } }
```

обозначает некоторое ограничение SP, а именно ограничение, которое содержит только те кортежи отношения «поставки», в которых значение SNO такое же, как в кортеже поставщика S*x* из отношения «поставщики». Что и требовалось доказать.

Предположим теперь, что база данных о поставщиках несколько изменена (одновременно обобщена и упрощена) и выглядит следующим образом (в общих чертах):

```
S   { SNO }       /* поставщики                       */
SP  { SNO, PNO }  /* поставщик поставляет деталь      */
PJ  { PNO, JNO }  /* деталь используется в проекте */
J   { JNO }       /* проекты                          */
```

Здесь переменная-отношение J представляет проекты (JNO – номер проекта), а переменная-отношение PJ описывает, какие детали в каких проектах используются. Рассмотрим запрос «Получить все пары (*sno*,*jno*), такие что *sno* – значение SNO, входящее в данный момент времени в переменную-отношение S, *jno* – значение JNO, входящее в данный момент времени в переменную-отношение J, и поставщик *sno* поставляет все детали, использующиеся в проекте *jno*». Это сложный запрос! Но с применением отношений-образов он записывается почти тривиально:

```
( S JOIN J ) WHERE !!PJ ⊆ !!SP
```

Вернемся к обычной базе данных о поставщиках и деталях и рассмотрим еще один запрос («Удалить поставки от поставщиков, находящихся в Лондоне», и на этот раз я приведу также аналог на SQL):

```
DELETE SP WHERE IS_NOT_EMPTY   |   DELETE FROM SP
  ( !!(S WHERE                 |   WHERE  SNO IN
        CITY = 'London') ) ;   |         ( SELECT SNO FROM S
                               |           WHERE CITY = 'London' ) ;
```

Для данной поставки заданное отношение-образ !!(S WHERE ...) либо пусто, либо содержит ровно один кортеж.

Деление

Я включил в эту главу обсуждение операции деления только для того, чтобы показать, почему (быть может, вразрез с общепринятым мнением) я не считаю ее очень важной; я даже думаю, что от нее следовало бы отказаться. Если хотите, можете пропустить этот раздел.

У меня есть несколько причин (по меньшей мере, три) желать исключения операции деления. Во-первых, любой запрос, который можно

выразить в терминах деления, можно сформулировать, причем гораздо проще, в терминах отношений-образов, что я скоро и продемонстрирую. Во-вторых, существует по меньшей мере семь различных операторов деления, то есть по меньшей мере семь различных операторов, которые претендуют на название «деление», и я, безусловно, не хочу рассказывать о каждом. Вместо этого я ограничусь базовым, самым простым определением.

> **Определение:** Пусть отношения *r1* и *r2* таковы, что заголовок {*Y*} отношения *r2* является некоторым подмножеством заголовка *r1*, и множество {*X*} состоит из остальных атрибутов *r1*. Тогда результатом *деления r1* на *r2*, *r1* DIVIDEBY *r2*,[1] называется следующее отношение:
>
> ```
> r1 { X } NOT MATCHING ( ( r1 { X } JOIN r2 ) NOT MATCHING r1 )
> ```

Например, выражение

```
SP { SNO , PNO } DIVIDEBY P { PNO }
```

(на наших обычных тестовых данных) дает:

SNO
S1

Таким образом, неформально это выражение можно охарактеризовать как представление запроса «Получить номера тех поставщиков, которые поставляют все детали» (чуть ниже я объясню, почему «неформально»). На практике, однако, нам обычно интересна вся информация о поставщиках, а не только номера, и, следовательно, деление должно сопровождаться соединением:

```
( SP { SNO , PNO } DIVIDEBY P { PNO } ) JOIN S
```

Но ведь мы уже знаем, как этот запрос сформулировать проще, воспользовавшись отношениями-образами:

```
S WHERE ( !!SP ) { PNO } = P { PNO }
```

Эта формулировка (а) более лаконична, (б) проще для понимания (по крайней мере, мне так кажется) и (в) *корректна*. Разумеется, последний пункт важнее всего, и ниже я остановлюсь на нем подробнее. Но сначала я хочу объяснить, почему эта операция называется делением. А дело в том, что если *r1* и *r2* – отношения, не имеющие общих имен атрибутов, и мы образуем произведение *r1* TIMES *r2*, а затем разделим его на *r2*, то получим снова *r1*.[2] Другими словами, произведение и деление – в некотором смысле взаимно обратные операции.

---

[1] Язык **Tutorial D** не поддерживает этот оператор напрямую, поэтому запись *r1* DIVIDEBY *r2* в этом языке синтаксически недопустима.

[2] При условии, что *r2* не пусто. А если пусто, что произойдет?

Как я уже сказал, выражение

```
SP { SNO , PNO } DIVIDEBY P { PNO }
```

можно неформально охарактеризовать как формулировку запроса «Получить номера тех поставщиков, которые поставляют все детали». На самом деле, именно этот пример часто используют для объяснения и обоснования необходимости оператора деления. Но, к сожалению, такая характеристика не вполне корректна. В действительности это выражение соответствует запросу «Получить номера поставщиков, которые поставляют хотя бы одну деталь и фактически поставляют все детали».[1] Иначе говоря, оператор деления не только сложен и недостаточно лаконичен, но еще и не решает ту задачу, для которой изначально задумывался.

Агрегатные операторы

В каком-то смысле этот раздел можно назвать отвлечением от темы, потому что обсуждаемые здесь операторы не реляционные, а скалярные – они возвращают скаляр в качестве результата.[2] Но я все же хочу сказать о них несколько слов, прежде чем вернусь в основной теме главы.

В реляционной модели агрегатным называется оператор, который порождает единственное значение из «агрегата» (то есть множества или мультимножества) значений некоторого атрибута какого-то отношения. Для COUNT(*), представляющего особый случай, таким «агрегатом» является все отношение. Приведу два примера:

```
X := COUNT ( S ) ;              |   SELECT COUNT ( * ) AS X
                                |   FROM   S

Y := COUNT ( S { STATUS } ) ;   |   SELECT COUNT ( DISTINCT STATUS )
                                |          AS Y
                                |   FROM   S
```

Сначала рассмотрим предложения языка **Tutorial D** в левой части. Для наших тестовых данных в первом предложении переменной X присва-

---

[1] Если вы не понимаете, в чем тут логическое различие, то давайте рассмотрим чуть измененный запрос «Получить поставщиков, которые поставляют все фиолетовые детали» (смысл, конечно, в том, что фиолетовых деталей в базе нет). Если фиолетовых деталей вообще не существует, то каждый поставщик поставляет их все! – даже поставщик S5, которые не поставляет никаких деталей и, следовательно, не представлен в переменной-отношении SP, а потому не может быть возвращен аналогичным выражением DIVIDEBY. Если что-то все же осталось непонятно, подождите до главы 11, где мы продолжим обсуждение этого примера.

[2] Можно определить и нескалярные агрегатные операторы, но они выходят за рамки этой книги.

ивается значение 5 (количество кортежей в текущем значении переменной-отношения S), а во втором – переменной Y присваивается значение 3 (количество кортежей в проекции текущего значения переменной-отношения S на множество {STATUS}, то есть количество различных значений атрибута STATUS в текущем значении переменной-отношения).

В общем случае вызов агрегатного оператора в **Tutorial D** выглядит так:

```
<agg op name> ( <relation exp> [, <exp> ] )
```

В качестве операции *<agg op name>* допустимы COUNT, SUM, AVG, MAX, MIN, AND, OR и XOR (последние три применяются к агрегатам, состоящим из булевых значений). В выражении *<exp>* в любом месте, где допустим литерал, может встречаться ссылка на атрибут *<attribute ref>*. Части *<exp>* не должно быть, если в роли *<agg op name>* выступает COUNT; в противном случае ее можно опускать, только если реляционное выражение *<relation exp>* обозначает отношение степени 1, и тогда предполагается, что *<exp>* состоит из ссылки на единственный атрибут этого отношения. Примеры:

1. `SUM ( SP , QTY )`

   Это выражение обозначает сумму всех значений QTY в переменной-отношении SP (на наших тестовых данных получится 3100).

2. `SUM ( SP { QTY } )`

   Это сокращенная запись SUM(SP{QTY},QTY), которая обозначает сумму всех различных значений QTY в переменной-отношении SP (то есть 1000).

3. `AVG ( SP , 3 * QTY )`

   Это выражение отвечает на вопрос, какова была бы средняя величина поставки, если бы количество деталей в каждой поставке увеличилось втрое (ответ 775). Более общо, выражение

   ```
   agg ( rx , x )
   ```

   (где *x* – некоторое выражение, более сложное, чем просто *<attribute ref>*) по существу является сокращенной записью для:

   ```
   agg ( EXTEND rx ADD ( x AS y ) , y )
   ```

Теперь я возвращаюсь к SQL. Для удобства повторю приведенные выше примеры:

```
X := COUNT ( S ) ;              |   SELECT COUNT ( * ) AS X
                                |   FROM   S

Y := COUNT ( S { STATUS } ) ;   |   SELECT COUNT ( DISTINCT STATUS )
                                |          AS Y
                                |   FROM   S
```

Возможно, вам будет странно слышать от меня, что SQL в действительности вообще не поддерживает агрегатные операторы! Я заявляю об этом, прекрасно понимая, что многие сочтут выражения справа в точности вызовами агрегатных операторов в SQL.[1] Но это не так. И я готов объяснить. Как мы знаем, значения этих счетчиков равны соответственно 5 и 3. Но эти SQL-выражения не вычисляют сами счетчики, как должны были бы делать настоящие агрегатные операторы, а вычисляют таблицы, содержащие значения счетчиков. Точнее, каждый вызов порождает таблицу с одной строкой и одним столбцом, и единственным значением в этой строке будет счетчик:

X
5

Y
3

Как видите, выражения SELECT не являются вызовами агрегатных операторов; в лучшем случае можно сказать, что они дают аппроксимацию таких вызовов. На самом деле агрегирование в SQL рассматривается как частный случай *обобщения*. Правда, я еще не касался обобщения в этой главе, но пока вы можете считать, что оно представляется в SQL выражением SELECT с фразой GROUP BY. Конечно, в приведенных выше SQL-выражениях фразы GROUP BY нет, но по определению они являются сокращенной записью таких выражений (и потому все же являются частными случаями обобщения, как и утверждалось):

```
SELECT COUNT ( * ) AS X
FROM   S
GROUP  BY ( )

SELECT COUNT ( DISTINCT STATUS ) AS Y
FROM   S
GROUP  BY ( )
```

Примечание

На случай, если эти выражения вас смущают, объясню, что в SQL разрешается (а) заключать списки операндов GROUP BY в скобки и (б) включать фразы GROUP BY без операндов. Задание GROUP BY без операндов трактуется так, будто этой фразы нет вовсе.

SQL поддерживает обобщение, но не агрегирование как таковое. Печально, но эти понятия часто смешивают, и, наверное, теперь вы понимаете почему. Но картина запутывается еще больше из-за того, что

[1] Можно сказать, с несколько большим основанием, что *вызовы COUNT в этих выражениях* – это вызовы агрегатных операторов. Но суть в том, что в SQL такой вызов не может выступать в роли «независимого» выражения; он обязательно должен быть частью какого-то табличного выражения.

в SQL нередко приводят таблицу, являющуюся результатом «агрегирования» к одной строке, которую она содержит, или даже к тому единственному значению, которое имеется в этой строке. Две разных ошибки (по крайней мере, концептуальных) – а в результате странное «агрегирование» становится больше похоже на настоящее! Такое двойное приведение типа происходит, в частности, когда выражение SELECT заключено в скобки и образует скалярный подзапрос, как, например, в следующих присваиваниях:

```
SET X = ( SELECT COUNT ( * ) FROM S ) ;

SET Y = ( SELECT COUNT ( DISTINCT STATUS ) FROM S ) ;
```

Но присваивание – далеко не единственный контекст, в котором возможно подобное приведение типа (см. главы 2 и 12).

Отступление

На самом деле, с агрегированием в духе SQL связана еще одна странность (я поместил это замечание сюда, так как оно сюда логически относится, однако оно опирается на понимание идеи обобщения в SQL, поэтому сейчас можете его пропустить):

- В общем случае выражение вида SELECT - FROM *T* - WHERE - GROUP BY - HAVING порождает результат, содержащий по одной строке для каждой «группы» в *G*, где *G* – «сгруппированная таблица», получающаяся путем применения фраз WHERE, GROUP BY и HAVING к таблице *T*.
- Отсутствие фраз WHERE и HAVING, характерное для типичного агрегирования в SQL, эквивалентно заданию фраз WHERE TRUE и HAVING TRUE соответственно. Поэтому сейчас нам достаточно рассмотреть только влияние фразы GROUP BY на вычисление сгруппированной таблицы *G*.
- Предположим, что таблица *T* состоит из *nT* строк. При объединении этих строк в группы может получиться не более *nT* групп. Иными словами, сгруппированная таблица *G* состоит из *nG* групп, где $nG \leq nT$, а окончательный результат применения фразы SELECT к *G* состоит из *nG* строк.
- Теперь предположим, что *nT* равно нулю (то есть таблица *T* пуста); тогда *nG*, очевидно, также должно быть равно нулю (то есть таблица *G* и, стало быть, результат выражения SELECT тоже будут пусты).
- Поэтому, в частности, выражение

```
SELECT COUNT ( * ) AS X
FROM   S
GROUP  BY ( )
```

которое, напомню, является полной записью SELECT COUNT(*) AS X FROM S, по логике вещей, должно порождать результат, показанный слева, а не справа, если таблица S пуста.

X

X
0

Но на самом деле получается результат, показанный справа. Почему? *Ответ:* это особый случай. Вот прямая цитата из стандарта: «Если не заданы столбцы, по которым производится группировка, то результатом фразы <group by clause> является сгруппированная таблица, содержащая *T* в качестве единственной группы». Иными словами, хотя группировка пустой таблицы в SQL действительно (как я и доказывал выше) порождает в общем случае пустое множество групп, однако *случай, когда множество столбцов группировки пусто, считается особым*; в этом случае порождается множество, содержащее ровно одну группу, которая идентична пустой таблице T. Таким образом, в нашем примере оператор COUNT применяется к пустой группе и, значит, «корректно» возвращает значение 0.

Возможно, вам кажется, что в этом несоответствии нет ничего ужасного; может быть, вы даже думаете, что правый результат чем-то «лучше» левого. Но (и это очевидно) между тем и другим есть логическое различие, а – снова цитируя Виттгенштейна, – «*любое логическое различие является существенным различием*». Такого рода логические ошибки попросту недопустимы в системе, которая, подобно реляционной системе, должна покоиться на твердых логических основаниях.

Еще об отношениях-образах

В этом разделе я хотел бы привести ряд примеров, демонстрирующих полезность агрегатных операторов, рассмотренных в предыдущем разделе, в применении к отношениям-образам. *Примечание*: Во всех примерах в каком-то виде, пусть неявном, присутствует обобщение, но пока я еще не готов обсуждать его детально (это будет темой следующих двух разделов).

Пример 1. Получить поставщиков, для которых количество деталей в поставке, просуммированное по всем поставкам данного поставщика, меньше 1000.

```
S WHERE SUM ( !!SP , QTY ) < 1000
```

Для любого поставщика выражение SUM(!!SP,QTY) обозначает в точности общее количество деталей, поставляемых этим поставщиком. Эквивалентная формулировка без использования отношения-образа такова:

```
S WHERE SUM ( SP MATCHING RELATION { TUPLE { SNO SNO } } , QTY ) < 1000
```

Ради интереса приведу «аналог» на SQL – я взял слово «аналог» в кавычки, потому что в этом примере есть подводный камень: показанное SQL-выражение не является точным эквивалентом выражений **Tutorial D** выше (почему?):

```
SELECT S.*
FROM   S , SP
WHERE  S.SNO = SP.SNO
```

```
GROUP  BY S.SNO , S.SNAME , S.STATUS , S.CITY
HAVING SUM ( SP.QTY ) < 1000
```

Примечание

Не могу не заметить мимоходом, что (как видно из примера выше) SQL допускает запись «S.*» во фразе SELECT, но не во фразе GROUP BY, где она имела бы ничуть не меньший смысл.

Пример 2. Получить поставщиков, для которых существует меньше трех поставок.

```
S WHERE COUNT ( !!SP ) < 3
```

Пример 3. Получить поставщиков, для которых максимальное поставленное количество менее чем в два раза превышает минимальное поставленное количество (в обоих случаях речь идет обо всех поставках данного поставщика).

```
S WHERE MAX ( !!SP , QTY ) < 2 * MIN ( !!SP , QTY )
```

Пример 4. Получить поставки, для которых существует еще по меньшей мере две поставки с таким же количеством.

```
SP WHERE COUNT ( !!(SP RENAME ( SNO AS SN , PNO AS PN )) ) > 2
```

Признаю, что это очень искусственный пример, но смысл его в том, чтобы показать, что иногда при использовании отношений-образов возникает необходимость в переименовании атрибутов. В этом примере переименование необходимо, чтобы нужное нам отношение-образ, связанное с данным кортежем поставки, было определено только в терминах атрибута QTY. Имена SN и PN выбраны произвольно.

Отмечу попутно, что этот пример иллюстрирует также «множественную» форму RENAME:

```
SP RENAME ( SNO AS SN , PNO AS PN )
```

Это выражение – сокращенная запись такого:

```
( SP RENAME ( SNO AS SN ) ) RENAME ( PNO AS PN )
```

Похожие сокращения определены и для разных других операторов, в том числе EXTEND (пример будет приведен ниже).

Пример 5. Для всех поставщиков, для которых общее количество, просуммированное по всем поставкам данного поставщика, меньше 1000, уменьшить значение статуса в два раза.

```
UPDATE S WHERE SUM ( !!SP , QTY ) < 1000 :
                    { STATUS := 0.5 * STATUS } ;
```

Обобщение

> **Определение:** Пусть отношения *r1* и *r2* таковы, что заголовок *r2* совпадает с заголовком некоторой проекции *r1*, и пусть *A, B, ..., C* – атрибуты *r2*. Тогда *обобщением* SUMMARIZE *r1* PER (*r2*) ADD (*summary* AS X) называется отношение, для которого (а) заголовком является заголовок *r2*, дополненный атрибутом *X*, и (б) тело состоит из множества всех кортежей *t* таких, что *t* является кортежем *r2*, дополненным значением *x* атрибута *X*. Это значение *x* подсчитывается путем вычисления некоторого *обобщающего выражения* по всем кортежам *r1*, которые имеют те же значения атрибутов *A, B, ..., C*, что и кортеж *t*. Отношение *r2* не должно содержать атрибута с именем *X*, а *обобщающее выражение* не должно ссылаться на *X*. Отметим, что кардинальность результата равна кардинальности *r2*, а степень результата равна степени *r2* плюс единица. Типом *X* в этом случае будет тип выражения *exp*.

Вот пример (который я для последующих ссылок назову SX1 – «SUMMARIZE Example 1»):

```
SUMMARIZE SP PER ( S { SNO } ) ADD ( COUNT ( PNO ) AS PCT )
```

Для наших тестовых данных получается такой результат:

SNO	PCT
S1	6
S2	2
S3	1
S4	3
S5	0

Иными словами, результат содержит по одному кортежу для каждого кортежа в отношении PER – то есть в этом примере по одному кортежу для каждого из пяти номеров поставщиков, – дополненному соответствующим счетчиком.

Отступление

Обратите особое внимание, что синтаксическая конструкция COUNT(PNO) – я сознательно не называю ее выражением, потому что она таковым не является (по крайней мере, не в смысле языка **Tutorial D**), – в предыдущем вызове SUMMARIZE – это *не* вызов агрегатного оператора с именем COUNT. Агрегатный оператор принимает в качестве аргумента отношение, а аргумент COUNT – атрибут; конечно, это атрибут некоторого отношения, но само отношение определено лишь косвенно. На самом деле синтаксическая конструкция COUNT(PNO) представляет собой особый случай – она не имеет никакого

смысла вне контекста подходящего вызова SUMMARIZE и не может встречаться вне такого контекста. Все это наводит на мысль, что операция SUMMARIZE в каком-то смысле неполноценна и хорошо было бы заменить ее чем-то получше... См. раздел «Еще об обобщении» ниже.

Если отношение *r2* не просто имеет такой же заголовок, как некоторая проекция отношения *r1*, а является такой проекцией, то обобщение можно записать короче, заменив спецификацию PER спецификацией BY, как в следующем примере («Пример SX2»):

```
SUMMARIZE SP BY { SNO } ADD ( COUNT ( PNO ) AS PCT )
```

Результат получается таким:

SNO	PCT
S1	6
S2	2
S3	1
S4	3

Как видите, результат отличается от предыдущего – он не содержит кортежа для поставщика S5. Объясняется это тем, что BY {SNO} в этом примере, по определению, является сокращением для PER (SP{SNO}) – SP, потому что именно по SP мы и хотим произвести обобщение, – а проекция SP{SNO} не содержит кортежа для поставщика S5.

Пример SX2 можно записать на SQL следующим образом:

```
SELECT SNO , COUNT ( ALL PNO ) AS PCT
FROM   SP
GROUP  BY SNO
```

Мы видим, что обобщения – в противоположность «агрегированию» – обычно формулируются на SQL посредством выражения SELECT с явной фразой GROUP BY (однако см. ниже!). Вот на какие мысли наводит этот пример.

- Можно считать, что такие выражения вычисляются в следующем порядке. Сначала таблица, заданная фразой FROM, разбивается на множество непересекающихся «групп» – фактически таблиц, – в соответствии со столбцами группировки во фразе GROUP BY. Затем формируются результирующие строки, по одной для каждой группы: вычисляется заданное обобщающее выражение (или выражения) для этой группы и в конец дописываются другие элементы, заданные в списке SELECT. *Примечание*: В SQL аналогом термина *обобщающее выражение* служит «функция множества» (set function) – термин вдвойне неподходящий, потому что (а) аргументом такой функции является не множество, а мультимножество, и (б) результатом тоже является не множество.

- В данном примере можно без опаски употреблять SELECT, а не SELECT DISTINCT, так как (а) гарантируется, что результирующая таблица содержит в точности одну строку для каждой группы (по определению) и (б) каждая группа содержит по одному значению для каждого столбца группировки (опять-таки по определению).
- В данном примере при вызове COUNT спецификатор ALL можно опустить, так как для функций множеств его наличие предполагается по умолчанию. (Фактически, в этом примере неважно, указан ли спецификатор ALL или DISTINCT, потому что сочетание номера поставщика и номера детали является ключом для таблицы SP).
- Функция множества COUNT(*) представляет собой особый случай – она применяется не к значениям в некотором столбце (как, например, SUM), а к строкам таблицы. (В данном примере COUNT(PNO) можно было бы заменить на COUNT(*), и результат не изменился бы.)

А теперь вернемся к примеру SX1. На SQL его можно записать следующим образом:

```
SELECT S.SNO , ( SELECT COUNT ( ALL PNO )
                 FROM   SP
                 WHERE  SP.SNO = S.SNO ) AS PCT
FROM   S
```

Здесь важно отметить, что теперь результат содержит строку для поставщика S5, потому что, по определению, результат должен содержать по одной строке для каждого номера поставщика, присутствующего в таблице S. И, как легко видеть, эта формулировка отличается от примера SX2 – того, где поставщик S5 был пропущен, – тем, что отсутствует фраза GROUP BY и не производится никакая группировка (по крайней мере, явно).

Отступление

Кстати, читателей, не искушенных в SQL, тут поджидает некий подвох. Как видите, второй элемент списка SELECT в приведенном выше SQL-выражении – то есть подвыражение (SELECT ... S.SNO) AS PCT – имеет вид *подзапрос* AS *имя* (и сам подзапрос скалярный). Если бы точно такое же выражение встретилось во фразе FROM, то спецификатор AS *имя* следовало бы понимать как определение переменной кортежа (range variable) (см. главу 10). Во фразе же SELECT этот спецификатор интерпретируется как определение имени столбца результата. Отсюда следует, что следующая формулировка этого примера логически не эквивалентна приведенной выше:

```
SELECT S.SNO , ( SELECT COUNT ( ALL PNO ) AS PCT
                 FROM   SP
                 WHERE  SP.SNO = S.SNO )
FROM   S
```

При такой формулировке таблица *t*, которая возвращается в результате вычисления подзапроса, имеет столбец PCT. Для этой таблицы *t* производится

двойное приведение типа к тому единственному скалярному значению, которое в ней содержится, в результате чего порождается значение столбца в конечной таблице, и – хотите верьте, хотите нет – этот столбец не называется PCT, он вообще не имеет имени.

Но вернемся к основной теме обсуждения. Фактически, пример SX2 можно было бы записать и без использования GROUP BY следующим образом:

```
SELECT DISTINCT SPX.SNO , ( SELECT COUNT ( ALL SPY.PNO )
                            FROM   SP AS SPY
                            WHERE  SPY.SNO = SPX.SNO ) AS PCT
FROM   SP AS SPX
```

Как следует из этих примеров, фраза GROUP BY в SQL логически избыточна – любое реляционное выражение, которое можно представить с ее помощью, можно записать и без нее. Однако в связи с этим следует сделать еще одно замечание. Предположим, что в примере SX1 требуется подсчитать не общее число номеров деталей, а суммарное количество, поставленное каждым поставщиком:

```
SUMMARIZE SP PER ( S { SNO } ) ADD ( SUM ( QTY ) AS TOTQ )
```

На наших тестовых данных результат будет такой:

SNO	TOTQ
S1	1300
S2	700
S3	200
S4	900
S5	0

Но SQL-выражение

```
SELECT S.SNO , ( SELECT SUM ( ALL QTY )
                 FROM   SP
                 WHERE  SP.SNO = S.SNO ) AS TOTQ
FROM   S
```

дает результат, в котором значение TOTQ для поставщика S5 равно null, а не 0. Так получается потому, что в SQL применение любой функции множества, кроме COUNT(*) и COUNT, к пустому аргументу, по определению (неправильному) дает null. Чтобы получить правильный результат, мы должны воспользоваться функцией COALESCE:

```
SELECT S.SNO , ( SELECT COALESCE ( SUM ( ALL QTY ) , 0 )
                 FROM   SP
                 WHERE  SP.SNO = S.SNO ) AS TOTQ
FROM   S
```

Предположим теперь, что в примере SX1 требуется найти суммарное количество для каждого поставщика, но только если оно больше 250:

```
( SUMMARIZE SP PER ( S { SNO } ) ADD ( SUM ( QTY ) AS TOTQ ) )
                                                    WHERE TOTQ > 250
```

Результат:

SNO	TOTQ
S1	1300
S2	700
S4	900

«Естественная» формулировка этого запроса на SQL выглядела бы так:

```
SELECT SNO , SUM ( ALL QTY ) AS TOTQ
FROM   SP
GROUP  BY SNO
HAVING SUM ( ALL QTY ) > 250  /* не TOTQ > 250 !!! */
```

Но можно переписать его и в таком виде:

```
SELECT DISTINCT SPX.SNO , ( SELECT SUM ( ALL SPY.QTY )
                            FROM   SP AS SPY
                            WHERE  SPY.SNO = SPX.SNO ) AS TOTQ
FROM   SP AS SPX
WHERE  ( SELECT SUM ( ALL SPY.QTY )
         FROM   SP AS SPY
         WHERE  SPY.SNO = SPX.SNO ) > 250
```

Как видно из этого примера, фраза HAVING, как и GROUP BY, тоже логически избыточна – любое реляционное выражение, которое можно представить с ее помощью, можно записать и без нее. Да, версии с GROUP BY и HAVING часто более компактны; с другой стороны, верно и то, что иногда они дают неправильные ответы (подумайте, например, что произойдет, если в предыдущем примере требуется, чтобы суммарное количество было не больше, а меньше 250).

Рекомендации: Применяя фразы GROUP BY и HAVING, убедитесь, что вы обобщаете именно ту таблицу, которую хотите обобщить (в примерах из этого раздела – таблицу поставщиков, а не поставок). Кроме того, не забывайте о возможности обобщения по пустому множеству и включайте COALESCE там, где необходимо.

Есть и еще одна вещь, о которой я хочу сказать в связи с GROUP BY и HAVING. Рассмотрим следующее SQL-выражение:

```
SELECT SNO , CITY , SUM ( ALL QTY ) AS TOTQ
FROM   S NATURAL JOIN SP
GROUP  BY SNO
```

Заметим, что CITY встречается с списке SELECT, но не входит в состав столбцов группировки. Но такая ситуация допустима, потому что

таблица S удовлетворяет некоторой *функциональной зависимости* (см. главу 8 или приложение B), согласно которой каждому значению SNO в этой таблице соответствует ровно одно значение CITY (в ней же). Более того, в стандарте SQL есть правила, согласно которым система знает о наличии такой функциональной зависимости. Поэтому, несмотря на отсутствие CITY в составе столбцов группировки, известно, что в каждой группе значение CITY одно и то же, так что CITY может встречаться во фразе SELECT, как показано выше (и во фразе HAVING, если бы таковая присутствовала).

Разумеется, логически допустимо (хотя это может негативно отразиться на производительности) включить этот столбец в состав столбцов группировки:

```
SELECT SNO , CITY , SUM ( QTY ) AS TOTQ
FROM   S NATURAL JOIN SP
GROUP  BY SNO , CITY
```

Еще об обобщении

Оператор SUMMARIZE входил в состав языка **Tutorial D** с самого начала. Но с появлением отношений-образов он стал логически избыточным, и хотя, возможно, есть причины сохранить его (например, в педагогических целях), факт остается фактом: как правило, обобщение более компактно записываются с помощью оператора EXTEND.[1] Вспомните пример SX1 из предыдущего раздела («Для каждого поставщика получить номер поставщика и количество поставленных им деталей»). Формулировка с применением SUMMARIZE выглядит так:

```
SUMMARIZE SP PER ( S { SNO } ) ADD ( COUNT ( PNO ) AS PCT )
```

А вот как выглядит эквивалентное выражение с применением оператора EXTEND:

```
EXTEND S { SNO } ADD ( COUNT ( !!SP ) AS PCT )
```

(Поскольку комбинация {SNO,PNO} является ключом переменной-отношения SP, необязательно проецировать отношение-образ на {PNO} перед вычислением счетчика.) Как видно из этого примера, оператор EXTEND дает еще один контекст, в котором отношения-образы имеют смысл; пожалуй, в этом контексте они даже полезнее, чем во фразах WHERE.

Далее в этом разделе приводятся дополнительные примеры. Я продолжаю нумерацию, начатую в разделе «Еще об отношениях-образах».

---

[1] Не говоря уже о том, что, как отмечалось выше в разделе об обобщении, в операторе SUMMARIZE по необходимости используется синтаксическая конструкция, которая, к несчастью, похожа на вызов агрегатного оператора, хотя таковым не является. Поэтому, как уже было сказано, было бы неплохо расстаться с оператором SUMMARIZE навсегда.

Пример 6. Для каждого поставщика получить подробную информацию о нем и общее количество поставленных им деталей, просуммированное по всем его поставкам.

```
EXTEND S ADD ( SUM ( !!SP , QTY ) AS TOTQ )
```

Пример 7. Для каждого поставщика получить подробную информацию о нем и общее, максимальное и минимальное количество деталей по всем его поставкам.

```
EXTEND S ADD ( SUM ( !!SP , QTY ) AS TOTQ ,
               MAX ( !!SP , QTY ) AS MAXQ ,
               MIN ( !!SP , QTY ) AS MINQ )
```

Обратите внимание на множественную форму EXTEND в этом примере.

Пример 8. Для каждого поставщика получить подробную информацию о нем, общее количество деталей, просуммированное по всем его поставкам, и общее количество деталей, просуммированное по всем поставкам всех поставщиков.

```
EXTEND S ADD ( SUM ( !!SP , QTY ) AS  TOTQ ,
               SUM (   SP , QTY ) AS GTOTQ )
```

Результат:

SNO	TOTQ	GTOTQ
S1	1300	3100
S2	700	3100
S3	200	3100
S4	900	3100
S5	0	3100

Пример 9. Для каждого города *c* получить *c* и общее и среднее количество деталей в поставке для всех поставок, для которых и поставщик, и деталь находятся в городе *c*.

```
WITH ( S JOIN SP JOIN P ) AS TEMP :
EXTEND TEMP { CITY } ADD ( SUM ( !!TEMP , QTY ) AS TOTQ ,
                           AVG ( !!TEMP , QTY ) AS AVGQ )
```

Смысл этого, довольно искусственного, примера – проиллюстрировать полезность WITH в сочетании с обобщением в случае, когда нужно несколько раз вывести некоторое, возможно, длинное подвыражение.

Группирование и разгруппирование

Напомню, что в главе 2 говорилось о допустимости атрибутов, значениями которых являются отношения (RVA-атрибутов). На рис. 7.1 показаны отношения R1 и R4 с рисунков 2.1 и 2.2; R4 имеет RVA-атрибуты, а R1 – нет, но очевидно, что оба отношения несут в себе одну и ту же информацию.

R1

SNO	PNO
S2	P1
S2	P2
S3	P2
S4	P2
S4	P4
S4	P5

R4

SNO	PNO_REL
S2	PNO: P1, P2
S3	PNO: P2
S4	PNO: P2, P4, P5

Рис. 7.1. *Отношения R1 и R4 с рис. 2.1 и 2.2 в главе 2*

Ясно, что необходим какой-то способ, позволяющий установить соответствие между отношениями с RVA-атрибутами и без них. Именно в этом и состоит назначение операторов GROUP и UNGROUP. Не хочу глубоко вдаваться детали, а просто скажу, что для отношений, показанных на рис. 7.1, выражение

```
R1 GROUP ( { PNO } AS PNO_REL )
```

порождает R4, а выражение

```
R4 UNGROUP ( PNO_REL )
```

порождает R1. В SQL нет прямых аналогов этим операторам.

Кстати, стоит отметить, что выражение

```
EXTEND R1 { SNO } ADD ( !!R1 AS PNO_REL )
```

дает в точности такой же результат, как в примере GROUP выше. Иными словами, GROUP можно определить в терминах EXTEND и отношений-образов. Но я вовсе не призываю отказаться от полезного оператора GROUP; даже если отвлечься от всего остального, язык, в котором есть явный оператор UNGROUP (как в **Tutorial D**), но отсутствует явный оператор GROUP, безусловно, стал бы мишенью для критики (хотя бы из «эргономических» соображений). Но все равно интересно, а, быть может, и полезно в педагогических целях отметить, как легко семантику оператора GROUP можно объяснить в терминах EXTEND и отношений-образов.

Попутное замечание: если R4 содержит ровно один кортеж для поставщика с номером S*x* и значение PNO_REL в этом кортеже пусто, то результат показанного выше оператора UNGROUP не будет содержать ни одного кортежа для поставщика с номером S*x*. За дополнительной информацией отсылаю вас к своей книге «Introduction to Database

Systems» (см. приложение D) или к книге «Databases, Types, and the Relational Model: The Third Manifesto» (снова см. приложение D), написанной Хью Дарвеном и мной.

И еще одно попутное замечание: возможно, вам интересно, как выглядят операции над отношениями с RVA-атрибутами. Операции над *любым* отношением, ссылающиеся на некоторый атрибут *A* типа *T* этого отношения, обычно подразумевают нечто, что можно было бы назвать «подоперациями» над значениями атрибута *A*, а точнее, операции, определенные для его типа *T*. Поэтому если *T* – тип отношения, то в роли «подопераций» выступают операции, отпределенные для типов отношений, то есть те самые реляционные операции (соединение и прочие), которые описаны в этой и предыдущей главах. См. упражнения 7.11 – 7.13 в конце главы.

Запросы «что если»

Поддержка запросов «что если» – часто выдвигаемое требование; они позволяют исследовать, какой эффект принесло бы некоторое изменение, не выполняя самого изменения (с последующим возвратом в исходное состояние). Вот пример («Что если бы детали, находящиеся в Париже, оказались в Ницце, а их вес удвоился бы?»):

```
UPDATE P WHERE CITY = 'Paris' :
     { CITY := 'Nice' ,
       WEIGHT := 2 * WEIGHT }
```

```
WITH T1 AS
  ( SELECT P.*
    FROM   P
    WHERE  CITY = 'Paris' ) ,
      T2 AS
  ( SELECT P.* , 'Nice' AS NC ,
           2 * WEIGHT AS NW
    FROM   T1 )
    SELECT PNO , PNAME , COLOR ,
           NW AS WEIGHT ,
           NC AS CITY
    FROM   T2
```

Отметим, что несмотря на наличие ключевого слово UPDATE, выражение языка **Tutorial D** слева – это именно выражение, а не предложение (отсутствует точка с запятой в конце), и, в частности, оно не оказывает влияния на переменную-отношение P. (Таким образом, ключевое слово UPDATE в данном случае обозначает оператор чтения! Льюис Кэрролл, где ты?) И это выражение вычисляет отношение, содержащее ровно один кортеж *t2* для каждого кортежа *t1* в текущем значении переменной-отношения P, для которого указан город Париж (Paris). Но кортеж *t2* отличается от *t1* тем, что значение веса (weight) в два раза больше, а значение города (city) равно Nice (Ницца), а не Paris. Иными словами, все это выражение является сокращенной записью для такого (кстати, полная форма поможет вам понять версию этого запроса на SQL):

```
WITH ( P WHERE CITY = 'Paris' ) AS R1 ,
     ( EXTEND R1 ADD ( 'Nice' AS NC , 2 * WEIGHT AS NW ) ) AS R2 ,
     R2 { ALL BUT CITY , WEIGHT } AS R3 :
     R3 RENAME ( NC AS CITY , NW AS WEIGHT )
```

А теперь я хочу закончить одно дело, начатое в главе 5. Там я сказал, что UPDATE – я имел в виду *предложение* UPDATE, а не только что рассмотренный оператор чтения, – соответствует некоторому реляционному присваиванию, но детали несколько сложнее, чем для INSERT и DELETE. Теперь я могу эти детали объяснить. Рассмотрим в качестве примера такое предложение UPDATE:

```
UPDATE P WHERE CITY = 'Paris' :
            { CITY := 'Nice' , WEIGHT := 2 * WEIGHT } ;
```

Логически это предложение эквивалентно следующему реляционному присваиванию (обратите внимание на обращение к рассмотренному выше оператору чтения UPDATE):

```
P := ( P WHERE CITY ≠ 'Paris' )
       UNION
     ( UPDATE P WHERE CITY = 'Paris' :
                    { CITY := 'Nice' , WEIGHT := 2 * WEIGHT } ) ;
```

А как насчет ORDER BY?

Напоследок я хочу рассмотреть в этой главе фразу ORDER BY. Эта конструкция не является частью реляционной алгебры, и как я отмечал в главе 1, ее вообще нельзя назвать реляционным оператором, потому что возвращаемый ею результат – не отношение (на входе она принимает отношение, но на выходе порождает нечто иное, а именно последовательность кортежей). Пожалуйста, не поймите меня неправильно, я вовсе не хочу сказать, что конструкция ORDER BY бесполезна; однако я *утверждаю*, что в реляционном выражении для нее нет места (разве что трактовать ее как «пустую операцию»).[1] Поэтому, по определению, следующие выражения, хотя и допустимы, реляционными выражениями не являются:

```
S MATCHING SP                    |   SELECT DISTINCT S.*
          ORDER ( ASC SNO )      |   FROM   S , SP
                                 |   WHERE  S.SNO = SP.SNO
                                 |   ORDER  BY SNO ASC
```

[1] В частности, поэтому она не может встречаться в определении представления – несмотря на то, что один хорошо известный продукт позволяет это! *Примечание*: Иногда высказывают мысль, что ORDER BY необходима для так называемых квотированных запросов, но это распространенное заблуждение (см. упражнение 7.14).

Наведя в этом вопросе ясность, я хотел бы отметить, что есть две причины, из-за которых ORDER BY (просто ORDER в **Tutorial D**) действительно оказывается довольно странным оператором. Во-первых, он по существу преобразует множество кортежей в отсортированную последовательность, а между тем, как нам известно из главы 3, для кортежей операторы «<» и «>» не определены. Все операторы реляционной алгебры – фактически все операторы чтения в обычном смысле этого слова – являются функциями, то есть для любых данных на входе возвращают единственное значение на выходе. А вот ORDER BY может порождать различные результаты для одних и тех же исходных данных. Для иллюстрации взглянем на результат применения операции ORDER BY CITY к нашему тестовому отношению «поставщики». Понятно, что эта операция может вернуть любой из четырех различных результатов, соответствующих показанным ниже последовательностям (для простоты я включил только номера поставщиков):

- S5 , S1 , S4 , S2 , S3
- S5 , S4 , S1 , S2 , S3
- S5 , S1 , S4 , S3 , S2
- S5 , S4 , S1 , S3 , S2

Кстати, было бы непростительным упущением с мой стороны не упомянуть о том, что SQL-аналоги большинства реляционных операторов тоже не являются функциями. Такое положение дел объясняется тем, что, как объяснялось в главе 2, SQL иногда считает, что результат сравнения *v1* = *v2* равен TRUE, даже если *v1* и *v2* различаются. Например, рассмотрим символьные строки ‘Paris’ и ‘Paris ’ (обратите внимание на пробел в конце второй строки); ясно, что эти значения различны, и тем не менее SQL иногда считает их равными. Поэтому в главе 2 и было сказано, что некоторые SQL-выражения «потенциально недерминированы». Вот простой пример:

```
SELECT DISTINCT CITY FROM S
```

Если для одного поставщика указан город ‘Paris’, а для другого – ‘Paris ’, то результат может содержать как ‘Paris’, так и ‘Paris ’ (а то и оба сразу), но что именно, заранее не известно. Можно даже в один день получить одно, а в другой – другое, хотя в промежутке база данных не изменялась, и это будет абсолютно законно. Можете сами поразмыслить о том, к каким последствиям способно привести такое положение вещей.

Упражнения

Упражнение 7.1. Что обозначают следующие выражения языка **Tutorial D** для нашей базы данных о поставщиках и деталях? В каждом случае дайте (а) SQL-аналог и (б) неформальную интерпретацию вы-

ражения (то есть соответствующий ему предикат) на естественном языке.

a. `S MATCHING ( SP WHERE PNO = 'P2' )`

b. `P NOT MATCHING ( SP WHERE SNO = 'S2' )`

c. `P WHERE ( !!SP ) { SNO } = S { SNO }`

d. `P WHERE SUM ( !!SP , QTY ) < 500`

e. `P WHERE TUPLE { CITY CITY } ∈ S { CITY }`

f. `EXTEND S ADD ( 'Supplier' AS TAG )`

g. `EXTEND S { SNO } ADD ( 3 * STATUS AS TRIPLE_STATUS )`

h. `EXTEND ( P JOIN SP ) ADD ( WEIGHT * QTY AS SHIPWT )`

i. `EXTEND P ADD ( WEIGHT * 454 AS GMWT , WEIGHT * 16 AS OZWT )`

j. `EXTEND P ADD ( COUNT ( !!SP ) AS SCT )`

k.
```
EXTEND S
    ADD ( COUNT ( ( SP RENAME ( SNO AS X ) ) WHERE X = SNO )
       AS NP )
```

l. `SUMMARIZE S BY { CITY } ADD ( AVG ( STATUS ) AS AVG_STATUS )`

m.
```
SUMMARIZE ( S WHERE CITY = 'London' )
             PER ( TABLE_DEE ) ADD ( COUNT ( SNO ) AS N )
```

n. `UPDATE SP WHERE SNO = 'S1' : { SNO := 'S7' , QTY = 0.5 * QTY }`

Упражнение 7.2. При каких обстоятельствах (если такое вообще возможно) выражения *r1* MATCHING *r2* и *r2* MATCHING *r1* эквивалентны?

Упражнение 7.3. Докажите, что оператор переименования не является примитивным.

Упражнение 7.4. Напишите выражение, содержащее EXTEND вместо SUMMARIZE, которое было бы эквивалентно такому:

```
SUMMARIZE SP PER ( S { SNO } ) ADD ( COUNT ( PNO ) AS NP )
```

Упражнение 7.5. Какие из следующих выражений **Tutorial D** эквивалентны? В каждом случае приведите SQL-аналог.

a. `SUMMARIZE r PER ( r { } ) ADD ( SUM ( 1 ) AS CT )`

b. `SUMMARIZE r PER ( TABLE_DEE ) ADD ( SUM ( 1 ) AS CT )`

c. `SUMMARIZE r BY { } ADD ( SUM ( 1 ) AS CT )`

d. `EXTEND TABLE_DEE ADD ( COUNT ( r ) AS CT )`

Упражнение 7.6. В языке **Tutorial D**, если аргумент при вызове агрегатного оператора – пустое множество, то COUNT, как и SUM, воз-

вращает 0; MAX и MIN возвращают соответственно минимальное и максимальное значение соответствующего типа; AND и OR возвращают соответственно TRUE и FALSE, а AVG возбуждает исключение (я сознательно ничего не говорю здесь об агрегатном операторе XOR в **Tutorial D**). Что в этих случаях возвращает SQL? И почему?

Упражнение 7.7. Пусть отношение R4 на рис. 7.1 обозначает текущее значение некоторой переменной-отношения. Если R4 описано, как в главе 2, то каков предикат этой переменной-отношения?

Упражнение 7.8. Пусть *r* – отношение, обозначаемое следующим выражением **Tutorial D**:

```
SP GROUP ( { } AS X )
```

Как выглядит *r* для нашего тестового значения SP? И что дает такое выражение?

```
r UNGROUP ( X )
```

Упражнение 7.9. Напишите на **Tutorial D** и/или SQL выражения для следующих запросов к базе данных о поставщиках и деталях:

a. Получить общее количество деталей, поставленных поставщиком S1.

b. Получить номера поставщиков, находящихся в городе, который идет первым по алфавиту в списке всех городов, где есть поставщики.

c. Получить названия городов, в которых находятся хотя бы два поставщика.

d. Поскольку оба атрибута SNAME и CITY имеют тип CHAR, имеет смысл (пусть неотчетливый) сравнивать название поставщика с названием города... Вернуть в качестве результата 'Y', если название каждого поставщика предшествует названию города, в котором он находится, при сравнении в алфавитном порядке; в противном случае вернуть 'N'.

Упражнение 7.10. Ниже приведены два выражения **Tutorial D**, в которых участвует отношение R4, изображенное на рис. 7.1. Какие запросы они представляют?

```
( R4 WHERE TUPLE { PNO 'P2' } ∈ PNO_REL ) { SNO }

( ( R4 WHERE SNO = 'S2' ) UNGROUP ( PNO_REL ) ) { PNO }
```

Упражнение 7.11. Что обозначает следующее выражение **Tutorial D** для нашей базы данных о поставщиках и деталях?

```
EXTEND S
   ADD  ( ( ( SP RENAME ( SNO AS X ) ) WHERE X = SNO ) { PNO }
        AS PNO_REL )
```

Упражнение 7.12. Пусть отношение, которое возвращает выражение из предыдущего упражнения, сохранено в переменной-отношении SSP. Что делают следующие операции обновления?

```
INSERT SSP RELATION
 { TUPLE { SNO 'S6' , SNAME 'Lopez' , STATUS 30 , CITY 'Madrid' ,
           PNO_REL RELATION { TUPLE { PNO 'P5' } } } } ;

UPDATE SSP WHERE SNO = 'S2' :
 { INSERT PNO_REL RELATION { TUPLE { PNO 'P5' } } } ;
```

Упражнение 7.13. Считая, что SSP – переменная-отношение из предыдущего упражнения, напишите выражения для следующих запросов:

a. Получить номера поставщиков, которые поставляют в точности одно и то же множество деталей.

b. Получить номера деталей, поставляемых в точности одними и теми же поставщиками.

Упражнение 7.14. *Квотированным* называется запрос, в котором задано ограничение сверху на кардинальность результата, или *квота*, например: в запросе «Получить две самых тяжелых детали» задана квота 2. Приведите формулировки этого запроса на **Tutorial D** и SQL. Что они возвращают для наших тестовых данных?

Упражнение 7.15. Как бы вы записали следующий запрос с использованием явного оператора SUMMARIZE: «Для каждого поставщика получить номер поставщика и среднюю величину поставки (количество деталей в ней) для поставок данного поставщика с различными значениями количества деталей»?

Упражнение 7.16. Рассмотрим следующую модифицированную версию базы данных о поставщиках и деталях:

```
S   { SNO }       /* поставщики                  */
SP  { SNO, PNO }  /* поставщик поставляет деталь */
SJ  { SNO, JNO }  /* поставщик поставляет проект */
J   { JNO }       /* проекты                     */
```

Выразите на **Tutorial D** и SQL такой запрос: «Для каждого поставщика получить всю информацию о нем, номера деталей, поставляемых этим поставщиком, и номера проектов, поставляемых этим поставщиком». На языке **Tutorial D** дайте формулировки с использованием операторов EXTEND и SUMMARIZE.

Упражнение 7.17. Что означает следующее выражение языка **Tutorial D**?

```
S WHERE ( !!(!!SP) ) { PNO } = P { PNO }
```

Упражнение 7.18. Существует ли логическое различие между следующими двумя выражениями языка **Tutorial D**? Если да, то в чем оно заключается?

```
EXTEND TABLE_DEE ADD ( COUNT (   SP ) AS NSP )

EXTEND TABLE_DEE ADD ( COUNT ( !!SP ) AS NSP )
```

Упражнение 7.19. Приведите пример соединения, которое не является полусоединением, и полусоединения, не являющегося соединением. Точно сформулируйте условия, при которых выражения *r1* JOIN *r2* и *r1* MATCHING *r2* эквивалентны.

Упражнение 7.20. Пусть *r1* и *r2* – отношения одного и того же типа, и пусть *t1* – кортеж *r1*. Что означает выражение !!*r2* для этого кортежа *t1*? И что будет, если отношения *r1* и *r2* не просто имеют одинаковый тип, а являются одним и тем же отношением?

Упражнение 7.21. Существует ли логическое различие между следующими SQL-выражениями, и если да, то в чем оно заключается?

```
SELECT COUNT ( * ) FROM S

SELECT SUM ( 1 ) FROM S
```

8

SQL и ограничения целостности

В предыдущих главах я несколько раз затрагивал тему ограничений целостности, а теперь пришло время заняться ею более пристально. Повторю неточное определение из главы 1: ограничением целостности (для краткости, просто ограничением) называется булево выражение, вычисление которого должно давать TRUE. Вообще говоря, ограничения получили свое название потому, что ограничивают множество значений, допустимых в качестве значения некоторой переменной, но нас здесь интересуют лишь ограничения, применяемые к переменным базы данных. Такие ограничения можно отнести к одной из двух широких категорий: ограничения типа и ограничения базы данных. По существу, ограничения типа определяют, какие значения составляют данный тип, а ограничения базы данных уточняют, какие значения могут встречаться в конкретной базе данных (здесь слово «уточняют» означает, что эти ограничения применяются в дополнение к ограничениям типа). Как обычно, я буду рассматривать все новые понятия в терминах реляционной теории и SQL.

Кстати, стоит отметить, что в общем случае ограничения можно считать формальным вариантом того, что иногда называют *бизнес-правилами*. У этого термина нет точного определения (по крайней мере, общепринятого), но, вообще говоря, бизнес-правило – это декларативное утверждение (ударение на слове «декларативное») о некотором аспекте деятельности предприятия, которое база данных должна поддерживать. Утверждения, ограничивающие значения переменных базы данных, безусловно, отвечают этому нестрогому определению. Но я пойду еще дальше. На мой взгляд, ограничения и составляют самую суть управления базами данных. Задача базы данных – представить некоторый аспект работы предприятия; это представление должно быть максимально верным, только тогда можно гарантировать, что решения, принимаемые на основе того, что предлагает база данных, правильны. А ограничения как раз дают наилучшее из имеющихся в нашем распо-

ряжении средств обеспечить верность представления. Важность ограничений трудно переоценить, как и важность надлежащей их поддержки со стороны СУБД.

Ограничения типа

В главе 2 мы видели, что составной частью определения типа является задание значений, составляющих этот тип, – именно для этого и предназначены ограничения типа. Если речь идет о системном типе, то эта задача возлагается на систему, так что говорить тут особо не о чем. А вот о типах, определенных пользователем, можно сказать больше, гораздо больше. Поэтому предположим – просто для примера, – что количество деталей в поставке описывается не системным типом INTEGER, а каким-то пользовательским типом, скажем QTY. Вот как может выглядеть его определение на языке **Tutorial D**:

```
1 TYPE QTY
2     POSSREP QPR
3           { Q INTEGER
4               CONSTRAINT Q ≥ 0 AND Q ≤ 5000 } ;
```

Объяснение:

- В строке 1 просто говорится, что мы определяем тип с именем QTY.
- В строке 2 говорится, что количество имеет «допустимое представление», которое называется QPR. Из главы 2 мы знаем, что физические представления всегда скрыты от пользователя. Однако в языке **Tutorial D** требуется, чтобы предложение TYPE содержало хотя бы одну спецификацию POSSREP, показывающую, что значения рассматриваемого типа *в принципе можно* представить каким-то конкретным способом.[1] В отличие от физических, допустимые представления, вне всякого сомнения, видны пользователю (в данном примере пользователю определенно нужно знать, что у количества есть допустимое представление QPR). Однако прошу заметить, что нигде не говорится, что заданное допустимое представление совпадает с физическим; может совпадать, а может и нет, но в любом случае пользователю это безразлично.
- В строке 3 говорится, что допустимое представление QPR состоит из единственного компонента Q типа INTEGER. Другим словами, значения типа QTY допустимо представлять целыми числами (и пользователь об этом знает).
- Наконец, в строке 4 говорится, что эти целые числа должны лежать в диапазоне от 0 до 5000 включительно. Таким образом, строки 2–4 вместе определяют, что допустимыми являются значения, которые

[1] Из этого правила есть несущественные исключения, которым нас здесь не интересуют.

можно представить целыми числами из указанного диапазона; именно это определение и составляет *ограничение типа* для типа QTY. (Заметим, что такие ограничения выражаются не в терминах самого типа, а в терминах допустимого представления этого типа. Собственно, одна из причин, по которым нам вообще нужно понятие допустимого представления, как раз и состоит в том, чтобы получить средство для выражения ограничений типа. Надеюсь, это стало понятно из рассмотрения примера.)

Вот несколько более сложный пример:

```
TYPE POINT
     POSSREP CARTESIAN { X FIXED , Y FIXED
             CONSTRAINT SQRT ( X ** 2 + Y ** 2 ) ≤ 100.0 } ;
```

Тип POINT описывает точку на двумерной плоскости, поэтому у него есть допустимое представление CARTESIAN с двумя компонентами X и Y (надо полагать, они соответствуют декартовым координатам). Эти компоненты имеют тип FIXED, и задана спецификация CONSTRAINT, смысл которой в том, что нас интересуют только точки, лежащие внутри или на границе круга с центром в начале координат и радиусом 100 (SQRT – неотрицательный квадратный корень). *Примечание*: В главе 2, если помните, я пользовался типом POINT, но тогда сознательно не показал спецификации POSSREP и CONSTRAINT; однако я молчаливо предполагал, что допустимое представление этого типа называется POINT, а не CARTESIAN. См. следующий подраздел.

Селекторы и операторы THE_

Прежде чем продолжить обсуждение ограничений типов, я хочу немного отклониться от темы и прояснить несколько вопросов, возникающих при внимательном изучении типов QTY и POINT.

В главе 3 отмечалось, что с любым типом, в том числе с пользовательским, ассоциирован селектор и операторы THE_. Эти операторы тесно связаны с понятием допустимого представления; на самом деле, существует взаимно однозначное соответствие между селекторами и допустимыми представлениями и между операторами THE_ и компонентами допустимого представления. Приведу несколько примеров.

1. `QPR ( 250 )`

   Это вызов селектора типа QTY. Селектор имеет такое же имя QPR, как единственное допустимое представление для этого типа; он принимает аргумент, который соответствует единственному компоненту допустимого представления и принадлежит тому же типу, и возвращает количество (то есть значение типа QTY). *Примечание*: На практике допустимые представления часто именуются так же, как ассоциированный с ними тип, в примере QTY я взял разные имена, чтобы высветить логическое различие между допустимым пред-

ставлением и типом, но обычно так не делают. В синтаксисе языка **Tutorial D** даже есть правило, позволяющее опускать имя допустимого представления в предложении TYPE, и тогда оно по умолчанию будет совпадать с именем ассоциированного типа. Поэтому внесем изменения в определение типа QTY:

```
TYPE QTY POSSREP { Q INTEGER CONSTRAINT Q ≥ 0 AND Q ≤ 5000 } ;
```

Теперь и допустимое представление, и соответствующий селектор называются QTY, а показанный выше вызов селектора принимает вид QTY(250) – именно так я вызывал селекторы в главе 2 (можете вернуться и убедиться). Далее в этой главе я буду опираться именно на это определение типа QTY, если явно не оговорено противное.

2. `QTY ( A + B )`

   При вызове селектора QTY в качестве аргумента можно передать выражение произвольной сложности (лишь бы оно имело тип INTEGER). Если выражение является литералом, как в предыдущем примере, то и вызов селектора – литерал; следовательно, литерал – это частный случай вызова селектора, как мы уже знаем из главы 2.

3. `THE_Q ( QZ )`

   Это вызов оператора THE_ для типа QTY. Оператор называется THE_Q, потому что Q – имя единственного компонента единственного допустимого представления типа QTY; он принимает аргумент (произвольное выражение) типа QTY и возвращает целое число – компонент Q допустимого представления аргумента.

Что касается типа POINT, то давайте сначала изменим его определение, так чтобы допустимое выражение называлось так же, как сам тип:

```
TYPE POINT POSSREP { X FIXED , Y FIXED CONSTRAINT ... } ;
```

А теперь вернемся к примерам:

1. `POINT( 5.7, -3.9 )`

   Это вызов селектора POINT (фактически литерал типа POINT).

2. `THE_X ( P )`

   Это выражение возвращает значение типа FIXED – координату X в декартовом допустимом представлении точки, являющейся текущим значением переменной P (которая должна иметь тип POINT).

Попутно хотел бы привлечь ваше внимание к тому факту, что (как я уже говорил) язык **Tutorial D** требует, чтобы в предложении TYPE была *по меньшей мере* одна спецификация POSSREP. Но **Tutorial D** разрешает задавать для одного типа несколько различных допустимых представлений. Хорошим примером может служить тип POINT; мы вполне могли бы определить два допустимых представления точки на двумерной плоскости: в декартовых и в полярных координатах. Дальнейшее углу-

бление в детали в этой книге вряд ли уместно, я просто хочу отметить, что в SQL никакого аналога этому нет.

Еще об ограничениях типа

Вернемся к самим ограничениям типа. Предположим, что тип QTY определен, как показано ниже, без явной спецификации CONSTRAINT:

```
TYPE QTY POSSREP { Q INTEGER } ;
```

По определению это не что иное, как сокращенная запись для

```
TYPE QTY POSSREP { Q INTEGER CONSTRAINT TRUE } ;
```

При таком определении все, что можно представить целым числом, будет допустимым значением QTY, и хотя с типом QTY по необходимости ассоциировано ограничение типа, оно весьма слабое. Другими словами, заданное допустимое представление уже определяет ограничение для типа, а спецификация CONSTRAINT лишь налагает добавочное ограничение помимо априорного. (Но неформально мы часто понимаем под «ограничением типа» именно то, что задано в спецификации CONSTRAINT.)

Я пока еще ничего не говорил о том, когда проверяются ограничения типа. Это делается *в момент вызова селектора*. Предположим снова, что у значений типа QTY есть допустимое представление целыми числами в диапазоне от 0 до 5000 включительно. Тогда выражение QTY(250) – вызов селектора QTY, который завершается успешно. А вычисление выражения QTY(6000), тоже являющегося вызовом селектора, приводит к ошибке. Полагаю, не вызывает сомнений, что никак нельзя смириться с выражением, которое должно бы обозначать значение некоторого типа *T*, но не обозначает; согласитесь, фраза «значение типа *T*, не являющееся значением типа *T*» содержит в себе внутреннее противоречие. Отсюда следует, что никакой переменной – и переменной-отношению, в частности, – невозможно присвоить значение неподходящего типа.

И последнее замечание в этом разделе: объявляя, что нечто имеет тип, мы тем самым налагаем на это нечто ограничение.[1] В частности, объявив, что атрибут QTY переменной-отношения SP (к примеру) имеет тип QTY, мы наложили ограничение – ни один кортеж в переменной-отношении SP не будет содержать в позиции QTY значение, не удовлетворяющее ограничению типа QTY. *Примечание*: Это ограничение на атрибут QTY – пример так называемого *ограничения атрибута*.

---

[1] Я предпочел бы использовать в этом предложении более формальный термин *объект* вместо туманного *нечто*, но это слово в информатике уж слишком перегружено.

Ограничения типа в SQL

Уверен, вы заметили, что в предыдущем разделе я не приводил примеров на SQL. А дело в том, что – хотите верьте, хотите нет – SQL вообще не поддерживает ограничения типа, если не считать тривиальные встроенные ограничения. Конечно, SQL позволит создать пользовательский тип QTY и сказать, что количество должно представляться целым числом, но указать, что эти числа должны принадлежать некоторому диапазону, вы не сможете. Иными словами, определение этого типа в SQL будет выглядеть примерно так:

```
CREATE TYPE QTY AS INTEGER FINAL ;
```

(Ключевое слово FINAL здесь означает, что у типа QTY нет собственных подтипов, но обсуждение подтипов выходит за рамки настоящей книги.)

При таком определении любое целое число может представлять количество. Если вы хотите, чтобы количество принадлежало заданному диапазону, то должны будете определять подходящее ограничение *базы данных* – на практике это, вероятно, будет ограничение на базовую таблицу (см. раздел «Ограничения базы данных в SQL») – при каждом использовании этого типа. Например, определение базовой таблицы SP, возможно, придется расширить следующим образом (обратите внимание на измененную спецификацию типа данных для столбца QTY и на спецификацию CONSTRAINT в самом конце):

```
CREATE TABLE SP
  ( SNO    VARCHAR(5)   NOT NULL ,
    PNO    VARCHAR(6)   NOT NULL ,
    QTY    QTY          NOT NULL ,
    UNIQUE ( SNO , PNO ) ,
    FOREIGN KEY ( SNO ) REFERENCES S ( SNO ) ,
    FOREIGN KEY ( PNO ) REFERENCES P ( PNO ) ,
    CONSTRAINT SPQC CHECK ( QTY >= QTY(0) AND
                            QTY <= QTY(5000) ) ) ;
```

Выражения QTY(0) и QTY(5000) в спецификации CONSTRAINT можно рассматривать как вызовы селектора QTY. Напомню, однако, что термин *селектор* в SQL не употребляется (как и термин *оператор THE_*); в главе 2 я мимоходом отметил, что ситуация с селекторами и операторами THE_ в SQL слишком сложна, чтобы детально обсуждать ее в этой книге. Скажу лишь, что обычно имеются аналоги этих операторов, хотя они и не предоставляются «автоматически», как в **Tutorial D**.

Для интересующихся читателей приведу также определение типа POINT на SQL (здесь я вместо FINAL написал NOT FINAL просто для того, чтобы показать, что это тоже возможно):

```
CREATE TYPE POINT AS
     ( X NUMERIC(5,1) , Y NUMERIC(5,1) ) NOT FINAL ;
```

Еще раз повторю, что SQL не поддерживает ограничения типа. Причины слишком сложны – они связаны с наследованием типов и потому в этой книге не рассматриваются, – но последствия весьма серьезны.[1] **Рекомендация**: По возможности используйте ограничения базы данных, чтобы компенсировать это упущение, как в примере для типа QTY выше. Конечно, эта рекомендация влечет за собой дублирование кода, но это все же лучше, чем альтернатива – некорректные данные в базе. См. упражнение 8.8 в конце главы.

Ограничения базы данных

Ограничения базы данных налагают ограничения на то, какие значения могут встречаться в базе данных. В языке **Tutorial D** подобные ограничения задаются с помощью предложения CONSTRAINT (или логически эквивалентной ему сокращенной нотации), а в SQL – с помощью предложения CREATE ASSERTION (или, опять-таки, эквивалентного сокращения). Вдаваться в детали этих сокращений я не хочу (по крайней мере, не сейчас), поскольку это не более чем синтаксические ухищрения. Так что пока покажу «длинную» нотацию. Вот несколько примеров (как обычно, слева **Tutorial D**, справа SQL):

Пример 1:

```
CONSTRAINT CX1 IS_EMPTY          |    CREATE ASSERTION CX1 CHECK
  ( S WHERE STATUS < 1           |     ( NOT EXISTS
      OR    STATUS > 100 ) ;     |     ( SELECT * FROM S
                                 |       WHERE  STATUS < 1
                                 |       OR     STATUS > 100 ) ) ;
```

Ограничение CX1 говорит: значения статуса должны принадлежать диапазону от 1 до 100 включительно. В этом ограничении участвует только один атрибут одной переменной-отношения. Особо отмечу, что для проверки ограничения для данного кортежа поставщика достаточно исследовать только этот кортеж – обращаться к другим кортежам этой или еще каких-нибудь переменных-отношений не нужно. Поэтому такие ограничения иногда неформально называют ограничениями кортежа, или ограничениями строки (в SQL), хотя в SQL этот термин применяется и в более узком смысле – как ограничение на строку, которое невозможно сформулировать в виде ограничения на один столбец (см. раздел «Ограничения базы данных в SQL»). Однако подобных ограничений

[1] Хотя я и сказал, что наследование типов не рассматривается в этой книге, не могу не указать на одно следствие, вытекающее из отсутствия поддержки ограничений типов в SQL, а именно то, что SQL вынужденно допускает такие нелепости, как «неквадратные квадраты» (точнее, значения типа SQUARE со сторонами разной длины, которые, следовательно, являются не квадратами, а прямоугольниками общего вида).

следует избегать, так как они ограничивают обновление и (как мы видели в главе 5) в реляционном мире нет понятия «обновления на уровне кортежа или строки».

Пример 2:

```
CONSTRAINT CX2 IS_EMPTY          |   CREATE ASSERTION CX2 CHECK
  ( S WHERE CITY = 'London'      |    ( NOT EXISTS
     AND   STATUS ≠ 20 ) ;       |    ( SELECT * FROM S
                                 |      WHERE  CITY = 'London'
                                 |      AND    STATUS <> 20 ) ) ;
```

Ограничение CX2 говорит: у поставщиков из Лондона должен быть статус 20. В этом ограничении участвуют два атрибута, но, поскольку это атрибуты одной и той же переменной-отношения, то, как и для ограничения CX1, для проверки достаточно исследовать всего один кортеж вне связи со всеми остальными (следовательно, это тоже «ограничение кортежа», или «ограничение строки»).

Пример 3:

```
CONSTRAINT CX3                   |    CREATE ASSERTION CX3 CHECK
   COUNT ( S ) =                 |     ( UNIQUE ( SELECT ALL SNO
   COUNT ( S { SNO } ) ;         |                FROM   S ) ) ;
```

Ограничение CX3 говорит: никакие два кортежа переменной-отношения S не могут иметь одинаковые номера; иными словами, {SNO} – это суперключ (фактически, ключ) данной переменной-отношения (напомню, что неформально суперключом называется надмножество ключа). Как и в CX1 и CX2, в этом ограничении тоже участвует только одна переменная-отношение, однако для его проверки недостаточно исследовать только кортеж данного поставщика (поэтому оно не является ограничением «кортежа», или «строки»). Конечно, на практике вряд ли кто-нибудь станет записывать ограничение CX3 так, как показано выше, – та или иная сокращенная форма с ключевым словом KEY почти всегда предпочтительнее. Я привел «длинную» форму только с одной целью: показать, что сокращения – это не более чем сокращения.

Кстати, формулировка ограничения CX3 в SQL нуждается в пояснении. В SQL UNIQUE – это оператор, который возвращает TRUE тогда и только тогда, когда все строки в таблице-аргументе различаются. Таким образом, вызов UNIQUE в ограничении возвращает TRUE тогда и только тогда, когда в таблице S нет двух строк с одинаковым номером поставщика. Я еще вернусь к этому оператору в главе 10.

Для тех, кому интересно, приведу SQL-формулировку ограничения CX3, которая больше напоминает формулировку на **Tutorial D**:

```
CREATE ASSERTION CX3 CHECK
 ( ( SELECT COUNT ( ALL SNO ) FROM S ) =
   ( SELECT COUNT ( DISTINCT SNO ) FROM S ) ) ;
```

Пример 4:

```
CONSTRAINT CX4                  |   CREATE ASSERTION CX4 CHECK
COUNT ( S { SNO } ) =           |   ( NOT EXISTS ( SELECT *
COUNT ( S { SNO , CITY } ) ) ;  |                  FROM   S AS SX
                                |   WHERE EXISTS ( SELECT *
                                |                  FROM   S AS SY
                                |   WHERE SX.SNO = SY.SNO
                                |   AND   SX.CITY <> SY.CITY ) ) ) ;
```

Ограничение CX4 говорит: если в двух кортежах переменной-отношения S совпадают номера поставщиков, то должны совпадать и города. Иначе говоря, между {SNO} и {CITY} существует *функциональная зависимость*, которую чаще выражают так:

```
{ SNO } → { CITY }
```

Приведу определение.

> **Определение**: Пусть A и B – подмножества заголовка переменной-отношения R. Говорят, что R удовлетворяет *функциональной зависимости* (ФЗ) $A \rightarrow B$, если для любого отношения, являющегося допустимым значением R, из того, что в двух кортежах значения A одинаковы, следует, что и значения B в них тоже одинаковы.

Обозначение $A \rightarrow B$ читается как «B функционально зависит от A», или «A функционально определяет B», или просто «A стрелка B». Однако, как показывает предыдущий пример, по существу функциональная зависимость – это просто частный случай ограничения целостности (хотя оно и не является ограничением «кортежа», или «строки»).

В главе 5 отмечалось, что переменная-отношение S удовлетворяет этой конкретной ФЗ просто потому, что {SNO} является для нее ключом. Поэтому, если явно указано, что {SNO} – ключ, то записывать еще и эту ФЗ явно нет необходимости. Однако не все ФЗ – следствия наличия ключа. Пусть, например, требуется, чтобы любые два поставщика, находящиеся в одном городе, имели один и тот же статус (отметим, что для наших тестовых данных это ограничение не удовлетворяется). Очевидно, что это гипотетическое ограничение – ФЗ:

```
{ CITY } → { STATUS }
```

Поэтому его можно записать в том же виде, что ограничение CX4 (упражнение для читателя).

Возможно, вам кажется, что для формулировки ФЗ было бы удобно иметь сокращенную нотацию, как для ключей. Лично я так не думаю, поскольку хотя в общем случае не все ФЗ – следствия наличия ключей, в правильно спроектированной базе данных они должны быть таковыми. Иными словами, раз ФЗ трудно сформулировать для плохо спроектированной базы данных, значит, – что? – проектируйте базу правильно!

Примечание

В предыдущем абзаце под «правильно спроектированной» я имею в виду *надлежащим образом нормализованную* базу данных. Собственно нормализация обсуждается в приложении В, но я все же отмечу, что реляционные (или SQL) предложения и выражения работают независимо от того, нормализованы или нет переменные-отношения (или таблицы). Однако к этому следует добавить, что предложения и выражения часто проще сформулировать (и, вопреки распространенному мнению, они и выполняться будут быстрее), если таблицы полностью нормализованы. Впрочем, нормализация как таковая – это вопрос проектирования баз данных, а не реляционной модели или SQL.

Пример 5:

```
CONSTRAINT CX5 IS_EMPTY          |   CREATE ASSERTION CX5 CHECK
   ( ( S JOIN SP )               |   ( NOT EXISTS
     WHERE STATUS < 20           |   ( SELECT *
     AND   PNO = 'P6' ) ;        |     FROM   S NATURAL JOIN SP
                                 |     WHERE  STATUS < 20
                                 |     AND    PNO = 'P6' ) ) ;
```

Ограничение CX5 говорит: никакой поставщик со статусом меньше 20 не может поставлять деталь P6. Отметим, что в этом ограничении участвуют две взаимосвязанные переменные-отношения: S и SP. В общем случае в ограничении базы данных может участвовать сколько угодно переменных-отношений. *Терминологическое замечание*: ограничение, в котором участвует одна переменная-отношение, неформально называют ограничением переменной-отношения (или, чтобы подчеркнуть суть дела, ограничением с одной переменной-отношением). Ограничение, в котором участвуют две или более переменных-отношений, называют ограничением с несколькими переменными-отношениями. (Таким образом, CX3 и CX4 – ограничения с одной переменной-отношением, а CX5 – ограничение с несколькими переменными-отношениями. CX1 и CX2 – также ограничения с одной переменной-отношением.) Впрочем, все эти термины вряд ли стоит употреблять по причинам, о которых мы будем говорить в следующей главе.

Пример 6:

```
CONSTRAINT CX6                   |   CREATE ASSERTION CX6 CHECK
   SP { SNO } ⊆ S { SNO } ;      |   ( NOT EXISTS
                                 |     ( SELECT SNO
                                 |       FROM   SP
                                 |       EXCEPT
                                 |       SELECT SNO
                                 |       FROM   S ) ) ;
```

Ограничение CX6 говорит: любой номер поставщика, встречающийся в переменной-отношении SP, должен присутствовать и в переменной-отношении S. Как видите, в формулировке на языке **Tutorial D** уча-

ствует реляционное сравнение. В SQL же реляционные сравнения не поддерживаются, поэтому приходится прибегать к обходному маневру. Учитывая, что {SNO} – ключ (на самом деле, единственный ключ) переменной-отношения S, ясно, что CX6 – это, по существу, ограничение внешнего ключа от SP к S. Поэтому привычный синтаксис FOREIGN KEY можно считать сокращенной нотацией для ограничений вида CX6.

Ограничения базы данных в SQL

Любое ограничение, которое можно выразить с помощью предложения CONSTRAINT языка **Tutorial D**, можно также выразить в виде предложения CREATE ASSERTION в SQL. Приведенных в предыдущем разделе примеров CX1-CX6 должно быть достаточно для иллюстрации этого утверждения.[1] Однако, в отличие от **Tutorial D**, в SQL любое такое ограничение можно альтернативно записать в виде ограничения базовой таблицы, то есть включить в определение некоторой базовой таблицы. Вот, например, как выглядит SQL-версия ограничения CX5 из предыдущего раздела:

```
CREATE ASSERTION CX5 CHECK
     ( NOT EXISTS ( SELECT *
                    FROM   S NATURAL JOIN SP
                    WHERE  STATUS < 20
                    AND    PNO = 'P6' ) ) ;
```

Этот пример можно было бы также переформулировать несколько иначе, воспользовавшись спецификацией CONSTRAINT в предложении CREATE TABLE для базовой таблицы SP:

```
CREATE TABLE SP
  ( ... ,
    CONSTRAINT CX5 CHECK
      ( PNO <> 'P6' OR ( SELECT STATUS FROM S
                         WHERE  SNO = SP.SNO ) >= 20 ) ) ;
```

Отметим, однако, что логически эквивалентное ограничение можно было бы включить в определение базовой таблицы S или базовой таблицы P, да и вообще абсолютно любой таблицы базы данных (см. упражнение 8.17 в конце главы).

Такая альтернативная запись, безусловно, полезна для «ограничений строки» (таких, которые можно проверить, исследуя только одну строку), поскольку она проще, чем CREATE ASSERTION. Вот, например, как можно переписать ограничения CX1 и CX2 из предыдущего раздела в виде ограничений базовой таблицы S:

---

[1] С тем исключением (см. главу 2), что в SQL ограничения не должны содержать «потенциально недетерминированных выражений»; на практике нарушение этого правила может привести к серьезным проблемам. Дополнительную информацию см. в главе 12.

```
CREATE TABLE S
  ( ... ,
    CONSTRAINT CX1 CHECK ( STATUS >= 1 AND STATUS <= 100 ) ) ;

CREATE TABLE S
  ( ... ,
    CONSTRAINT CX2 CHECK ( STATUS = 20 OR CITY <> 'London' ) ) ;
```

Однако для ограничений, в которых участвует несколько базовых таблиц, предложение CREATE ASSERTION обычно лучше, поскольку оно позволяет избежать произвольного выбора таблицы, к которой присоединяется ограничение.

Примечание

Некоторые ограничения, например NOT NULL и ограничения ключа по нескольким столбцам, в SQL можно также записывать в виде «ограничений столбца». Так называются ограничения, которые задаются не просто в определении базовой таблицы, а как часть определения одного столбца этой таблицы. Для простоты я не буду прибегать к этой возможности, сделав исключение только для ограничений NOT NULL.

И в заключение этого раздела еще два замечания.

- Имейте в виду, что любое ограничение, заданное как часть предложения CREATE TABLE для базовой таблицы *T*, автоматически удовлетворяется, если *T* пуста, – даже если ограничение имеет вид «*T* не должна быть пуста»! (И даже такие ограничения: «*T* должна содержать -5 строк» или «1 = 0».) См. упражнения 8.15 и 8.16 в конце главы.
- (*Важно!*) Хотя в большинстве современных продуктов поддерживаются ограничения ключа и внешнего ключа, предложение CREATE ASSERTION не поддерживается вовсе, а ограничения базовых таблиц не могут быть сложнее, чем простые ограничения строки. (Если говорить формально, то ограничение базовой таблицы не может содержать подзапросов.) Поэтому на практике большинство ограничений приходится обеспечивать с помощью процедурного кода (возможно, в виде триггерных процедур), а писать такой код иногда совсем непросто.[1] Такое положение вещей нельзя расценить иначе, как серьезный дефект в современных продуктах, который нужно срочно устранить.

Транзакции

Несмотря на упомянутый в конце предыдущего раздела дефект, я все же буду до конца этой главы предполагать (как того требует реляционная модель), что произвольно сложные ограничения базы данных мож-

[1] В этой связи я хотел бы порекомендовать книгу Лекса де Хаана (Lex de Haan) и Тоона Коппелаарса (Toon Koppelaars) «Applied Mathematics for Database Professionals» (см. приложение D).

но сформулировать декларативно. Тогда возникает вопрос: в какой момент проверяются эти ограничения? Традиционно принято считать, что ограничение с одной переменной-отношением проверяется *немедленно* (то есть в момент обновления этой переменной-отношения), тогда как проверка ограничений с несколькими переменными-отношениями *откладывается* до конца транзакции («момента фиксации»). Однако я хочу доказать, что любые ограничения должны проверяться немедленно, а отложенная проверка, которая поддерживается в стандарте SQL и в некоторых продуктах, – логическая ошибка. Чтобы обосновать эту неортодоксальную точку зрения, я должен буду ненадолго отвлечься и поговорить о транзакциях.

Теория транзакций – сама по себе весьма обширная тема. Но она слабо связана с реляционной моделью (по крайней мере, связана не напрямую), поэтому я не хочу обсуждать ее подробно. Да и в любом случае вы, как человек, профессионально занимающийся базами данных, наверняка знакомы с основами транзакций. Стандартный учебник по этой теме (кстати, настоятельно рекомендую) – Джим Грей (Jim Gray) и Андреас Ройтер (Andreas Reuter) «Transaction Processing: Concepts and Techniques» (см. приложение D). Здесь я хочу лишь кратко остановиться на так называемых *ACID-свойствах* транзакций. Акроним ACID расшифровывается как «atomicity – consistency – isolation – durability» (атомарность – непротиворечивость – изолированность – долговечность).

- *Атомарность* означает, что транзакция выполняет «все или ничего».
- *Непротиворечивость* означает, что любая транзакция переводит базу данных из одного непротиворечивого состояния в другое непротиворечивое состояние, но в промежуточных точках непротиворечивость может нарушаться. (Состояние базы данных называется непротиворечивым, если в нем удовлетворяются все определенные в базе ограничения; в данном контексте непротиворечивость – просто другое название целостности.)
- *Изолированность* означает, что любые изменения, произведенные в одной транзакции, скрыты от всех остальных транзакций до тех пор, пока транзакция не будет зафиксирована.
- *Долговечность* означает, что после того как транзакция зафиксирована, произведенные в ней изменения сохраняются в базе данных даже в случае последующего сбоя системы.

Одним из аргументов в поддержку транзакций всегда было то, что транзакция – это «единица целостности» (собственно, вся идея непротиворечивости к этому и сводится). Но я в этот аргумент не верю; выше я уже намекал, возможно, не слишком отчетливо, что, на мой взгляд, такой единицей должно быть предложение. Иными словами, я считаю, что ограничения базы данных должны удовлетворяться на *границах предложений*. В следующем разделе я приведу обоснование этой позиции.

Почему ограничения базы данных должны проверяться немедленно

У меня есть по меньшей мере пять причин считать, что ограничения базы данных должны удовлетворяться на границах предложений. Первая и самая основательная такова: из главы 5 мы знаем, что базу данных можно рассматривать как совокупность высказываний, предположительно истинных. И если допустить наличие противоречий в этой совокупности, то уже *никаких гарантий дать нельзя*. Ниже в разделе «Ограничения и предикаты» я покажу, что мы не можем доверять ни одному ответу, полученному от противоречивой базы данных! Хотя из свойства изолированности следует, что не более, чем одна транзакция в некий момент может видеть некоторое конкретное противоречие, факт остается фактом: эта транзакция видит базу в противоречивом состоянии и, стало быть, может давать неверные ответы.

Думаю, что и одного этого аргумента было бы достаточно, но хотя бы для полноты приведу и остальные четыре. Итак, вторая причина – я не согласен с тем, что любое противоречие видно только одной транзакции; иными словами, я не очень-то верю в свойство изолированности. Отчасти беда в том, что слово *изолированность* в мире транзакций означает не то, что в обычном языке, в частности, оно вовсе не означает, что транзакции не могут взаимодействовать между собой. Ибо, если транзакция *TX1* порождает некий результат в базе данных или еще где-то, который затем считывается транзакцией *TX2*, то транзакции *TX1* и *TX2* уже нельзя считать по-настоящему изолированными (и это замечание сохраняет силу вне зависимости от того, выполняются транзакции параллельно или последовательно). Таким образом, если (а) *TX1* видит противоречивое состояние базы данных и потому порождает неверный результат и (б) этот результат впоследствии становится виден *TX2*, то (в) противоречие, увиденное *TX1*, распространяется и на *TX2*. Иными словами, невозможно гарантировать, что противоречие, если они вообще допустимы, будет видно только одной транзакции.

Третья причина – мы, безусловно, не хотим, чтобы любая программа (или иная «программная единица») была вынуждена учитывать возможность противоречивого состояния базы данных. Если некоторый код, предполагающий, что база данных непротиворечива, нельзя безопасно использовать в условиях отложенной проверки ограничений, то мы имеем существенное нарушение принципа ортогональности. Иначе говоря, я хочу иметь возможность писать код, не зависящий от того, будет ли он выполняться как самостоятельная транзакция или как часть транзакции. (Фактически, мне нужна поддержка вложенных транзакций, но эту тему мы отложим до другого раза.)

Четвертая причина – *принцип взаимозаменяемости* (базовых переменных-отношений и представлений, см. следующую главу) подразумевает, что одно и то же ограничение может быть ограничением с одной

переменной-отношением при одной структуре базы данных и ограничением с несколькими переменными-отношениями – при другой. Пусть имеется два представления с такими определениями на **Tutorial D** (LS = поставщики в Лондоне, NLS = поставщики не в Лондоне):

```
VAR LS  VIRTUAL ( S WHERE CITY = 'London' ) ;

VAR NLS VIRTUAL ( S WHERE CITY ≠ 'London' ) ;
```

Эти представления удовлетворяют ограничению – ни один номер поставщика не встречается в обоих сразу. Однако формулировать это ограничение явно нет необходимости, поскольку оно вытекает из ограничения с одной переменной-отношением, согласно которому {SNO} – ключ переменной-отношения S (а также из того факта, что каждый поставщик находится ровно в одном городе, неявно присутствующем в структуре переменной-отношения S). А теперь предположим, что мы превратили LS и NLS в базовые переменные-отношения и определили их объединение, назвав его S. Тогда ограничение пришлось бы задавать явно:

```
CONSTRAINT CX7 IS_EMPTY       |   CREATE ASSERTION CX7 CHECK
   ( LS { SNO } JOIN          |     ( NOT EXISTS
           NLS { SNO } ) ;    |     ( SELECT *
                              |       FROM   LS , NLS
                              |       WHERE  LS.SNO = NLS.SNO ) ) ;
```

Итак, ограничение, которое раньше содержало одну переменную-отношение, теперь содержит несколько. Значит, если мы согласились, что ограничения с одной переменной-отношением следует проверять немедленно, то должны согласиться и с тем, что это справедливо для ограничений с несколькими переменными-отношениями (поскольку никакого существенного логического различия между ними нет).

Пятая, и последняя, причина связана с техникой так называемой *семантической* оптимизации (она касается трансформации выражений, но я сознательно не стал обсуждать этот вопрос в главе 6). Рассмотрим, к примеру, выражение (SP JOIN S){PNO}. Здесь соединение основано на соответствии между внешним ключом в ссылающейся переменной-отношении SP и соответствующим ключом в переменной-отношении S, на которую ведет ссылка. Следовательно, каждый кортеж SP соединяется с некоторым кортежем S, и, значит, каждый кортеж SP привносит номер детали в проекцию, которая и является окончательным результатом. А раз так, то соединение вообще излишне! – выражение можно упростить до SP{PNO}. Следует отметить, что такая трансформация допустима лишь благодаря семантическим особенностям ситуации; в общем случае каждый операнд соединения может содержать кортежи, которым нет соответствия в другом операнде и которые, стало быть, не дают вклада в результат, поэтому подобная трансформация невозможна. Однако в нашем случае каждому кортежу SP есть соответствие в S из-за ограничения целостности – ограничения внешнего ключа, если быть точным, – которое говорит, что у каждой поставки должен быть

поставщик, поэтому трансформация все-таки возможна. Трансформации, допустимые лишь благодаря наличию некоторого ограничения целостности, называются семантическими, а соответствующая оптимизация – семантической оптимизацией.

В принципе для семантической оптимизации можно использовать любое ограничение, а не только ограничение внешнего ключа, как в этом примере.[1] Предположим, например, что на базу данных о поставщиках и деталях наложено следующее ограничение: «все красные детали должны находиться на складе в Лондоне», и рассмотрим такой запрос:

> *Получить поставщиков, которые поставляют только красные детали и находятся в том же городе, где находится хотя бы одна из поставляемых ими деталей.*

Это довольно сложный запрос, но благодаря ограничению целостности он может быть трансформирован – оптимизатором, конечно, а не пользователем – в гораздо более простой:

> *Получить поставщиков в Лондоне, которые поставляют только красные детали.*

Можно с уверенностью утверждать, что производительность повысится на несколько порядков. Поэтому, хотя сейчас лишь немногие продукты применяют семантическую оптимизацию (насколько мне известно), я ожидаю, что в будущем ситуация изменится, так как выигрыш очень значителен.

Возвращаясь к основной теме обсуждения, я хочу заметить, что для того чтобы ограничение было пригодно для семантической оптимизации, оно должно удовлетворяться в любой момент времени (точнее, на границах предложений), а не только на границах транзакций. Как мы видели, семантическая оптимизация подразумевает использование ограничений, чтобы упростить запросы и, как следствие, повысить производительность. Ясно, что если ограничение в какой-то момент нарушается, то любое достигнутое на его основе упрощение в этот момент некорректно и результаты упрощенного запроса тоже в этот момент будут неверны (в общем случае). *Примечание*: Можно было бы занять более умеренную позицию, согласившись с тем, что «отложенные ограничения» (то есть те, проверка которых откладывается) нельзя использовать для семантической оптимизации, но, думается, что так мы просто создадим себе лишние проблемы.

Подведем итог. Ограничение базы данных должно удовлетворяться (то есть при имеющихся в базе данных значениях его вычисление должно возвращать TRUE) *на границе предложений* (или, если говорить совсем

---

[1] Однако ограничение должно формулироваться декларативно; очевидно, что никакая семантическая оптимизация не в состояния «понять» и использовать ограничения, выраженные процедурно (и это еще одна веская причина требовать поддержки декларативных ограничений).

уж неформально, в местах, где встречаются точки с запятой). Иными словами, оно должно проверяться в конце каждого предложения, которое теоретически могло бы привести к нарушению. Если проверка завершается неудачно, то воздействие предложения на базу отменяется и возбуждается исключение.

Но разве можно не откладывать проверку некоторых ограничений?

Несмотря на все приведенные в предыдущем разделе аргументы, обычно считается, что уж проверку ограничений с несколькими переменными-отношениями необходимо откладывать до момента фиксации транзакции. Предположим, к примеру, что на базу данных о поставщиках и деталях наложено следующее ограничение:

```
CONSTRAINT CX8
    COUNT ( ( S WHERE SNO = 'S1' ) { CITY }
              UNION
            ( P WHERE PNO = 'P1' ) { CITY } ) < 2 ;
```

Это ограничение говорит, что поставщик S1 и деталь P1 не должны находиться в разных городах. Поясняю: если переменные-отношения S и P содержат кортежи для поставщика S1 и детали P1 соответственно, то значение атрибута CITY в этих кортежах должно быть одинаково (если это не так, то вызов COUNT вернул бы значение 2); однако если переменная-отношение S вообще не содержит кортежа для S1 или переменная-отношение P1 не содержит кортежа для P1, то это считается допустимым (в этих случаях вызов COUNT вернет 1 или 0). При таком ограничении и наших обычных тестовых данных ни одно из SQL-предложений UPDATE ниже не пройдет немедленную проверку:

```
UPDATE S SET CITY = 'Paris' WHERE SNO = 'S1' ;

UPDATE P SET CITY = 'Paris' WHERE PNO = 'P1' ;
```

Я специально записал эти предложения на SQL, потому что в **Tutorial D** проверка производится немедленно, так что традиционное решение проблемы для **Tutorial D** не годится. А что это за традиционное решение? *Ответ*: Мы откладываем проверку ограничения до момента фиксации[1] и помещаем оба предложения UPDATE в одну транзакцию:

---

[1] Если вас интересует, как реализуется откладывание, то объясню, что в общем случае (есть кое-какие исключения, которые нас сейчас не волнуют) любое ограничение в SQL определяется как (а) DEFERRABLE (откладываемое) или NOT DEFERRABLE (не откладываемое) и (б) INITIALLY DEFERRED (режим отложенной проверки в начале транзакции) или INITIALLY IMMEDIATE (режим немедленной проверки в начале транзакции). Далее, на этапе выполнения предложение SET CONSTRAINTS <*constraint name commalist*> <*option*>, где <*option*> – DEFERRED или IMMEDIATE,

```
START TRANSACTION ;
   UPDATE S SET CITY = 'Paris' WHERE SNO = 'S1' ;
   UPDATE P SET CITY = 'Paris' WHERE PNO = 'P1' ;
COMMIT ;
```

При таком традиционном решении ограничение проверяется в конце транзакции, а между двумя предложениями UPDATE база данных оказывается в противоречивом состоянии. В частности, если бы в этой транзакции между двумя UPDATE был задан вопрос «Находятся ли поставщик S1 и деталь P1 в разных городах?» (в предположении, что кортежи, или строки, для S1 и P1 существуют), то был бы получен *утвердительный* ответ.

Множественное присваивание

У этой задачи есть более правильное решение – поддержать множественную форму присваивания, которая позволяет выполнять сразу несколько присваиваний «одновременно». Например (мы возвращаемся к языку **Tutorial D**):

```
UPDATE S WHERE SNO = 'S1' : { CITY := 'Paris' } ,
UPDATE P WHERE PNO = 'P1' : { CITY := 'Paris' } ;
```

Пояснение: Во-первых, обратите внимание на запятую в качестве разделителя, она означает, что оба UPDATE являются частями одного предложения. Во-вторых, UPDATE на самом деле, как мы знаем, представляет собой присваивание, поэтому такой «двойной UPDATE» – это просто сокращенная запись следующей общей конструкции:

```
S := ... , P := ... ;
```

При таком двойном присваивании одно значение присваивается переменной-отношению S, а другое – переменной-отношению P, причем в рамках единой операции. В общем случае семантика множественного присваивания такова:

- Сначала вычисляются все выражения в правых частях
- Затем отдельные присваивания (переменным в левых частях) выполняются в указанной последовательности

(На самом деле, это определение нуждается в небольшом уточнении в случае, когда производится два или более присваиваний одной и той же переменной, но детали выходят за рамки этой книги.) Отметим, что поскольку все выражения в правых частях вычисляются до выполнения присваивания, ни одна из отдельных операций присваивания не зависит от результата всех остальных (поэтому фраза «в указанной по-

---

устанавливает режим заданных ограничений. Предложение COMMIT переводит все ограничения в режим немедленной проверки; если какая-то проверка завершится неудачно, то COMMIT тоже завершается с ошибкой, и вся транзакция откатывается.

следовательности» – просто фигура речи, в действительности последовательность не играет роли). Так как множественное присваивание считается единой операцией, проверка целостности «в середине» предложения не производится, – собственно, именно этот факт и является основным доводом в пользу такой операции. Следовательно, в нашем примере транзакция не может увидеть противоречивое состояние базы данных между двумя UPDATE, так как сама фраза «между двумя UPDATE» лишена смысла. Отметим еще, что теперь вообще не возникает необходимости в отложенной проверке.

Отступление

Быть может, стоит напомнить, что в реляционной модели все предложения *семантически атомарны*. Фактически, большинство предложений атомарны и синтаксически. Множественное присваивание является исключением, так как оно семантически атомарно, а синтаксически – нет.

А что можно сказать о множественном присваивании в SQL? В SQL эта операция в какой-то мере поддерживается, на самом деле поддержка существует уже много лет. Во-первых, в таких ссылочных действиях, как CASCADE, неявно подразумевается, что одно предложение DELETE или UPDATE может привести к «одновременному» обновлению нескольких базовых таблиц в рамках единственной операции. Во-вторых, возможность обновлять некоторые представления с соединением, если она поддерживается, подразумевает то же самое. В-третьих, операции FETCH INTO и SELECT INTO тоже в каком-то смысле можно считать множественным присваиванием. В-четвертых, SQL явно поддерживает множественное присваивание в предложении SET (это как раз то, что называется присваиванием строк, см. главы 2 и 3). И так далее (этот список неполон). Однако в настоящее время SQL не поддерживает явное «одновременное» присваивание нескольким разным *таблицам*, а именно эта операция иллюстрируется вышеприведенным примером и именно ее необходимо поддержать, чтобы уйти от отложенных проверок.[1]

И последнее замечание. Вы должны понимать, что поддержка множественного присваивания вовсе не означает, что можно отказаться от поддержки транзакций. Транзакции все равно необходимы – хотя бы для восстановления данных и обеспечения конкурентного доступа. Я лишь хочу сказать, что транзакции не являются «единицами целостности», как принято считать.

Рекомендация: Текущее состояние дел в SQL-системах заставляет откладывать некоторые проверки ограничений, хотя логически проверка должна производиться немедленно. В таких случаях вы должны сделать все необходимое (на практике это, вероятно, означает завершение

[1] Впрочем, мне говорили, что эта функциональность появится в будущей версии стандарта.

транзакции), чтобы выполнить проверку до того, как начнется операция, результат которой может зависеть от проверяемого ограничения.

Ограничения и предикаты

В главе 5 мы говорили о том, что предикат переменной-отношения – это ее подразумеваемая интерпретация (неформально говоря, смысл). Например, предикат переменной-отношения S звучит так:

Поставщик SNO связан контрактом, называется SNAME, имеет статус STATUS и находится в городе CITY.

Поэтому в идеальном мире предикат играл бы роль «критерия приемлемости обновления» переменной-отношения S, то есть определял бы, допустима ли данная операция INSERT, DELETE или UPDATE для данной переменной-отношения. Но эта цель недостижима.

- Во-первых, система не знает, что такое «поставщик связан контрактом» или «находится там-то»; это лишь вопрос интерпретации. Например, если в одном кортеже встретились поставщик с номером S1 и город Лондон, то лишь пользователь способен интерпретировать этот факт, сказав, что поставщик S1 находится в Лондоне,[1] а система ничего подобного сделать не может.
- Во-вторых, даже если бы система знала, что такое «поставщик связан контрактом» или «находится там-то», она все равно не может априори знать, правду ли говорит пользователь! Если пользователь сообщает системе (посредством некоторой операции обновления), что существует поставщик S6 по имени Лопес со статусом 30, находящийся в Мадриде, то у системы нет способа удостовериться в истинности этого утверждения. Система в состоянии лишь проверить, что он не нарушает никакого ограничения целостности. И, если это так, то система примет утверждение пользователя и, *начиная с этого момента, будет считать его истинным* (до того момента, как пользователь в другой операции обновления не скажет, что оно больше истинным не является).

Таким образом, прагматический «критерий приемлемости обновления», в отличие от идеального, – это не предикат, а соответствующее множество ограничений, которое можно рассматривать как аппроксимацию предиката с точки зрения системы. Или, по-другому:

Система гарантирует не истинность, а непротиворечивость.

---

[1] Или что поставщик когда-то находился в Лондоне, или что поставщик имеет офис в Лондоне, или что поставщик не имеет офиса в Лондоне, или любая другая из бесконечного множества возможных интерпретаций (которому, естественно, соответствует бесконечное множество возможных предикатов).

Как это ни печально, истинность и непротиворечивость – не одно и то же. Точнее, если база данных содержит только истинные высказывания, то она непротиворечива, но обратное может быть неверно. Если база данных противоречива, то она содержит хотя бы одно ложное утверждение, но обратное опять-таки неверно. Или, другими словами, *правильность* влечет за собой *непротиворечивость* (но не наоборот), а *противоречивость* влечет за собой *неправильность* (но не наоборот). Говоря, что база данных правильна, мы лишь говорим, что она верно отражает положение вещей в реальном мире, – не более и не менее.

Теперь позвольте мне сформулировать эти понятия чуть более формально. Пусть R – базовая переменная-отношение, и пусть $C1$, $C2$, .., Cm ($m \geq 0$) – ограничения базы данных с одной или несколькими переменными-отношениями, в которых упоминается R. Предположим для простоты, что каждое Ci – просто булево выражение (то есть не будем обращать внимания на имена ограничений). Тогда булево выражение

```
( C1 ) AND ( C2 ) AND ... AND ( Cm ) AND TRUE
```

называется *полным ограничением переменной-отношения R*. Кстати, обратите внимание на последний член «AND TRUE»; он нужен для того, чтобы в том маловероятном[1] случае, когда для переменной-отношения не определено никаких ограничений, по умолчанию возвращалось значение TRUE.

Теперь пусть RC – полное ограничение для переменной-отношения R. Понятно, что R никогда не должна принимать значение, при котором RC становится равным FALSE. Такое положение вещей служит обоснованием для утверждения, которое я люблю называть **золотым правилом** (его первой версией):

> *Никакая операция обновления не должна приводить к состоянию, в котором ограничение хотя бы одной переменной-отношения принимает значение FALSE.*

Пусть теперь DB – база данных, и пусть $R1$, $R2$, ..., Rn ($n \geq 0$) – все переменные-отношения в DB. Пусть для этих переменных-отношений заданы полные ограничения $RC1$, $RC2$, ..., RCn соответственно. Тогда булево выражение

```
( RC1 ) AND ( RC2 ) AND ... AND ( RCn ) AND TRUE
```

называется *полным ограничением базы данных DB*. И вот соответственно измененная – и окончательная – формулировка **золотого правила**:

> *Никакая операция обновления не должна приводить к состоянию, в котором хотя бы одно ограничение базы данных принимает значение FALSE.*

---

[1] Именно «маловероятном»; предполагается, что любая переменная-отношение имеет, по крайней мере, ограничение ключа.

Особо отмечу, что в соответствии с моим мнением о том, что все проверки целостности должны производиться немедленно, это правило сформулировано в терминах операций обновления, а не транзакций.

Теперь можно вернуться к некоторым незаконченным делам. Я говорил, что мы никогда не можем доверять ответам, полученным от противоречивой базы данных. И вот доказательство. Как мы знаем, базу данных можно рассматривать как совокупность высказываний. Предположим, что эта совокупность противоречива, то есть из нее можно сделать вывод, что существует некоторое высказывание *p*, для которого истинно как *p*, так и NOT *p*. Пусть теперь *q* – произвольное высказывание. Тогда:

- Из истинности *p* можно заключить, что *p* OR *q* истинно.
- Из истинности *p* OR *q* и истинности NOT *p* можно заключить, что *q* истинно.

Но *q* было произвольным! Следовательно, в противоречивой системе можно доказать, что вообще любое высказывание (даже такое очевидно ложное, как 1 = 0) «истинно». *Примечание*: Если это вас еще не убедило, отсылаю к дальнейшему обсуждению в главе 10.

Разное

Есть еще несколько моментов, касающихся целостности, которые трудно отнести к какому-то из разделов выше, а обсудить все-таки надо.

Во-первых, всякое ограничение, будучи по существу булевым выражением, которое должно принимать значение TRUE, фактически является высказыванием (в предыдущем разделе я уже намекал на этот факт, но не сформулировал его явно). Например, приведем еще раз ограничение CX1 из раздела «Ограничения базы данных»:

```
CONSTRAINT CX1 IS_EMPTY ( S WHERE STATUS < 1 OR STATUS > 100 ) ;
```

Здесь переменная-отношение «S» – это то, что в формальной логике называется *обозначением* (designator); в момент проверки ограничения она обозначает конкретное значение, а именно значение рассматриваемой переменной-отношения в этот момент. По определению, это значение является отношением (назовем его *s*), поэтому ограничение принимает вид:

```
CONSTRAINT CX1 IS_EMPTY ( s WHERE STATUS < 1 OR STATUS > 100 ) ;
```

Очевидно, что булево выражение здесь (которое в действительности и является ограничением, а слова «CONSTRAINT CX1» – не более чем украшение) либо истинно, либо ложно, без дополнительных условий, а это и есть определение высказывания (см. главу 5).

Во-вторых, предположим, что переменная-отношение S уже содержит кортеж, который нарушает ограничение CX1 в момент выполнения

предложения CONSTRAINT; тогда это выполнение должно завершиться неудачно. Более общо, всякий раз, как мы пытаемся определить новое ограничение базы данных, система должна проверить, удовлетворяется ли оно в данный момент; если нет, ограничение следует отвергнуть, иначе принять и в дальнейшем следить за его истинностью.

В-третьих, предполагается, что реляционные базы данных удовлетворяют ограничению ссылочной целостности, которое говорит, что не должно быть внешних ключей, не имеющих соответствия. В главе 1 я назвал это правило «обобщенным ограничением целостности». Но должно быть ясно, что это совсем не то, что ограничения, рассматриваемые в данной главе. В действительности это в некотором смысле *мета*ограничение, оно говорит, что любая конкретная база данных должна удовлетворять тем конкретным ограничениям ссылочной целостности, которые применимы к этой базе. Так, в случае базы данных о поставщиках и деталях оно говорит о том, что должны удовлетворяться ссылочные ограничения от SP к S и P, поскольку если это не так, то будет нарушено метаограничение ссылочной целостности. Аналогично в случае базы данных об отделах и служащих из главы 1 должно удовлетворяться ссылочное ограничение от EMP к DEPT, поскольку в противном случае снова было бы нарушено метаограничение ссылочной целостности.

И еще один момент, о котором у меня не было случая упомянуть раньше, – это возможность поддержки ограничений *перехода*. Так называются ограничения на допустимые переходы переменных (в частности, переменных-отношений) от одного значения к другому (ограничения же, которые не являются ограничениями перехода, иногда называют ограничениями *состояния*). Например, семейное положение человека может измениться с «никогда не состоял в браке» на «женат/замужем», но не наоборот. Вот пример такого ограничения («статус поставщика не может уменьшаться»):

```
CONSTRAINT CX9 IS_EMPTY
   ( ( ( S′ { SNO , STATUS } RENAME ( STATUS AS OLD ) )
         JOIN
       ( S  { SNO , STATUS } RENAME ( STATUS AS NEW ) ) )
     WHERE OLD > NEW ) ;
```

Объяснение: Я следую соглашению о том, что имя переменной-отношения со штрихом (например, S′) относится к указанной переменной-отношению в состоянии до обновления. Таким образом, ограничение CX9 гласит: «если соединить старое значение S с новым и ограничить результат только теми кортежами, для которых старый статус меньше нового, то должно получиться пустое множество». (Так как соединение производится по атрибуту SNO, то любой принадлежащий соединению кортеж, для которого старый статус больше нового, представлял бы поставщика с уменьшившимся статусом.)

В настоящее время ограничения перехода не поддерживаются ни в **Tutorial D**, ни в SQL (разве что процедурно).

И наконец, я надеюсь, что, прочитав эту главу, вы согласитесь, что ограничения необычайно важны, но тем не менее очень слабо поддержаны в современных продуктах. Создается даже впечатление, что их недооценивают, а то и вовсе не понимают. На практике упор всегда делают на *производительность, производительность и еще раз производительность*; прочие цели, например простота использования, независимость от данных и, в том числе, целостность, похоже приносят в жертву этому всепоглощающему стремлению или в лучшем случае отодвигают на задворки.[1]

Я бы не хотел быть понятым неправильно. Разумеется, производительность тоже важна. С функциональной точки зрения, система, не обеспечивающая хотя бы адекватную производительность, вообще не может считаться системой (по крайней мере, практически пригодной). Но какой смысл в быстро работающей системе, если нельзя доверять правильности получаемой от нее информации? Откровенно говоря, мне все равно, насколько быстро система работает, если я не уверен в правильности ответов на свои запросы.

Упражнения

Упражнение 8.1. Определите термины *ограничение типа* и *ограничение базы данных*. В какой момент такие ограничения проверяются? Что произойдет, если проверка завершится неудачно?

Упражнение 8.2. Сформулируйте **золотое правило**. Верно ли, что это правило может быть нарушено тогда и только тогда, когда нарушено какое-то ограничение с одной переменной-отношением?

Упражнение 8.3. Что вы понимаете под терминами *утверждение* (assertion), *ограничение атрибута*, *ограничение базовой таблицы*, *ограничение столбца*, *ссылочное ограничение*, *ограничение переменной-отношения*, *ограничение строки*, *ограничение с одной переменной-отношением*, *ограничение состояния*, *полное ограничение базы данных*, *полное ограничение переменной-отношения*, *ограничение перехода*, *ограничение кортежа*? В какие из этих категорий попадают (а) ограничения ключа, (б) ограничения внешнего ключа?

Упражнение 8.4. В чем различие между допустимым и физическим представлением?

---

[1] Я не хочу сказать, что надлежащая поддержка проверки ограничений целостности влечет за собой снижение производительности; напротив, я думаю, что производительность должна повыситься. Я лишь имею в виду, что усилия разработчиков направлены, прежде всего, на улучшение производительности в ущерб другим аспектам, и целостности данных в частности.

Упражнение 8.5. В предположении, что тип QTY определен на языке **Tutorial D**, как в тексте главы, – что возвращают следующие выражения?

a. `THE_Q ( QTY ( 345 ) )`

b. `QTY ( THE_Q ( QTY ) )`

Упражнение 8.6. Объясните максимально полно: (а) что такое селектор; (б) что такое оператор THE_. *Примечание*: Это упражнение повторяет уже встречавшиеся в предыдущих главах, но теперь вы в состоянии дать более детальный ответ.

Упражнение 8.7. Предположим, что допустимы только такие города: London, Paris, Rome, Athens, Oslo, Stockholm, Madrid и Amsterdam. Определите на **Tutorial D** тип CITY, который удовлетворяет этому ограничению.

Упражнение 8.8. Продолжая предыдущее упражнение, покажите, как в SQL наложить соответствующее ограничение на столбцы CITY в базовых таблицах S и P. Предложите по меньшей мере два решения. Сравните их между собой и со своим ответом на предыдущее упражнение.

Упражнение 8.9. Определите номера поставщиков в виде пользовательского типа на **Tutorial D**. Считайте допустимыми только номера поставщиков, которые можно представить строкой, содержащей не менее двух символов, из которых первый равен «S», а остальные обозначают десятичное число от 1 до 9999. Сформулируйте предположения о том, какие операторы необходимы, чтобы выразить определение такого типа.

Упражнение 8.10. Дайте на языке **Tutorial D** определение отрезка прямой, соединяющего две точки на евклидовой плоскости.

Упражнение 8.11. Можете ли вы придумать тип, для которого желательно иметь два допустимых представления? Имеет ли смысл включать ограничения типа в каждое из нескольких допустимых представлений одного и того же типа?

Упражнение 8.12. Можете ли вы придумать тип, для которого различные допустимые представления содержат разное количество компонентов?

Упражнение 8.13. Какие операции могли бы привести к нарушению ограничений CX1-CX9, приведенных в тексте главы?

Упражнение 8.14. Существует ли в языке **Tutorial D** какой-нибудь прямой аналог ограничений базовых таблиц в SQL?

Упражнение 8.15. Какая именно особенность SQL (дайте формальный ответ) упрощает формулировку ограничений базовых таблиц по сравнению с предложением CREATE ASSERTION? *Примечание*:

Уже изложенного материала недостаточно для ответа на этот вопрос. Тем не менее, подумайте над ним или обсудите в группе.

Упражнение 8.16. Продолжая тему предыдущего вопроса, скажу, что ограничение базовой таблицы автоматически считается в SQL выполненным, если таблица, к которой оно относится, пуста. Как вы думаете, почему принято такое решение (я хотел бы услышать формальную причину)? Аналогично ли поведение в **Tutorial D**?

Упражнение 8.17. В тексте главы я сформулировал ограничение CX5 в виде ограничения на базовую таблицу SP. Однако я отметил, что можно было бы записать его в виде ограничения на базовую таблицу S или на таблицу P, да и вообще на любую базовую таблицу. Приведите эти альтернативные формулировки.

Упражнение 8.18. У ограничения CX1 (к примеру) было свойство, для проверки которого для данного кортежа достаточно было исследовать только этот кортеж; у ограничения CX5 такого свойства не было. Какая формальная особенность отвечает за это различие? Каково прагматическое значение этого различия (если оно есть)?

Упражнение 8.19. Можете ли вы написать на **Tutorial D** ограничение базы данных, в точности эквивалентное спецификации KEY{SNO} для переменной-отношения S?

Упражнение 8.20. Приведите формулировку ограничения CX8 на SQL.

Упражнение 8.21. Напишите на **Tutorial D** и/или SQL ограничения для базы данных о поставщиках и деталях, которые бы выражали следующие требования:

a. Все красные детали должны весить меньше 50 фунтов.

b. Каждый поставщик, находящийся в Лондоне, должен поставлять деталь P2.

c. Никакие два поставщика не должны находиться в одном городе.

d. В любой момент времени в Афинах не должно быть более одного поставщика.

e. В Лондоне должен быть хотя бы один поставщик.

f. Хотя бы одна красная деталь должна весить меньше 50 фунтов.

g. Средний статус поставщика должен быть не меньше 10.

h. Ни в одной поставке количество деталей не должно более чем вдвое превышать среднее количество деталей по всем поставкам.

i. Поставщик с максимальным статусом не может находиться в одном городе с любым поставщиком, имеющим минимальный статус.

j. Любая деталь должна находиться в городе, где есть хотя бы один поставщик.

k. Любая деталь должна находиться в городе, где есть хотя бы один поставщик этой детали.

l. Поставщики из Лондона должны поставлять больше различных видов деталей, чем поставщики из Парижа.

m. Суммарное количество деталей, поставленных поставщиками из Лондона, должно быть больше, чем поставленных поставщиками из Парижа.

n. Ни для какой поставки общий вес (вес детали, умноженный на количество в поставке) не должен превышать 20 000 фунтов.

В каждом случае укажите, какие операции могли бы нарушить ограничение. *Примечание*: Не забывайте про отношения-образы, которые могут помочь при формулировании некоторых ограничений.

Упражнение 8.22. В тексте главы я определил полное ограничение базы данных как булево выражение вида:

```
( RC1 ) AND ( RC2 ) AND ... AND ( RCn ) AND TRUE
```

В чем смысл части «AND TRUE»?

Упражнение 8.23. В сноске в разделе «Ограничения и предикаты» я сказал, что если значения S1 и London встречаются в одном кортеже, то это может означать (в числе многих других возможных интерпретаций), что поставщик S1 не имеет офиса в Лондоне. На самом деле, такая интерпретация крайне маловероятна. Почему? *Подсказка*: Вспомните о *допущении замкнутости мира*.

Упражнение 8.24. Предположим, что для поставщиков и поставок задано правило «каскадного удаления». Напишите на языке **Tutorial D** предложение, которое удалит указанного поставщика и все его поставки за одну операцию (то есть не давая шанс нарушить ограничение ссылочной целостности).

Упражнение 8.25. Применяя синтаксис, намеченный для ограничений перехода в разделе «Разное», напишите ограничения перехода, которые выражали бы следующие требования:

a. Поставщики из Афин могут переезжать только в Лондон или Париж, а поставщики из Лондона могут переезжать только в Париж.

b. Общее поставленное количество деталей данного вида не может уменьшаться.

c. Общее количество деталей, поставленное данным поставщиком, не может быть уменьшено в одной операции обновления более чем на половину текущего значения. (Как вы понимаете уточнение «в одной операции обновления»? Почему оно важно? И важно ли оно?)

Упражнение 8.26. В чем различие между правильностью и непротиворечивостью?

Упражнение 8.27. Исследуйте любую доступную вам SQL-систему. Поддерживаются ли в ней какие-нибудь семантические оптимизации?

Упражнение 8.28. Как вы думаете, почему SQL не поддерживает ограничения типа? Каковы последствия такого положения вещей?

Упражнение 8.29. При обсуждении в этой главе типов вообще и ограничений типов в частности молчаливо подразумевалось, что все типы (а) скалярные и (б) определенные пользователем. В какой мере изложенные положения применимы к нескалярным и/или системным типам?

Упражнение 8.30. Встречаются ли в каких-нибудь найденных вами решениях упражнений на SQL «потенциально недетерминированные выражения»? Если да, то составляет ли это проблему? Если составляет, то что с этим можно поделать?

9

SQL и представления

Есть несколько интуитивно понятных способов описать, что такое представление. Все они корректны и могут оказаться полезны в определенных ситуациях.

- Представление – это виртуальная переменная-отношение, то есть оно «выглядит» как базовая переменная-отношение, но, в отличие от последней, не существует независимо от других переменных-отношений – фактически, оно и определено-то в терминах других переменных-отношений.
- Представление – это производная переменная-отношение, то есть оно явно произведено (и пользователям, по крайней мере некоторым, об этом известно) из других переменных-отношений. *Примечание*: На всякий случай поясню, что любая виртуальная переменная-отношение является производной, но существуют производные переменные-отношения, не являющиеся виртуальными. См. раздел «Представления и снимки» ниже в этой главе.
- Представление – это «окно», через которое можно смотреть на те переменные-отношения, из которых оно произведено; таким образом, операции над представлением на самом деле производятся над базовыми переменными-отношениями.
- Представление – это то, что некоторые авторы называют «готовым запросом» (или, более точно, именованным реляционным выражением).

Как обычно, я буду рассматривать эти понятия в терминах реляционной теории и SQL. Но по поводу SQL позвольте напомнить сказанное в главе 1: представление – это таблица! – или, как я предпочитаю говорить, переменная-отношение. В документации по SQL часто встречаются выражения типа «таблицы и представления», наводящие на мысль, будто таблицы и представления – вещи разные. Это не так, вся суть пред-

ставлений в том, что это таблицы. Поэтому не впадайте в распространенную ошибку и не считайте, что термин *таблица* относится исключительно к базовым таблицам. Тот, кто так думает, мыслит не реляционно и, вероятно, из-за этого будет допускать ошибки; собственно, ряд таких ошибок можно найти в самом языке SQL. Действительно, можно сказать, что даже названия операторов CREATE TABLE и CREATE VIEW – ошибка, по крайней мере психологическая, так как тем самым утверждается (а) мысль о том, что термин *таблица* относится только к базовым таблицам, и (б) мысль о том, что таблицы и представления – разные вещи. Берегитесь путаницы!

Последнее предварительное замечание: вопрос о том, следует ли «всегда» обращаться к базе данных через представления, обсуждается в разделе «Именование столбцов в SQL» главы 3 и в разделе «Зависимость от имен атрибутов» главы 6.

Представления – это переменные-отношения

Из перечисленных выше неформальных характеристик представления можно извлечь следующее предпочтительное определение, хотя все неформальные характеристики ему, в общем-то, эквивалентны.

> **Определение:** *Представление V* – это переменная-отношение, значением которого в момент времени *t* является результат вычисления некоторого реляционного выражения в момент *t*. Это выражение (*выражение, определяющее представление*) задается при определении *V* и должно содержать хотя бы одну ссылку на переменную-отношение.

Следующие примеры («поставщики в Лондоне» и «поставщики не в Лондоне») уже приводились в главе 8, но сейчас я дам также определения на SQL:

```
VAR LS VIRTUAL                    |  CREATE VIEW LS
  ( S WHERE CITY = 'London' ) ;   |    AS ( SELECT *
                                  |         FROM   S
                                  |         WHERE  CITY = 'London' )
                                  |         WITH CHECK OPTION ;

VAR NLS VIRTUAL                   |   CREATE VIEW NLS
 ( S WHERE CITY j 'London' ) ;    |   AS ( SELECT *
                                  |        FROM   S
                                  |        WHERE  CITY <> 'London' )
                                  |        WITH CHECK OPTION ;
```

Обратите внимание, что в обоих случаях речь идет о *представлениях с ограничением*: в любой момент времени значением представления является некоторое ограничение значения, которое базовая переменная-отношение S принимает в этот момент. Отмечу некоторые синтаксические тонкости.

- Скобки в этих примерах необязательны, но и не являются ошибкой. Я включил их только ради понятности;
- В SQL-предложении CREATE VIEW после имени представления может находиться заключенный в скобки список имен столбцов, разделенных запятыми, например:

```
CREATE VIEW SDS ( SNAME , DOUBLE_STATUS )
  AS ( SELECT DISTINCT SNAME , 2 * STATUS
       FROM   S ) ;
```

  Рекомендация: Не делайте так – следуйте рекомендациям из раздела «Именование столбцов в SQL» главы 3. Так, представление SDS можно с тем же успехом (и даже лучше) определить следующим образом:

```
CREATE VIEW SDS
  AS ( SELECT DISTINCT SNAME , 2 * STATUS AS DOUBLE_STATUS
       FROM   S ) ;
```

  Отмечу, в частности, что при таком подходе нужно лишь один, а не два раза сообщить системе о наличии столбца с именем SNAME.

- Предложение CREATE VIEW в SQL допускает также фразу WITH CHECK OPTION, если представление рассматривается как обновление. **Рекомендация**: По возможности старайтесь включать эту фразу. См. также раздел «Операции обновления» ниже.

Принцип взаимозаменяемости

Коль скоро представления – это переменные-отношения, практически все сказанное в предыдущих главах о переменных-отношениях в общем, относится в частности и к представлениям. Ниже мы подробно обсудим различные аспекты этого наблюдения. Но сначала необходимо разъяснить более фундаментальный момент.

Снова рассмотрим пример с поставщиками из Лондона и не из Лондона. Здесь S – базовая переменная-отношение, а LS и NLS – представления. *Но все могло бы быть и наоборот*, то есть ничто не мешает сделать LS и NLS базовыми переменными-отношениями, а S – представлением (для простоты показана только формулировка на **Tutorial D**):

```
VAR LS BASE RELATION
  { SNO CHAR , SNAME CHAR , STATUS INTEGER , CITY CHAR }
  KEY { SNO } ;

VAR NLS BASE RELATION
  { SNO CHAR , SNAME CHAR , STATUS INTEGER , CITY CHAR }
  KEY { SNO } ;

VAR S VIRTUAL ( LS D_UNION NLS ) ;
```

(Напомню, что D_UNION – «дизъюнктное объединение» – это вариант объединения, при котором требуется, чтобы у операндов не было общих кортежей.) *Примечание*: Чтобы гарантировать, что эта конструкция логически эквивалентна исходной, нам пришлось бы задать и проверять некоторые ограничения, – в частности, что в LS атрибут CITY в любом кортеже имеет значение London, а в NLS, напротив, кортежей с таким значением CITY нет, – но эти детали я здесь опущу. В разделе «Представления и ограничения» мы еще вернемся к этой теме.

Как бы то ни было, смысл этого примера в том, что в общем случае безразлично, какие переменные-отношения делать базовыми, а какие – виртуальными (по крайней мере, с формальной точки зрения). Мы могли бы спроектировать базу данных как минимум двумя разными способами, которые логически различны, но информационно эквивалентны. (Под *информационной эквивалентностью* я здесь понимаю, что в обоих случаях представлена одна и та же информация, то есть для любого запроса к одной базе найдется логически эквивалентный ему запрос к другой базе.) Отсюда естественно вытекает *принцип взаимозаменяемости*:

> **Определение:** Принцип взаимозаменяемости (базовых и виртуальных переменных-отношений) утверждает, что не должно быть произвольных, не обусловленных необходимостью различий между базовыми и виртуальными переменными-отношениями, то есть виртуальные переменные-отношения должны «выглядеть» неотличимо от базовых с точки зрения пользователя.

Вот некоторые следствия этого принципа.

- Я уже отмечал мимоходом, что представления, как и базовые переменные-отношения, могут быть субъектами ограничений целостности. (Обычно мы считаем, что ограничения целостности применяются только к базовым переменным-отношениям, но *принцип взаимозаменяемости* показывает, что это мнение ни на чем не основано.) См. раздел «Представления и ограничения» ниже.
- В частности, у представлений могут быть потенциальные ключи (поэтому мне, наверное, следовало включить спецификации каких-нибудь ключей в примеры представлений; **Tutorial D** допускает наличие таких спецификаций, а SQL – нет). Они могут иметь также внешние ключи, и, наоборот, внешние ключи могут ссылаться на представления. Опять же см. раздел «Представления и ограничения» ниже.
- Я не упомянул об этом в главе 1, но предполагается, что правило «целостности сущностей» относится исключительно к базовым переменным-отношениям, а не к представлениям, и потому нарушает *принцип взаимозаменяемости*. Конечно, я в принципе отвергаю это

правило, так как оно связано с null-значениями. (Я отвергаю его также потому, что оно касается только первичных ключей, а не потенциальных ключей вообще, но это просто попутное замечание.)

- Во многих SQL-системах и в стандарте SQL предлагается некоторый вид «идентификаторов строк». Если эта функция применима только к базовым таблицам, а не к представлениям, – а на практике это весьма вероятно, – то она нарушает *принцип взаимозаменяемости*. (Она может нарушать также и *принцип информации*. См. приложение А.) Идентификаторы строк как таковые не являются частью реляционной модели, но это еще не означает, что их не следует поддерживать. Однако хочу заметить, что если такие идентификаторы рассматриваются (а, к величайшему сожалению, и в стандарте SQL, и в большинстве основных SQL-систем так оно и есть) как некая разновидность идентификаторов объектов в объектно-ориентированном смысле, то их, безусловно, следует запретить! Идентификаторы объектов по сути своей являются указателями, а в реляционной модели (см. главу 2) указатели явно запрещены.
- Обсуждавшиеся в предыдущей главе различия между ограничениями с одной и несколькими переменными-отношениями скорее кажущиеся, чем реально существующие (как раз поэтому употребление такой терминологии не поощряется). Как показывает ранее приведенный пример, одно и то же ограничение может включать одну переменную-отношение при одной структуре базы данных и несколько – при другой.
- И пожалуй, самое главное – должна быть возможность обновлять представления, потому что, если это не так, то мы имеем очевидное нарушение *принципа взаимозаменяемости*. См. раздел «Операции обновления» ниже.

Константы-отношения

Возможно, вы обратили внимание на то, что в приведенном выше формальном определении требуется, чтобы выражение, определяющее представление, ссылалось хотя бы на одну переменную-отношение. Почему? Потому что в противном случае «виртуальная переменная-отношение» вообще не была бы переменной-отношением! Я хочу сказать, что она не была бы переменной и не допускала бы обновления. Например, следующее предложение CREATE VIEW является допустимым в SQL:

```
CREATE VIEW S_CONST ( SNO , SNAME , STATUS , CITY ) AS
             VALUES ( 'S1' , 'Smith' , 20 , 'London' ) ,
                    ( 'S2' , 'Jones' , 10 , 'Paris'  ) ,
                    ( 'S3' , 'Blake' , 30 , 'Paris'  ) ,
                    ( 'S4' , 'Clark' , 20 , 'London' ) ,
                    ( 'S5' , 'Adams' , 30 , 'Athens' ) ;
```

Но такое представление, разумеется, нельзя обновить. Иными словами, это и не переменная вовсе, что уж говорить о виртуальности; скорее, ее

можно назвать *именованной константой-отношением*. Позвольте пояснить.

- Прежде всего, я считаю термины *константа* и *значение* синонимами. Отмечу поэтому, что существует логическое различие между константой и литералом; литерал является не константой, а символом (иногда его называют «самоопределенным» символом), который обозначает константу.
- Строго говоря, существует также логическое различие между константой и именованной константой; константа – это значение, а именованная константа похожа на переменную с тем отличием, что ее значение нельзя изменять. Тем не менее, в дальнейшем я буду считать, что термин *константа* означает именованную константу.
- Естественно, константы могут иметь любой тип, но меня здесь интересуют только константы-отношения. В настоящее время язык **Tutorial D** не поддерживает константы-отношения, но, если бы поддерживал, то определение константы-отношения, наверное, выглядело бы примерно так:

```
CONST PERIODIC_TABLE INIT ( RELATION
   { TUPLE { ELEMENT 'Hydrogen' , SYMBOL 'H'  , ATOMICNO  1 } ,
     TUPLE { ELEMENT 'Helium'   , SYMBOL 'He' , ATOMICNO  2 } ,
       ..................................................
     TUPLE { ELEMENT 'Uranium'  , SYMBOL 'U'  , ATOMICNO 92 } } ) ;
```

Я полагаю, что было бы желательно обеспечить какую-то поддержку констант-отношений в описанном духе. На самом деле, в языке **Tutorial D** уже есть две системных константы-отношения: TABLE_DUM и TABLE_DEE, которые, как мы знаем, весьма важны. Но, если не считать этих двух, то ни **Tutorial D**, ни SQL пока не поддерживают констант-отношений. Правда, такую поддержку можно эмулировать с помощью существующего механизма представлений (мы это уже видели), но между константами и переменными существует гигантское логическое различие, поэтому я не думаю, что, притворившись, будто константы-отношения являются переменными-отношениями, мы както проясним ситуацию.

Представления и предикаты

Из *принципа взаимозаменяемости* следует, что с представлением (как и с базовой переменной-отношением) связан некий предикат, параметры которого взаимно однозначно соответствуют атрибутам переменной-отношения (то есть представления). Однако предикат, применимый к представлению *V*, является *производным* предикатом: он произведен из предикатов тех переменных-отношений, через которые определено *V*, в соответствии с семантикой реляционных операций, участвующих в выражении, определяющем представление. Да вы это, собствен-

но, уже знаете – в главе 6 я объяснял, что с каждым реляционным выражением ассоциирован предикат, и, разумеется, представлению соответствует именно тот предикат, который ассоциирован с определяющим его выражением. Рассмотрим снова представление LS («поставщики в Лондоне»), которое было определено в начале раздела «Представления – это переменные-отношения». Это представление является ограничением переменной-отношения S, следовательно, его предикат есть логическое AND предиката S и условия ограничения:

Поставщик SNO связан контрактом, называется SNAME, имеет статус STATUS и находится в городе CITY

AND

город CITY – это London.

Или в более естественной форме:

Поставщик SNO связан контрактом, называется SNAME, имеет статус STATUS и находится в Лондоне.

Отметим, однако, что эта более естественная форма затушевывает тот факт, что CITY – параметр. А ведь это действительно параметр, но соответствующий ему аргумент всегда равен константе 'London'. (Именно по этой причине на практике в представлении LS атрибут CITY, скорее всего, был бы удален из проекции.)

Аналогично предикат для представления NLS выглядит так:

Поставщик SNO связан контрактом, называется SNAME, имеет статус STATUS и находится в городе CITY, отличном от Лондона.

Операции выборки

Принцип взаимозаменяемости подразумевает, что пользователь может оперировать представлениями, как будто это базовые переменные-отношения, а СУБД должна уметь отображать эти операции на подходящие операции над «самыми» базовыми переменными-отношениями, в терминах которых определено представление. *Примечание*: я употребил слово «самыми», потому что, коль скоро представления во всем подобны базовым переменным-отношениям, то мы, в частности, можем определять на их основе новые представления, как в примере ниже:

```
CREATE VIEW LS_STATUS
  AS ( SELECT SNO , STATUS
       FROM   LS ) ;
```

В этом разделе мы для простоты будем говорить только об отображении операций чтения или «выборки» (напомню, что все операции реляционной алгебры в действительности являются операциями чтения). Процедура отображения операции чтения представления на операции над

базовыми переменными-отношениями довольно проста. Рассмотрим, к примеру, такой SQL-запрос к представлению LS:

```
SELECT LS.SNO
FROM   LS
WHERE  LS.STATUS > 10
```

Сначала СУБД заменяет ссылку на представление во фразе FROM выражением, определяющим это представление. В результате получаем:

```
SELECT LS.SNO
FROM ( SELECT S.*
       FROM   S
       WHERE  S.CITY = 'London' ) AS LS
WHERE  LS.STATUS > 10
```

Это выражение можно уже вычислить непосредственно. Однако – и с точки зрения производительности это, пожалуй, существенно – его можно упростить до:

```
SELECT S.SNO
FROM   S
WHERE  S.CITY = 'London'
AND    S.STATUS > 10
```

По всей вероятности, именно это выражение и будет реально вычислено.

Важно понимать, что описанная процедура работает лишь благодаря свойству реляционной замкнутости. В числе прочего замкнутость подразумевает, что всюду, где можно употребить какое-то имя (например, в запросе), можно подставить и более сложное выражение, при вычислении которого получается нечто подходящего типа.[1] Так, во фразе FROM может находиться имя SQL-таблицы, следовательно, может находиться и более общее табличное выражение SQL. Именно поэтому мы и смогли подставить выражение, определяющее представление LS, вместо самого имени LS.

По очевидным причинам описанная выше процедура реализации операций чтения над представлениями называется *процедурой подстановки*. Кстати, стоит отметить, что эта процедура не работала в ранних версиях SQL; если быть точным, в версиях до 1992 года. Причина в том, что в ранних версиях не в полной мере поддерживалось свойство замкнутости. А в результате безобидные, на первый взгляд, запросы к ничем, казалось бы, не примечательным таблицам (а на самом деле представлениям) завершались ошибкой, да еще так, что и объяснить, отку-

[1] Как и всюду в этой книге, я предпочел бы более формальный термин *объект* туманному *нечто*, но (как уже отмечалось в предыдущей главе) это слово в информатике слишком перегружено.

да взялась ошибка, было затруднительно. Вот простой пример. Рассмотрим такое определение представления:

```
CREATE VIEW V
  AS ( SELECT CITY , SUM ( STATUS ) AS ST
       FROM   S
       GROUP  BY CITY ) ;
```

И такой запрос:

```
SELECT CITY
FROM   V
WHERE  ST > 25
```

В версиях SQL до 1992 года этот запрос завершился бы ошибкой, поскольку простая подстановка дала бы синтаксически недопустимое выражение:

```
SELECT CITY
FROM   S
WHERE  SUM ( STATUS ) > 25
GROUP  BY CITY
```

Позже стандарт, как вы, наверное, знаете, был исправлен, однако это еще не значит, что были исправлены и сами продукты! И действительно в последний раз, когда я этим интересовался, существовал, по меньшей мере, один популярный продукт, где эта ошибка еще присутствовала. И именно из-за подобных проблем в этом продукте выборка из представлений реализована посредством *материализации*, а не подстановки; то есть вычисляется выражение, определяющее представление, строится таблица, содержащая результат вычисления, а затем производится выборка из такой материализованной таблицы. И хотя можно говорить, что такая реализация соответствует букве реляционной модели, не думаю, что она соответствует ее духу.

Представления и ограничения

Из *принципа взаимозаменяемости* следует, что представления, как и базовые переменные-отношения, имеют не только предикаты, но и ограничения – как индивидуальные, так и полные ограничения переменной-отношения (см. главу 8). Но, как и предикаты, ограничения, применяемые к представлению *V*, являются *производными* от ограничений для тех переменных-отношений, на основе которых определено *V*, и в соответствии с семантикой реляционных операций, которые участвуют в выражении, определяющем представление. В качестве примера снова рассмотрим представление LS. Оно является ограничением переменной-отношения S, то есть определяющее его выражение задает операцию ограничения над переменной-отношением S, и потому его (полное) ограничение переменной-отношения представляет собой

логическое AND между (полным) ограничением переменной-отношения S и заданным условием ограничения. Предположим для определенности, что к базовой переменной-отношению S применимо только ограничение ключа, обусловленное тем фактом, что {SNO} – ключ. Тогда полное ограничение переменной-отношения для представления LS является результатом применения AND к этому ограничению ключа и к ограничению «город должен быть Лондоном», и представление LS должно в любой момент удовлетворять этому ограничению (другими словами, **золотое правило** применимо к представлениям точно так же, как к базовым переменным-отношениям).

Для простоты я буду, начиная с этого места, употреблять термин *ограничение представления* для обозначения любых ограничений, применимых к некоторому представлению. Из того, что ограничения представления всегда являются производными в описанном выше смысле, еще не следует, что на практике их не нужно объявлять явно. Во-первых, система не всегда достаточно «разумна», чтобы автоматически вывести ограничения, применимые к некоторому представлению. Во-вторых, такие явные объявления полезны хотя бы для документирования (то есть помогают понять семантику представления если не системе, то пользователям). Есть и еще одна причина, о которой я скажу позже.

Таким образом, я утверждаю, что должна иметься возможность явного объявления ограничений для представлений. В частности, должно быть возможно (а) включать явные спецификации KEY и FOREIGN KEY в определения представлений и (б) разрешить указание представления в качестве целевой переменной-отношения в спецификации FOREIGN KEY. Вот пример, иллюстрирующий требование (а):

```
VAR LS VIRTUAL ( S WHERE CITY = 'London' )
    KEY { SNO } ;
```

Язык **Tutorial D** допускает такие спецификации, SQL – нет. **Рекомендация**: В SQL включайте такие спецификации в виде комментариев, например:

```
CREATE VIEW LS
  AS ( SELECT *
       FROM   S
       WHERE  CITY = 'London' )
       /* UNIQUE ( SNO ) */
  WITH CHECK OPTION ;
```

Примечание

Как я только что сказал, SQL не позволяет явно задавать ограничения представлений в определениях представлений, однако логически эквивалентные ограничения можно выразить с помощью предложения CREATE ASSERTION (если, конечно, оно поддерживается). Более общо, CREATE ASSERTION позволяет сформулировать любое ограничение для таблицы, которая могла бы

быть и представлением, где под «таблицей» я понимаю значение, обозначенное произвольным табличным выражением. Чуть ниже я скажу еще несколько слов об этой возможности.

Сказав, что должна быть возможность явно объявлять ограничения для представлений, я должен добавить, что иногда этого лучше не делать, поскольку это может повлечь за собой избыточные проверки. Например, я уже говорил, что спецификация KEY{SNO}, очевидно, применяется к представлению LS, но лишь потому, что она применяется и к базовой переменной-отношению S. Поэтому, если явно объявить ее еще и для представления LS, то мы просто будем проверять одно и то же ограничение дважды. (Но все равно это ограничение следует как-то включить в документацию, потому что оно – часть семантики представления.)

Но, пожалуй, важнее тот факт, что существуют-таки ситуации, когда явное объявление ограничений представления – идея здравая. Вот пример, написанный для определенности в терминах SQL. Пусть даны следующие две базовые таблицы (без деталей):

```
CREATE TABLE FDH
     ( FLIGHT ... , DESTINATION ... , HOUR ... ,
       UNIQUE ( FLIGHT ) ) ;

CREATE TABLE DFGP
     ( DAY ... , FLIGHT ... , GATE ... , PILOT ... ,
       UNIQUE ( DAY , FLIGHT ) ) ;
```

Предикаты этих таблиц можно сформулировать так: *рейс FLIGHT в город DESTINATION вылетает в час HOUR* (для FDH) и *в день DAY посадка на рейс FLIGHT с пилотом PILOT производится из выхода GATE* (для DFGP). На таблицы наложены следующие ограничения (выраженные в виде псевдокода):

```
IF ( f1, n1, h ), ( f2, n2, h ) IN FDH AND
   ( d, f1, g, p1 ), ( d, f2, g, p2 ) IN DFGP
THEN f1 = f2 AND p1 = p2

IF ( f1, n1, h ), ( f2, n2, h ) IN FDH AND
   ( d, f1, g1, p ), ( d, f2, g2, p ) IN DFGP
THEN f1 = f2 AND g1 = g2
```

Объяснение: Первое ограничение говорит, что если (а) в двух строках FDH одинаковы значения часа *h* и (б) в двух строках DFGP, по одной для рейсов *f1* и *f2* из указанных выше строк FDH, одинаковы значения дня *d* и выхода *g*, то обе строки FDH должны совпадать и обе строки DFGP должны совпадать. Иными словами, если известны час, день и выход, то рейс и пилот однозначно определены. Второе ограничение аналогично: если (а) в двух строках FDH одинаковы значения часа *h* и (б) в двух строках DFGP, по одной для рейсов *f1* и *f2* из указанных

выше строк FDH, одинаковы значения дня *d* и пилота *p*, то обе строки FDH должны совпадать и обе строки DFGP должны совпадать. Иными словами, если известны час, день и пилот, то рейс и выход однозначно определены.

Сформулировать эти ограничения непосредственно в терминах базовых таблиц отнюдь не тривиально:

```
CREATE ASSERTION BTCX1 CHECK
 ( NOT ( EXISTS ( SELECT * FROM FDH AS FX WHERE
         EXISTS ( SELECT * FROM FDH AS FY WHERE
         EXISTS ( SELECT * FROM DFGP AS DX WHERE
         EXISTS ( SELECT * FROM DFGP AS DY WHERE
                  FY.HOUR = FX.HOUR AND
                  DX.FLIGHT = FX.FLIGHT AND
                  DY.FLIGHT = FY.FLIGHT AND
                  DY.DAY = DX.DAY AND
                  DY.GATE = DX.GATE AND
                ( FX.FLIGHT <> FY.FLIGHT OR
                  DX.PILOT <> DY.PILOT ) ) ) ) ) ) ) ;

CREATE ASSERTION BTCX2 CHECK
 ( NOT ( EXISTS ( SELECT * FROM FDH AS FX WHERE
         EXISTS ( SELECT * FROM FDH AS FY WHERE
         EXISTS ( SELECT * FROM DFGP AS DX WHERE
         EXISTS ( SELECT * FROM DFGP AS DY WHERE
                  FY.HOUR = FX.HOUR AND
                  DX.FLIGHT = FX.FLIGHT AND
                  DY.FLIGHT = FY.FLIGHT AND
                  DY.DAY = DX.DAY AND
                  DY.PILOT = DX.PILOT AND
                ( FX.FLIGHT <> FY.FLIGHT AND
                  DX.GATE <> DY.GATE ) ) ) ) ) ) ) ;
```

Зато формулировка в виде ограничений ключей в определении представления, будь она разрешена, выглядела бы весьма лаконично:

```
CREATE VIEW V AS
     ( FDH NATURAL JOIN DFGP ,
       UNIQUE ( DAY , HOUR , GATE ) ,     /* гипотетический */
       UNIQUE ( DAY , HOUR , PILOT ) ) ;  /* синтаксис !!!! */
```

Поскольку такое решение нам недоступно, нужно хотя бы задать эти гипотетические ограничения представления в виде утверждений:

```
CREATE VIEW V AS FDH NATURAL JOIN DFGP ;

CREATE ASSERTION VCX1
       CHECK ( UNIQUE ( SELECT DAY , HOUR , GATE FROM V ) ) ;

CREATE ASSERTION VCX2
       CHECK ( UNIQUE ( SELECT DAY , HOUR , PILOT FROM V ) ) ;
```

Для задания этих ограничений даже не обязательно определять представление *V* – достаточно заменить ссылки на *V* в выражении UNIQUE выражением, определяющим представление *V*:

```
CREATE ASSERTION VCX1
       CHECK ( UNIQUE ( SELECT DAY , HOUR , GATE
                        FROM   FDH NATURAL JOIN DFGP ) ) ;

CREATE ASSERTION VCX2
       CHECK ( UNIQUE ( SELECT DAY , HOUR , PILOT
                        FROM   FDH NATURAL JOIN DFGP ) ) ;
```

Операции обновления

Из *принципа взаимозаменяемости* следует, что представления должны допускать обновление (то есть присваивание). Я уже слышу возражения читателей: «Но ведь представление же возможно обновить, разве не так?» Взять, к примеру, представление, которое определено как соединение – многие ко многим, заметим, – переменных-отношений S и P по атрибуту {CITY}; разумеется, для него мы не можем ни вставить, ни удалить кортеж, так? *Примечание*: Приношу извинения за небрежность формулировок; как мы знаем из главы 5, в реляционной модели нельзя говорить о «вставке или удалении кортежа». Но излишняя педантичность в данном случае только помешала бы пониманию сути дела.

Что ж, даже если мы действительно не можем вставить или удалить кортеж для S JOIN P (впрочем, это еще вилами на воде писано), позвольте заметить, что некоторые обновления некоторых *базовых* переменных-отношений тоже невозможны. Например, вставить кортеж в переменную-отношение SP не получится, если содержащееся в этом кортеже значение SNO отсутствует в переменной-отношении S. Таким образом, обновление базовых переменных-отношений может завершаться неудачно из-за нарушения ограничений целостности, и *то же самое справедливо для обновления представлений*. Иными словами, дело не в том, что некоторые представления принципиально не обновляемы, а в том, что некоторые обновления некоторых представлений не проходят из-за нарушения ограничений целостности (то есть нарушений **золотого правила**). *Примечание*: На самом деле обновления – как базовых переменных-отношений, так и представлений, – могут завершаться и ошибкой и вследствие нарушения *принципа присваивания*. Но для простоты я не стану обращать на это внимания.

Итак, пусть *V* – представление. Чтобы корректно поддержать обновление *V*, система должна знать полное ограничение для *V*, назовем его *VC*. Иначе говоря, она должна уметь выводить *ограничения*, то есть, зная, какие ограничения применяются к переменным-отношениям, в терминах которых определено *V*, уметь вычислить *VC*. Но вы наверняка знаете, что современные SQL-системы очень плохо справляются с задачей

вывода ограничений. Поэтому и поддержка обновления представления разработана так слабо (и это относится не только к основным продуктам, но и к самому стандарту). Фактически SQL-системы обычно позволяют обновлять лишь представления, которые определены как простое сочетание ограничения и проекции для одной базовой таблицы (и даже тут есть сложности). Обратимся снова к представлению LS. Это всего лишь ограничение базовой таблицы S, поэтому мы можем выполнить для него следующую операцию DELETE:

```
DELETE
FROM   LS
WHERE  LS.STATUS > 15 ;
```

Эта операция отображается на такую:

```
DELETE
FROM   S
WHERE  S.CITY = 'London'
AND    S.STATUS > 15 ;
```

Однако, как я уже отметил, немногие продукты поддерживают обновление более сложных представлений. *Примечание*: я еще вернусь к вопросу об обновляемости представлений в SQL в подразделе «Еще об SQL».

Опция CHECK

Рассмотрим следующее SQL-предложение INSERT для представления LS:

```
INSERT INTO LS ( SNO , SNAME , STATUS , CITY )
       VALUES ( 'S6' , 'Lopez' , 30 , 'Madrid' ) ;
```

Эта операция отображается на такую:

```
INSERT INTO S ( SNO , SNAME , STATUS , CITY )
       VALUES ( 'S6' , 'Lopez' , 30 , 'Madrid' ) ;
```

(Изменено лишь имя обновляемой базовой таблицы.) Заметим, что «новая строка» нарушает полное ограничение для представления LS, потому что город отличен от Лондона. Так что же произойдет? По умолчанию SQL вставит строку в базовую таблицу S, но поскольку она не удовлетворяет выражению, определяющему представление (именно представление LS), то в этом представлении строка не будет видна. Следовательно, с точки зрения представления LS новая строка просто пропала из виду (можно также сказать, что с той же точки зрения операция INSERT оказалась «пустой»). Однако это нарушение *принципа присваивания*.

Полагаю, никого не нужно убеждать, что такое поведение логически некорректно. В языке **Tutorial D** оно попросту невозможно. Что же касается SQL, то для решения этой проблемы в нем существует опция CHECK: если (и только в этом случае) для представления задана спецификация WITH CASCADED CHECK OPTION, то любое его обновле-

ние должно быть согласовано с выражением, определяющим представление. **Рекомендация:** Всюду, где возможно, включайте в определения представлений спецификацию WITH CASCADED CHECK OPTION. Но имейте в виду, что SQL допускает наличие такой спецификации лишь в том случае, когда считает представление обновляемым, а (как уже отмечалось) не все представления, логически допускающие обновление, считаются таковыми в SQL. *Примечание*: Альтернативой слову CASCADED служит LOCAL, но не пользуйтесь этой возможностью.[1] Впрочем, можно вообще не указывать ни CASCADED, ни LOCAL, так как CASCADED подразумевается по умолчанию.

Еще об SQL

Как мы видели, поддержка обновления представлений в SQL ограничена. К тому же в ней очень трудно разобраться – стандарт в этом вопросе еще более непостижим, чем обычно! Следующая цитата (лишь немного подредактированная) дает представление о возникающих сложностях:

> <Выражение запроса> (query expression) *QE1 допускает обновление* тогда и только тогда, когда для любого <выражения запроса> (query expression) или <спецификации запроса> (query specification) *QE2*, которое просто содержится (simply contained) в *QE1*:
>
> a) *QE1* содержит *QE2* без промежуточного <выражения запроса без соединения> (non join query expression) вида UNION DISTINCT, EXCEPT ALL или EXCEPT DISTINCT.
>
> b) Если *QE1* просто содержит <выражение запроса без соединения> (non join query expression) *NJQE* вида UNION ALL, то:
>
> i) *NJQE* непосредственно содержит <выражение запроса> (query expression) *LO* и <терм запроса> (query term) *RO* такие, что никакая листовая обобщенно исходная таблица *LO* (leaf generally underlying table) не является также листовой обобщенно исходной таблицей *RO*.
>
> ii) Для каждого столбца *NJQE* исходные столбцы (underlying columns) таблиц, определяемые соответственно по *LO* и *RO*, одновременно допускают или не допускают обновление.
>
> c) *QE1* содержит *QE2* без промежуточного <терма запроса без соединения> (non join query term) вида INTERSECT.
>
> d) *QE2* допускает обновление.

[1] Семантика WITH LOCAL CHECK OPTION слишком причудлива, чтобы описывать ее здесь во всех подробностях. Но в любом случае непонятно, кому может понадобиться такая семантика; невольно закрадывается подозрение, что она была включена в стандарт только для того, чтобы производители каких-то существовавших в то время некорректных реализаций могли заявить о соответствии стандарту.

Вот что лично я думаю об этой цитате...

- Это лишь одно из многих правил, которые нужно учитывать при определении того, допускает ли данное представление обновление в SQL.
- Правила, относящиеся к этому вопросу, приведены не в одном месте, а разбросаны по разным частям документа.
- Формулировки этих правил зависят от различных дополнительных понятий и конструкций, например: обновляемые столбцы, листовые обобщенно исходные таблицы, <термы запроса без соединения>, которые сами определены в других частях документа.

Поэтому я даже не буду пытаться точно описать, какие представления в SQL считаются допускающими обновление. Но, говоря нестрого, к ним относятся, по крайней мере, следующие:

1. Представления, определенные как ограничение и/или проекция одной базовой таблицы.
2. Представления, определенные как соединение один-к-одному или один-ко-многим двух базовых таблиц (в случае соединения один-ко-многим обновление допускает только таблица на стороне «многие»).
3. Представления, определенные как результат операции UNION ALL или INTERSECT для двух различных базовых таблиц.
4. Некоторые комбинации пунктов 1–3 выше.

Но даже эти случаи обрабатываются некорректно из-за того, что SQL «не понимает», что такое вывод ограничений, **золотое правило** и *принцип присваивания*, а также из-за наличия в SQL null-значений и строк-дубликатов. Картина становится еще запутаннее вследствие того, что в SQL выделено четыре разных случая, когда представление допускает обновление: *обновляемое*, *потенциально обновляемое*, *просто обновляемое* и *допускающее вставку*. В стандарте эти термины определены формально, но нет даже намека на то, что они означают на интуитивном уровне и почему так названы. Но я хотя бы могу сказать, что «обновляемое» относится к UPDATE и DELETE, а «допускающее вставку» – к INSERT, и что представление не может допускать вставку, не будучи обновляемым. Отмечу, однако, что могут существовать представления, которые допускают одни операции обновления, но не допускают другие (например, DELETE, но не INSERT), и что операции DELETE и INSERT не обязательно являются взаимно обратными. Оба эти факта, если их можно назвать фактами, также являются нарушениями *принципа взаимозаменяемости*.

Что касается случая 1 выше, то я могу немного уточнить ситуацию. Конкретно, SQL-представление, безусловно, допускает обновление, если одновременно удовлетворяются следующие условия:

- Выражение, определяющее представления, – это либо (а) простое выражение SELECT (без UNION, INTERSECT и EXCEPT), либо (б) «явная таблица» (см. главу 12), которая логически эквивалентна такому

выражению. *Примечание*: Для простоты я в дальнейшем буду предполагать, что случай (б) автоматически преобразуется в случай (а).

- Во фразе SELECT этого выражения SELECT задан спецификатор ALL (явно или неявно).
- После раскрытия всех элементов, заданных звездочками, каждый элемент в списке является простым именем столбца (возможно, квалифицированным или содержащим спецификатор AS), и ни один элемент не встречается более одного раза.
- Фраза FROM в этом выражении SELECT содержит ровно одну ссылку на таблицу (см. главу 12), и табличное выражение в этой ссылке представляет собой только имя (скажем, *T*) таблицы, допускающей обновление (либо базовой таблицы, либо обновляемого представления).
- Если в этом выражении SELECT присутствует фраза WHERE, то она не содержит подзапросов, в которых фраза FROM ссылается на *T*.
- В этом выражении SELECT нет фраз GROUP BY или HAVING.

Рекомендация: Настаивайте на том, чтобы производители SQL-систем как можно скорее улучшили поддержку в части обновления представлений.

Зачем нужны представления?

До сих пор в этой главе я молчаливо предполагал, что вам известно, для чего нужны представления, но теперь все же хочу сказать несколько слов на эту тему. Фактически представления служат двум совершенно разным целям.

- Пользователь, который определяет представление *V*, очевидно, знает соответствующее выражение *X*, определяющее представление. Поэтому он может использовать имя *V* всюду, где подразумевается выражение *X*. Но такое применение – не более чем сокращенная запись, и автор представления именно так это и понимает.
- Напротив, пользователь, которому известно лишь, что *V* существует и доступно для применения, предположительно (по крайней мере, в идеале) *не знает* о выражении *X*. Для такого пользователя *V* должно выглядеть и вести себя, как базовая переменная-отношение, о чем я уже подробно говорил в этой главе. Именно этот второй случай по-настоящему важен, и именно его я имел в виду до сих пор.

Логическая независимость от данных

Вторая из названных выше целей тесно связана с вопросом о *логической независимости от данных*. Напомню (см. главу 1), что под физической независимостью от данных мы понимаем возможность изменить способ физического хранения данных и доступа к ним, не меняя воспри-

ятия данных пользователем. Неудивительно поэтому, что под логической независимостью от данных понимается возможность изменить способ логического хранения данных и доступа к ним, не меняя восприятия данных пользователем. И обеспечить такую логическую независимость призваны именно представления.

Предположим, что по какой-то причине (по какой именно, нам сейчас не важно) мы хотим следующим образом заменить базовую переменную-отношение S двумя базовыми переменными-отношениями LS и NLS:

```
VAR LS BASE RELATION        /* поставщики в Лондоне */
  { SNO CHAR , SNAME CHAR , STATUS INTEGER , CITY CHAR }
  KEY { SNO } ;

VAR NLS BASE RELATION       /* поставщики не в Лондоне */
  { SNO CHAR , SNAME CHAR , STATUS INTEGER , CITY CHAR }
  KEY { SNO } ;
```

Мы уже говорили в этой главе о том, что старая переменная-отношение S представляет собой дизъюнктное объединение LS и NLS (и LS, и NLS – ограничения S). Поэтому можно определить представление, объединяющее эти переменные-отношения, и назвать его S:

```
VAR S VIRTUAL ( LS D_UNION NLS ) KEY { SNO } ;
```

Любое выражение, которое раньше ссылалось на базовую переменную-отношение S, теперь ссылается на представление S. В предположении, что система корректно поддерживает операции над представлениями, – к сожалению, слишком смелое предположение, принимая во внимание современное состояние дел, – пользователи не заметят этого изменения в логической структуре базы данных.

Попутно замечу, что замена исходной переменной-отношения S («поставщики») двумя ограничениями LS и NLS не совсем тривиальное действие. В частности, нужно что-то делать с переменной-отношением SP («поставки»), так как для нее задан внешний ключ, который ссылается на исходную переменную-отношение. См. упражнение 9.11 в конце главы.

Взгляды и снимки

На протяжении этой главы я употреблял термин *представление* в его исконном смысле, то есть том, который был определен в оригинальной реляционной модели. Но в последние годы возникла терминологическая путаница: в академических кругах – безусловно, а в коммерческих – до некоторой степени. Напомню, что представление можно рассматривать как производную переменную-отношение. Но существует и другой вид производной переменной-отношения, называемый *снимком*. Снимок – вещь, хотя и производная, но не виртуальная, то есть он

существует не только как определение в терминах других переменных-отношений, но и как отдельная копия данных (по крайней мере, концептуально). Например (изобретаю синтаксис на ходу):

```
VAR LSS SNAPSHOT ( S WHERE CITY = 'London' )
    KEY { SNO }
    REFRESH EVERY DAY ;
```

Определение снимка аналогично выполнению запроса со следующими отличиями:

- Результат запроса сохраняется в базе данных под указанным именем (в данном примере LSS) в виде константы-отношения или «переменной-отношения, доступной только для чтения» (если не считать периодического обновления, см. следующий пункт).
- Периодически (в примере каждый день – EVERY DAY) снимок обновляется, то есть текущее значение отбрасывается и заменяется новым значением снимка.

Следовательно, в этом примере LSS представляет данные в том виде, в котором они находились не более 24 часов назад.

Снимки важны в хранилищах данных, распределенных системах и во многих других контекстах. Но во всех случаях за ними стоит одно и то же допущение (а иногда это даже требование) – что система может примириться с данными на конкретный момент времени в прошлом. Типичные приложения такого рода – составление отчетов и бухгалтерия; здесь обычно необходимо, чтобы данные были в нужный момент заморожены (например, в конце отчетного периода), а снимки позволяют осуществить такое замораживание, не блокируя работу других приложений.

Пока все хорошо. Беда в том, что снимки получили известность (по крайней мере, в определенных кругах) под названием *материализованные представления*. Но это не представления! Представления никогда и не предполагалось материализовывать[1]; как мы видели, предполагается, что операции над представлениями реализуются посредством отображения их на подходящие операции с исходными переменными-отношениями. Таким образом, «материализованное представление» – внутренне противоречивый термин. Хуже того, сам термин *представление* безо всяких прилагательных все чаще воспринимают как «материализованное представление» (опять же в определенных кругах), поэтому нам грозит опасность утратить хорошее слово для обозначения представлений в их первоначальном смысле. В этой книге я употребляю термин *представление* только в исконном смысле, но имейте в виду, что в других местах под ним могут понимать нечто иное. **Реко-**

[1] Несмотря на тот факт, что, как мы видели выше, по крайней мере один продукт на рынке иногда производит материализацию представлений.

мендации: Никогда не употребляйте неуточненный термин *представление* для обозначения снимка; никогда не употребляйте термин *материализованное представление* и будьте начеку, памятуя о том, что другие люди могут нарушать эти рекомендации.

Упражнения

Упражнение 9.1. Определите представление, состоящее из номеров поставщиков и номеров деталей для тех поставщиков и деталей, которые находятся в разных городах. Приведите варианты на SQL и **Tutorial D**.

Упражнение 9.2. Пусть представление LSSP определено следующим образом (на SQL):

```
CREATE VIEW LSSP
  AS ( SELECT S.SNO , S.SNAME , S.STATUS , SP.PNO , SP.QTY
       FROM   S NATURAL JOIN SP
       WHERE  S.CITY = 'London' ) ;
```

Вот запрос к этому представлению:

```
SELECT DISTINCT STATUS , QTY
FROM   LSSP
WHERE  PNO IN
     ( SELECT PNO
       FROM   P
       WHERE  CITY <> 'London' )
```

Как мог бы выглядеть реально выполняемый запрос с участием базовых таблиц?

Упражнение 9.3. Какой ключ (или ключи) имеет представление LSSP из предыдущего упражнения? Как можно сформулировать предикат для этого представления?

Упражнение 9.4. Пусть дано такое определение представления на языке **Tutorial D**:

```
VAR HEAVYWEIGHT VIRTUAL
  ( ( P RENAME ( WEIGHT AS WT , COLOR AS COL ) )
      WHERE WT > 14.0 ) { PNO , WT , COL } ;
```

Для каждого из следующих выражений и предложений напишите, во что оно преобразуется после применения процедуры подстановки:

a. `HEAVYWEIGHT WHERE COL = 'Green'`

b. `( EXTEND HEAVYWEIGHT ADD ( WT + 5.3 AS WTP ) ) { PNO , WTP }`

c. `INSERT HEAVYWEIGHT RELATION { TUPLE { PNO 'P99' , WT 12.0 , COL 'Purple' } } ;`

d. `DELETE HEAVYWEIGHT WHERE WT < 9.0 ;`

e. `UPDATE HEAVYWEIGHT WHERE WT = 18.0 : { COL := 'White' } ;`

Упражнение 9.5. Пусть определение представления HEAVYWEIGHT из упражнения 9.4 модифицировано следующим образом:

```
VAR HEAVYWEIGHT VIRTUAL
( ( ( EXTEND P ADD ( WEIGHT * 454 AS WT ) )
    RENAME ( COLOR AS COL ) ) WHERE WT > 6356.0 )
                                  { PNO , WT , COL } ;
```

(то есть атрибут WT теперь обозначает вес в граммах, а не в фунтах). Повторите задания из упражнения 9.4.

Упражнение 9.6. Приведите решения упражнений 9.4 и 9.5 на SQL.

Упражнение 9.7. Для каких SQL-представлений из предыдущих упражнений следует задавать спецификацию WITH CHECK OPTION?

Упражнение 9.8. Какие ключи есть у представлений из упражнений 9.4 и 9.5? Какие у них предикаты? Каковы полные ограничения переменных-отношений?

Упражнение 9.9. Приведите столько причин, сколько сможете придумать, в обоснование желательности объявления ключей для представления.

Упражнение 9.10. Пользуясь базой данных о поставщиках и деталях или любой другой знакомой вам базой, приведите дополнительные примеры (помимо примера с поставщиками из Лондона и не из Лондона), доказывающие, что решение о том, какие переменные-отношения делать базовыми, а какие – виртуальными, носит в значительной мере произвольный характер.

Упражнение 9.11. Обсуждая вопрос о логической независимости от данных, я рассмотрел возможность реструктуризации (то есть изменения логической структуры) базы данных о поставщиках и деталях путем замены базовой переменной-отношения S двумя ее ограничениями (LS и NLS). Однако я заметил, что такая замена – не вполне тривиальное действие. Почему?

Упражнение 9.12. Исследуйте какой-нибудь доступный вам продукт на основе SQL.

a. Существуют ли какие-нибудь допустимые на первый взгляд запросы, которые в этом продукте не работают? Если да, то сформулируйте максимально точно, что это за запросы. Какие доводы поставщик приводит в обоснование неполной поддержки?

b. Какие обновления и каких именно представлений поддерживаются в этом продукте? Постарайтесь ответить как можно точнее. Совпадают ли правила обновления с теми, что сформулированы в стандарте?

c. Более общо, как именно этот продукт нарушает *принцип взаимозаменяемости* (а нарушает обязательно!)?

Упражнение 9.13. Опишите различие между представлениями и снимками. Поддерживаются ли снимки в SQL? А в каком-нибудь известном вам продукте?

Упражнение 9.14. Что такое «материализованное представление»? Почему этот термин не следует употреблять?

Упражнение 9.15. Рассмотрим базу данных о поставщиках и деталях, но для простоты забудем о переменной-отношении «детали». Вот набросок двух возможных способов описать поставщиков и поставки:

a. `S { SNO , SNAME , STATUS , CITY }`

   `SP { SNO , PNO , QTY }`

b. `SSP { SNO , SNAME , STATUS , CITY , PNO , QTY }`

   `XSS { SNO , SNAME , STATUS , CITY }`

Вариант (a) – стандартный. В варианте (b) переменная-отношение SSP содержит кортеж для каждой поставки, в котором хранятся номер детали, количество ее в поставке и полная информация о поставщике, а переменная-отношение XSS содержит информацию о тех поставщиках, которые не поставили ни одной детали. (Можно ли сказать, что эти варианты информационно эквивалентны?) Напишите определения представлений, выражающих вариант (b) через вариант (a) и наоборот. Кроме того, покажите ограничения, применимые к каждому варианту. Есть ли у какого-нибудь варианта очевидные преимущества по сравнению с другим? Если да, то в чем они состоят.

Упражнение 9.16. Предполагается, что представления обеспечивают логическую независимость от данных. Но не говорил ли я в главе 6, что для решения этой задачи предназначен гипотетический механизм «открытых таблиц»? Как вы относитесь к такому несоответствию?

10

SQL и формальная логика

В главе 1 я упоминал, что реляционной алгебре есть альтернатива – реляционное исчисление. Это означает, что все запросы, ограничения, определения представлений и так далее можно сформулировать не только в терминах реляционной алгебры, но и реляционного исчисления. Иногда последняя формулировка оказывается даже проще, хотя бывает и наоборот.

Что такое реляционное исчисление? По существу, это прикладная форма исчисления предикатов (известного также под названием логика предикатов), приспособленная под нужды реляционных баз данных. Поэтому цель настоящей главы – познакомить вас с некоторыми средствами логики предикатов (для краткости будем называть ее просто *логикой*), показать, как эти средства реализуются конкретно в реляционном исчислении, и по ходу дела рассмотреть соответствующие средства SQL.

Кстати говоря, из сказанного выше следует, что реляционный язык может быть основан как на алгебре, так и на исчислении. Например, язык **Tutorial D** явным образом основан на алгебре (поэтому на него в этой главе не так уж много ссылок), а язык Query-By-Example (см. приложение D) столь же явно основан на исчислении. А на чем основан SQL? Ответ, к сожалению, неоднозначен: серединка на половинку... Когда SQL только проектировался, ставилась задача сделать его отличным и от алгебры, и от исчисления; к слову сказать, именно из-за этого в языке появился «подзапрос IN».[1] Но со временем выяснилось, что без некоторых

[1] Эта цель базировалась на фундаментальном заблуждении, будто и реляционная алгебра, и реляционное исчисление в какой-то мере «враждебны пользователю». Но такая характеристика, как мне кажется, основана на смешении синтаксиса и семантики. Конечно, синтаксис в работах Кодда способен напугать человека, не искушенного в математическом формализме. Но семантика – совсем другое дело; и у алгебры, и у исчисления семан-

средств алгебры и исчисления не обойтись, и развивающийся язык впитал их. Сегодня ситуация такова, что некоторые аспекты SQL «похожи на алгебру», некоторые «похожи на исчисление», а некоторые ни на что не похожи. А отсюда следует, что большинство запросов, ограничений и т. д., которые вообще можно выразить на SQL, выражаются многими разными способами, о чем я мимоходом упомянул в главе 6.

Простые и составные высказывания

Напомню (см. главу 5), что в логике высказыванием называется суждение, которое безусловно принимает значение TRUE или FALSE. Вот несколько примеров (1, 4 и 5 – истинные высказывания, 2 и 3 – ложные):

1. `2 + 3 = 5`
2. `2 + 3 > 7`
3. `Юпитер – звезда`
4. `У Марса два спутника`
5. `Венера расположена между Землей и Меркурием`

Связки

Имея некоторое множество высказываний, мы можем строить из них новые с помощью *связок*. Чаще всего на практике встречаются связки NOT, AND, OR, IF ... THEN ... (известная также под названием IMPLIES, или «⇒») IF AND ONLY IF (известная также под названием IFF, или BI-IMPLIES, или IS EQUIVALENT TO, или «⇔», или «≡»). Вот несколько примеров высказываний, которые можно образовать из высказываний 3, 4 и 5 в предыдущем списке:

6. `( Юпитер – звезда ) OR ( У Марса два спутника )`
7. `( Юпитер – звезда ) AND ( Юпитер – звезда )`
8. `( Венера расположена между Землей и Меркурием ) AND NOT (Юпитер – звезда)`
9. `IF ( У Марса два спутника ) THEN ( Венера расположена между Землей и Меркурием )`
10. `IF ( Юпитер – звезда ) THEN ( У Марса два спутника )`

Примечание

Я поставил скобки, чтобы область действия связок была понятнее. На практике применяются правила предшествования, которые позволяют опускать многие скобки, которые в противном случае были бы необходимы. Конечно, если

тика очень проста (на мой взгляд), и, как продемонстрировали многие авторы, совсем несложно обернуть эту семантику вполне дружелюбным к пользователю синтаксисом.

включить логически избыточные скобки, ничего плохого не произойдет, а иногда они помогают быстрее понять смысл выражения.

Вообще говоря, связки можно рассматривать как *логические операторы* – они принимают некоторые высказывания на входе и возвращают новое высказывание на выходе. NOT – одноместный оператор, остальные – двуместные. Если в высказывании нет связок, оно называется *простым*, в противном случае – *составным*. Значение истинности составного высказывания можно определить, зная значения истинности составляющих его простых высказываний и применяя следующие таблицы истинности (из соображений экономии места я сократил TRUE и FALSE до T и F сответственно):

p	NOT *p*
T	F
F	T

p q	*p* AND *q*	*p* OR *q*	IF *p* THEN *q*	*p* IFF *q*
T T	T	T	T	T
T F	F	T	F	F
F T	F	T	T	F
F F	F	F	T	T

Кстати говоря, таблицы истинности можно изображать и несколько по-другому (опять же для экономии места я сократил IF ... THEN ... до IF):

NOT	
T	F
F	T

AND	T	F
T	T	F
F	F	F

OR	T	F
T	T	T
F	T	F

IF	T	F
T	T	F
F	T	T

IFF	T	F
T	T	F
F	F	T

Иногда удобнее один способ, иногда – другой. Как бы то ни было, рассмотрим внимательнее одно из приведенных выше составных высказываний.

9. `IF ( У Марса два спутника ) THEN (Венера расположена между Землей и Меркурием)`

Это высказывание имеет вид IF *p* THEN *q* (или, эквивалентно, *p* IMPLIES *q*), где *p* – *посылка*, а *q* – *следствие*. Так как и посылка, и следствие равны TRUE, то и все высказывание принимает значение TRUE, как легко видеть из таблицы истинности. Но очевидно, что расположение Венеры между Землей и Меркурием не имеет ничего общего с тем фактом, что у Марса два спутника! Так что же тут происходит?

Предыдущий пример иллюстрирует проблему, с которой часто сталкиваются люди, не имеющие опыта работы с формальной логикой, а именно трудности усвоения импликации. Поэтому я хотел бы привести следующее рассуждение, надеясь, что оно немного прояснит суть дела.

- Прежде всего, отметим, что существует ровно 16 двуместных связок, которые соответствуют 16 возможным таблицам истинности (из которых выше показано только четыре).
- Некоторые, но не все из этих 16 двуместных связок имеют названия, например AND и OR. Но эти названия – не более чем мнемоники; у них нет никакой внутренней семантики и выбраны они лишь потому, что поведение связок похоже (но не обязательно идентично) на тот смысл, который принято придавать соответствующим союзам в естественном языке. Действительно, легко видеть, что даже AND означает не в точности то же самое, что «и» в естественном языке. В логике *p* AND *q* и *q* AND *p* – эквивалентные высказывания, а в естественном языке это необязательно. Вот простой пример: предложения

  «Я был сильно разочарован и голосовал за смену лидера»

  и

  «Я голосовал за смену лидера и был сильно разочарован».

  очевидно, не эквивалентны! Другими словами, связка AND – своего рода логически очищенный идеал союза «и»; очень важно, что ее смысл *не зависит от контекста* (чего никак нельзя сказать о союзе «и»). Аналогичные соображения применимы и к остальным связкам.
- Из 16 двуместных связок поведение IMPLIES наиболее близко напоминает то, что называется импликацией (следствием) в естественном языке. Например, «если Марс имеет два спутника, то, конечно же, он имеет хотя бы один спутник» – правильная импликация как в логике, так и в естественном языке. Но никто не стал бы, да и не должен утверждать, что логическая и «естественная» импликация – одно и то же. На самом деле логическая импликация, как и все остальные связки, по необходимости имеет *формальное определение*, то есть она определена исключительно в терминах значений истинности, а не семантики операндов. Сказать такое об аналоге импликации в естественном языке, очевидно, нельзя.
- Рассмотрим еще один пример (десятый в списке выше):

  ```
  IF ( Юпитер - звезда ) THEN ( У Марса два спутника )
  ```

  Хотя и вразрез с интуицией, значением этого высказывания тоже является TRUE, поскольку посылка ложна (сверьтесь с таблицей истинности); но очевидно же, что наличие у Марса двух спутников не имеет ничего общего с тем, звезда Юпитер или нет. Как и раньше, отчасти обоснованием того, что импликация в этом случае принимает значение TRUE, является формальное определение связки IMPLIES. Однако в данном случае есть и еще один довод (фактически, из области баз данных), который может показаться вам более убедительным. Предположим, что на базу данных о поставщиках и деталях наложено ограничение, согласно которому красные детали долж-

ны находиться на складе в Лондоне (я сознательно формулирую это ограничение в слегка упрощенном виде):

```
IF ( COLOR = 'Red' ) THEN ( CITY = 'London' )
```

Понятное дело, мы не хотим, чтобы это ограничение нарушалось из-за деталей другого цвета. А значит, высказывание в целом (логическая импликация) должно принимать значение TRUE, если посылка равна FALSE.

Из всего сказанного выше следует, что высказывание *p* IMPLIES *q* (или, что то же самое, IF *p* THEN *q*) логически эквивалентно высказыванию (NOT *p*) OR *q* – оно принимает значение FALSE тогда и только тогда, когда *p* равно TRUE и *q* равно FALSE. И это наблюдение иллюстрирует тот факт, что не все связки NOT, AND, OR, IMPLIES и IF AND ONLY IF – примитивные; некоторые можно выразить через другие. Фактически все возможные одноместные и двуместные связки можно выразить через NOT и либо AND, либо OR в подходящих сочетаниях. (*Упражнение*: проверьте это утверждение.) Но, пожалуй, еще интереснее, что любую связку можно выразить всего через один примитив. Сумеете ли вы отыскать его?

Замечание о коммутативности

Связки AND и OR коммутативны; то есть составные высказывания *p* AND *q* и *q* AND *p* эквивалентны. Следовательно, никогда не следует писать содержащий такие высказывания код, в котором предполагается, что *p* будет вычислено прежде *q* или наоборот. Например, пусть функция SQRT (неотрицательный квадратный корень) определена так, что в случае отрицательного аргумента возбуждается исключение. Рассмотрим следующее SQL-выражение:

```
SELECT ...
FROM   ...
WHERE  X >= 0 AND SQRT ( X ) ...
```

Не гарантируется, что это выражение не возбудит исключение, поскольку функция SQRT может быть вызвана еще до проверки неотрицательности X.

Простые и составные предикаты

Рассмотрим следующие суждения:

11. *x* – звезда
12. *x* имеет два спутника
13. *x* имеет *m* спутников
14. *x* расположен между Землей и *y*
15. *x* расположен между *y* и *z*

Здесь *x*, *y*, *z* и *m* – *параметры*, которые иногда называют *метками-заменителями* (*placeholders*). Следовательно, эти суждения не являются высказываниями (нельзя сказать, что они безусловно истинны или ложны) – именно потому, что содержат параметры. Например, суждение «*x* – звезда» зависит от параметра *x*, и потому мы не можем сказать, истинно оно или ложно, пока не будем знать, что скрывается за *x*. А узнав это, мы уже будем иметь дело не с данным суждением, а с неким другим, что станет ясно из следующего абзаца.

Мы можем подставлять *аргументы* вместо параметров и тем самым получать высказывания из параметризованных суждений. Например, если подставить аргумент *Солнце* вместо параметра *x* в суждение «*x* – звезда», то получится «Солнце – звезда». И это суждение уже является высказыванием, потому что оно безусловно истинно или ложно (на самом деле, конечно, истинно). Но исходное суждение («*x* – звезда»), повторю, не является высказыванием. А является оно *предикатом*, который (см. главу 5) представляет собой функцию, возвращающую значение истинности. Как и у любой функции, у предиката имеется множество параметров; при вызове вместо параметров подставляются аргументы, в результате чего предикат превращается в высказывание. Говорят, что аргументы *удовлетворяют* предикату, если это высказывание истинно. Например, аргумент *Солнце* удовлетворяет предикату «*x* – звезда», а аргумент *Луна* – не удовлетворяет.

Попутно напомню, что в формальной логике говорят не о вызове, а о *порождении* предиката (на самом деле, есть причины, которые здесь нас не интересуют, по которым понятие порождения несколько более общее, чем хорошо знакомое понятие вызова функции). Однако в этой главе я все же буду употреблять термин *вызов*. Отмечу еще, что в упражнении 5.23 из главы 5 было показано, что высказывание можно считать вырожденным предикатом с пустым множеством параметров (а функция, которой этот предикат и является, при каждом вызове возвращает один и тот же результат – TRUE или FALSE). Иными словами, все высказывания – предикаты, но большинство предикатов высказываниями не является.

Теперь рассмотрим предикат «*x* имеет *m* спутников» с двумя параметрами: *x* и *m*. Подстановка вместо *x* аргумента *Марс*, а вместо *m* – *2* дает истинное высказывание; подстановка вместо *x* аргумента *Земля*, а вместо *m* – *2* дает ложное. На самом деле предикаты удобно классифицировать по кардинальности множества параметров. *N-местным* называется предикат, имеющий ровно *n* параметров; например, «*x* расположен между *y* и *z*» – 3-местный предикат, а «*x* имеет *m* спутников» – 2-местный. Высказывание – это 0-местный предикат. *Примечание*: иногда *n*-местный предикат называют также *n-адическим*. Если *n* = *1*, то предикат называется монадическим, если *n* = *2* – диадическим.

Если задано некое множество предикатов, то входящие в него предикаты можно комбинировать для получения новых, применяя логические

связки (NOT, AND, OR и т. д.); другими словами, связки – это логические операторы, которые применяются к предикатам общего вида, а не только к высказываниям, являющимся частными случаями предикатов. Предикат без связок называется *простым*; предикат, не являющийся простым, называется *составным*. Вот пример составного предиката:

16. (*x* – звезда) OR (*x* расположен между Землей и *y*)

Это двуместный предикат – не потому, что он состоит из двух простых предикатов, а потому, что имеет два параметра: *x* и *y*.

Квантификация

В предыдущем разделе я показал, что один из способов получить высказывание из предиката – вызвать его с подходящими аргументами. Но есть и другой способ – *квантификация*. Пусть *p(x)* – одноместный предикат (для ясности я решил показать его единственный параметр). Тогда:

- Выражение

  EXISTS *x* (p (*x*))

  является высказыванием и означает: «Существует хотя бы одно допустимое значение аргумента *a*, соответствующего параметру *x*, такое, что *p(a)* обращается в TRUE» (иными словами, значение аргумента *a* удовлетворяет предикату *p*). Например, если *p* – предикат «*x* является логиком», то

  EXISTS *x* (*x* является логиком)

  является высказыванием, принимающим значение TRUE (возьмите, например, в качестве *a* Бертрана Рассела).

- Выражение

  FORALL *x* (p (*x*))

  является высказыванием и означает: «Все допустимые значения аргумента *a*, соответствующего параметру *x*, таковы, что *p(a)* обращается в TRUE» (иными словами, все такие значения аргумента *a* удовлетворяют предикату *p*). Например, взяв в качестве *p* все тот же предикат «*x* является логиком», получим высказывание

  FORALL *x* (*x* является логиком)

  которое принимает значение FALSE (в качестве примера, возьмите Джорджа Буша).

Отметим, что для доказательства истинности высказывания EXISTS достаточно предъявить один пример, а для доказательства ложности высказывания FORALL – один контрпример. Отметим также, что в обоих случаях параметр должен быть ограничен множеством допус-

тимых значений (в нашем примере множеством всех людей). К этой мысли я еще вернусь в разделе «Реляционное исчисление» ниже.

В формальной логике выражения вида EXISTS *x* и FORALL *x* называются *кванторами* (термин происходит от глагола *quantify*, который означает «*давать количественное выражение*» – каков объем чего-либо или сколько экземпляров чего-либо). EXISTS называется квантором *существования*, а FORALL – квантором *всеобщности*. В текстах по математической логике квантор EXISTS обычно представляется отраженной буквой E (∃), а квантор FORALL – перевернутой буквой A (∀). Я же буду для удобства восприятия употреблять ключевые слова EXISTS и FORALL.

Отступление

В этом месте один из рецензентов спросил, а не является ли квантор просто еще одной связкой. Нет, не является. Пусть *p* и *q* – предикаты с одним параметром *x*. Тогда *p* и *q* можно разными способами комбинировать с помощью связок, но в результате всегда будет получаться предикат с тем же единственным параметром *x*. С другой стороны, квантификация по *x*, то есть образование такого выражения, как EXISTS *x* (*p*) или FORALL *x* (*q*), преобразует рассматриваемый предикат в высказывание. Следовательно, существует четкое логическое различие между двумя этими понятиями. (Хотя должен добавить, что, по крайней мере, в контексте баз данных кванторы можно определить в терминах некоторых связок. Я вернусь к этому моменту позже, в разделе «Еще о квантификации».)

В качестве еще одного примера рассмотрим двуместный предикат «*x* выше, чем *y*». Если предварить его квантором существования по *x*, то получим:

```
EXISTS x ( x выше, чем y )
```

Это суждение не является высказыванием, поскольку у него нет безусловного значения истинности; на самом деле, это одноместный предикат с единственным параметром *y*. Вызовем этот предикат с аргументом Стив, получим:

```
EXISTS x ( x выше, чем Стив )
```

Это суждение является высказыванием (и если существует хотя бы один человек, скажем Арнольд, который выше Стива, то его значение равно TRUE). Но есть и другой способ получить высказывание из исходного предиката: квантифицировать по обоим параметрам, например:

```
EXISTS x ( EXISTS y ( x выше, чем y ) )
```

Это суждение является высказыванием; оно принимает значение FALSE тогда и только тогда, когда не существует ни одного человека, который был бы выше любого другого, и TRUE в противном случае (подумайте над этим!).

Из этого примера можно извлечь несколько уроков:

- Чтобы получить высказывание из n-местного предиката с помощью одной лишь квантификации, необходимо выполнить квантификацию по каждому параметру. Более общо, результатом квантификации по m параметрам ($m \leq n$) является k-местный предикат, где $k = n - m$.
- Забудем ненадолго о кванторе всеобщности. Очевидно, что существуют два, на первый взгляд, различных высказывания, которые можно получить с помощью «квантификации по всему»:

```
EXISTS x ( EXISTS y ( x is taller than y ) )

EXISTS y ( EXISTS x ( x is taller than y ) )
```

  Однако должно быть понятно, что смысл этих высказываний одинаков: «Существуют два человека x и y такие, что x выше, чем y». Более общо, легко видеть, что последовательность одинаковых кванторов (только существования или только всеобщности) можно записать в любом порядке, не изменяя смысла высказывания. Напротив, для разных кванторов порядок важен (см. следующий пункт).
- Когда мы «квантифицируем по всему», каждый отдельный квантор может быть как квантором существования, так и квантором всеобщности. Поэтому в примере выше существует шесть различных высказываний, которые можно получить в результате полной квантификации; все они перечислены ниже. (На самом деле их восемь, но два можно игнорировать в силу замечания в предыдущем пункте.) В каждом случае я привел интерпретацию на естественном языке. Обратите внимание, что все интерпретации логически различаются! И еще заметьте, что в большинстве интерпретаций я был вынужден предположить, что во «вселенной» существует хотя бы два человека. Я вернусь к этому допущению в разделе «Еще о квантификации».

```
EXISTS x ( EXISTS y ( x выше, чем y ) )
```

  Семантика: Кто-то выше, чем кто-то еще; TRUE, если только все люди не одинакового роста.

```
EXISTS x ( FORALL y ( x выше, чем y ) )
```

  Семантика: Кто-то выше всех (включая и самого «кого-то»!); очевидно FALSE.

```
FORALL x ( EXISTS y ( x выше, чем y ) )
```

  Семантика: каждый выше кого-то; очевидно FALSE.

```
EXISTS y ( FORALL x ( x выше, чем y ) )
```

  Семантика: Кто-то ниже всех (включая и самого «кого-то»); очевидно FALSE. *Примечание*: Здесь я немного схитрил, потому что не сказал, что имеется в виду под «ниже»! Но я мог бы это сделать, явно объ-

явив, что предикаты «*x* выше, чем *y*» и «*y* ниже, чем *x*» логически эквивалентны, – и далее в этом разделе именно так и буду считать.

```
FORALL y ( EXISTS x ( x выше, чем y ) )
```

Семантика: Каждый ниже кого-то; очевидно FALSE.

```
FORALL x ( FORALL y ( x выше, чем y ) )
```

Семантика: Каждый выше кого-то; очевидно FALSE.

И последнее (рискуя быть обвиненным в том, что разжевываю очевидное): из того что пять из шести приведенных выше высказываний принимают одно и то же значение истинности FALSE, вовсе не следует, что все они обозначают одно и то же; на самом деле, никакие два не имеют одинаковой семантики.

Свободные и связанные переменные

То, что я называю параметрами, в формальной логике чаще называют *свободными переменными*, а квантификация по свободной переменной превращает ее в *связанную переменную*. Снова рассмотрим двуместный предикат из предыдущего раздела:

```
x выше, чем y
```

Здесь *x* и *y* – свободные переменные. Если теперь применить к *x* квантор существования, то получим:

```
EXISTS x ( x выше, чем y )
```

Теперь переменная *y* свободна (все еще), а переменная *x* связана. А в полностью квантифицированной формуле

```
EXISTS x EXISTS y ( x выше, чем y )
```

связаны и *x*, и *y*, а свободных переменных не осталось (предикат выродился в высказывание).

Мы знаем, что свободные переменные соответствуют параметрам в смысле, принятом в программировании. Но у связанных переменных в традиционном программировании вообще нет аналогов; они служат лишь для того, чтобы связать предикат внутри скобок с квантором вне скобок. Рассмотрим, к примеру, следующий простой предикат (на самом деле, высказывание):

```
EXISTS x ( x > 3 )
```

В этом высказывании утверждается, что существует целое число, большее трех. (Я предполагаю, что областью значений *x* является множество целых чисел. Позже я еще вернусь к этому моменту.) Поэтому смысл высказывания не изменится, если заменить оба вхождения *x* какой-нибудь другой переменной *y*. Иными словами, высказывание

```
EXISTS y ( y > 3 )
```

семантически идентично предыдущему.

Теперь рассмотрим такой предикат:

```
EXISTS x ( x > 3 ) AND x < 0
```

Здесь есть три вхождения *x*, но они означают разные вещи. Первые два связаны и могут быть заменены переменной *y* (например) без изменения семантики. Третья же переменная свободна и безнаказанно заменить ее не получится. Таким образом, из следующих двух предикатов первый эквивалентен показанному выше, а второй – нет:

```
EXISTS y ( y > 3 ) AND x < 0

EXISTS y ( y > 3 ) AND y < 0
```

Как видно из этого примера, термины «свободные и связанные переменные» обозначают не столько переменные, сколько *вхождения переменных*, точнее, вхождения ссылок на переменные в некоторый предикат. Так, во втором из приведенных выше предикатов второе *вхождение ссылки* на *y* связано, а третье вхождение свободно. Несмотря на это, обычно (и об этом можно лишь сожалеть) говорят о свободных и связанных переменных как таковых[1], пусть даже такое словоупотребление отдает легкой небрежностью. Будьте начеку и не путайтесь!

В завершение этого раздела отмечу, что теперь мы можем определить (переопределить, если хотите) высказывание как предикат, в котором все переменные связаны, или, что эквивалентно, нет ни одной свободной переменной.

Реляционное исчисление

Все сказанное в этой главе до сих пор имеет самое прямое отношение к реляционному исчислению. Рассмотрим простой пример – представление в реляционном исчислении запроса «Получить номера поставщиков и статусы поставщиков в Париже, которые поставляют деталь P2». Сначала для сравнения приведем алгебраическую формулировку:

```
( S WHERE CITY = 'Paris' ) { SNO , STATUS }
                           MATCHING ( SP WHERE PNO = 'P2' )
```

А вот эквивалентная формулировка в реляционном исчислении:

```
RANGEVAR SX  RANGES OVER S  ;
RANGEVAR SPX RANGES OVER SP ;

{ SX.SNO , SX.STATUS }
          WHERE SX.CITY = 'Paris' AND
                EXISTS SPX ( SPX.SNO = SX.SNO AND SPX.PNO = 'P2' )
```

[1] Иногда даже в учебниках по математической логике, где такую практику следует осудить.

Объяснение:

- В первых двух строках даются определения *переменных кортежа* (range variable) с множествами значений S и SP соответственно. Смысл этих определений в том, что в любой момент времени допустимыми значениями SX являются кортежи отношения, являющегося значением переменной-отношения S в этот момент времени. Аналогично допустимыми значениями SPX являются кортежи отношения, являющегося значением переменной-отношения SP в этот момент времени.
- Оставшиеся строки содержат сам запрос. Они имеют такую общую форму:

  прототип кортежа WHERE *предикат*

  Это аналог реляционного выражения (т. е. выражения, обозначающего отношение) в реляционном исчислении, и значением его является отношение, содержащее все возможные значения прототипа кортежа, для которых предикат принимает значение TRUE, и больше никаких кортежей не содержащее. (Термин *прототип кортежа* удачный, но не стандартный, однако стандартного термина для этого понятия, похоже, не существует). Таким образом, в нашем примере результатом является отношение степени 2, содержащее все пары (SNO,STATUS) из переменной-отношения S такие, что (а) городом является Париж и (б) существует поставка в переменной-отношении SP с тем же номером поставщика, что в этой паре, и номером детали P2.

Обратите внимание на имена с точками (и в прототипе кортежа, и в предикате); я не стану уделять этому много внимания, поскольку квалифицированные имена должны быть вам знакомы по SQL. На самом деле формулировка этого запроса в SQL очень напоминает показанную выше формулировку в реляционном исчислении:

```
SELECT SX.SNO , SX.STATUS
FROM   S AS SX
WHERE  SX.CITY = 'Paris'
AND    EXISTS
     ( SELECT *
       FROM   SP AS SPX
       WHERE  SPX.SNO = SX.SNO
       AND    SPX.PNO = 'P2' )
```

Отметим, что в SQL поддерживаются переменные кортежа, хотя сам термин там обычно не используется (в главе 12 я еще вернусь к теме переменных кортежа в SQL). Но важнее тот факт, что SQL поддерживает также квантор EXISTS; впрочем, эта поддержка не совсем прямая.[1]

[1] Наверное, будет полезно подчеркнуть, что EXISTS в SQL похож на конструкцию IS_NOT_EMPTY в **Tutorial D** (см. главу 3). См. раздел «Некоторые эквиваленции» ниже.

Конкретно, пусть *sq* – некоторый подзапрос, тогда EXISTS *sq* – булево выражение (и потому представляет собой предикат), которое принимает значение FALSE, если таблица, обозначенная подзапросом *sq*, пуста, и TRUE – в противном случае. (Табличное выражение *tx* в скобках, составляющее подзапрос *sq*, обычно, хотя и не всегда, имеет вид SELECT * FROM ... WHERE ..., а фраза WHERE опять обычно, но не обязательно, включает ссылку на некоторую «внешнюю» таблицу, то есть *sq* является *коррелированным подзапросом*. В примере выше такой внешней таблицей является S, а ссылка на нее производится с помощью переменной кортежа SX. Дополнительные пояснения см. в главе 12.)

Отступление

Здесь таится некая ирония. В главе 4 мы видели, что SQL в силу того, что поддерживает null-значения, основан на так называемой *трехзначной логике* – 3VL (а не на традиционной двузначной логике, которая рассматривается в этой главе и лежит в основе реляционной модели). В 3VL квантор существования может возвращать три разных результата: TRUE, FALSE и UNKNOWN (опять-таки см. главу 4). Но в SQL оператор EXISTS возвращает только TRUE или FALSE и никогда UNKNOWN. Например, EXISTS(*tx*) вернет TRUE, а не UNKNOWN, если значением *tx* окажется таблица, не содержащая ничего, кроме null-значений (тут я позволил себе некоторую вольность в словоупотреблении), хотя логически правильно было бы вернуть именно UNKNOWN. Следовательно, (a) EXISTS в SQL – не вполне корректная реализация квантора существования в 3VL и (b) SQL иногда возвращает неверные ответы (как мы уже неоднократно отмечали). Конкретный пример такой ситуации я приведу в следующей главе.

Рассмотрим еще один пример – запрос «Получить названия тех поставщиков, которые поставляют все детали». Я предполагаю, что уже определены те же переменные кортежа, что и выше, а также еще одна – PX:

```
RANGEVAR PX RANGES OVER P ;

{ SX.SNAME } WHERE FORALL PX ( EXISTS SPX ( SPX.SNO = SX.SNO AND
                                            SPX.PNO = PX.PNO ) )
```

На естественном языке это, несколько коряво, звучит так: «Получить названия поставщиков таких, что для всех деталей существует поставка с таким же номером поставщика, как у поставщика, и с таким же номером детали, как у детали». *Примечание*: Как вы, наверное, знаете, в SQL вообще нет прямой поддержки квантора FORALL. Поэтому я сейчас не покажу SQL-аналог этого примера, но вернусь к этому вопросу позже, в разделе «Еще о квантификации». Однако я все же отмечу логическое различие между приведенным выше выражением реляционного исчисления и следующим, в котором кванторы поменяны местами:

```
{ SX.SNAME } WHERE EXISTS SPX ( FORALL PX ( SPX.SNO = SX.SNO AND
                                            SPX.PNO = PX.PNO ) )
```

Упражнение: Что означает это выражение? И как вы думаете, «разумен» ли этот запрос?

Еще один пример («Получить названия тех поставщиков, которые поставляют все красные детали»):

```
{ SX.SNAME } WHERE FORALL PX ( IF PX.COLOR = 'Red' THEN
                               EXISTS SPX ( SPX.SNO = SX.SNO AND
                                            SPX.PNO = PX.PNO ) )
```

Обратите внимание на логическую импликацию в этом примере. Отмечу еще, что я предполагаю то же самое множество значений переменных, что и раньше; собственно, это предположение будет действовать до конца главы.

Кстати, предыдущий запрос можно сформулировать и по-другому:

```
{ SX.SNAME } WHERE FORALL PX
                 ( EXISTS SPX ( IF PX.COLOR = 'Red' THEN
                                   SPX.SNO = SX.SNO AND
                                   SPX.PNO = PX.PNO ) )
```

В этой формулировке предикат во фразе WHERE записан в так называемой предваренной нормальной форме. Неформально говоря, это означает, что все кванторы находятся в начале. Вот точное определение:

> **Определение**: Говорят, что предикат записан в *предваренной нормальной форме* (PNF), если либо (а) он не содержит кванторов вовсе, либо (б) имеет вид EXISTS *x* (*p*) или FORALL *x* (*p*), где предикат *p* записан в предваренной нормальной форме. Иными словами, PNF-предикат имеет вид:
>
> ```
> Q1 x1 ( Q2 x2 ( ... ( Qn xn ( q ) ) ... ) )
> ```
>
> где $n \geq 0$, все *Qi* ($i = 1, 2, \ldots, n$) – либо EXISTS, либо FORALL, и предикат *q* (который иногда называют *матричным*) не содержит кванторов.

Предваренная нормальная форма не более и не менее правильна, чем любая другая, но при некоторой практике она во многих случаях дает наиболее естественную формулировку.

Еще о переменных кортежа

Из сказанного выше должно быть ясно, что переменные кортежа в реляционном исчислении играют роль свободных и связанных переменных в смысле формальной логики. Я уже упоминал, что для таких переменных всегда должно быть определено некое множество допустимых значений; в контексте реляционного исчисления это множество всегда является телом некоторого отношения (обычно, хотя и необязательно, отношения, которое является текущим значением какой-то пе-

ременной-отношения). *Примечание*: Отсюда следует, что данная переменная кортежа всегда обозначает некоторый кортеж. Поэтому реляционное исчисление иногда называют *исчислением кортежей*, а вместо термина *range variable* употребляют *tuple variable*. Но последнее может приводить к путанице, так как термин *tuple variable* (переменная кортежа) уже имеет несколько иное значение.[1]

Теперь я могу сказать еще несколько слов о синтаксисе выражений реляционного исчисления.

- Прежде всего, прототип кортежа – это заключенный в круглые скобки список элементов, разделенных запятыми, в котором каждый элемент является либо *ссылкой на атрибут кортежа* (возможно, в сочетании с ключевым словом AS, которое вводит новое имя атрибута), *либо ссылкой на переменную кортежа*. (Существуют и другие возможности, но пока я ограничусь только этими случаями. См. пример 5 ниже.) *Примечание*: принято опускать скобки, если список состоит всего из одного элемента, но я для ясности буду включать их даже тогда, когда без этого можно было бы обойтись.
- Ссылка на атрибут кортежа – это выражение вида *R.A*, где *A* – атрибут отношения, являющегося множеством значений переменной кортежа *R* (в примере выше SX.SNO). А ссылка на переменную кортежа – это просто имя переменной кортежа, например SX, которое есть не что иное, как сокращенная запись списка ссылок на атрибуты кортежа, по одной для каждого атрибута отношения, являющегося множеством значений переменной кортежа.
- Пусть в прототипе кортежа встречается некая ссылка на атрибут кортежа, в которой участвует, явно или неявно, переменная кортежа *R*. Тогда предикат в соответствующей фразе WHERE может содержать и обычно содержит по крайней мере одну *свободную ссылку на атрибут кортежа, в которой участвует R*, где под выделенной курсивом фразой понимается ссылка на атрибут кортежа вида *R.A*, который не находится в области действия какого-либо квантора, в котором *R* является связанной переменной.
- Фраза WHERE необязательна; если она опущена, подразумевается WHERE TRUE.

---

[1] Тем не менее в переводе мы остановились на термине «переменная кортежа» отчасти потому, что в русской терминологии не нашлось адекватного перевода range variable (предлагались разные варианты: от совершенно неприемлемой «ранжированной переменной» до не слишком вразумительной и длинной «переменной с множеством значений»), а отчасти потому, что именно так этот термин переведен в издании книги К. Дж. Дейта «Введение в системы баз данных». – Пер. с англ. – СПб.: Вильямс, 2001. – *Прим. перев.*

Дополнительные примеры запросов

Я приведу еще несколько примеров запросов реляционного исчисления, чтобы проиллюстрировать ряд особенностей; однако я не ставлю своей целью дать исчерпывающее изложение вопроса. Для простоты я опускаю определения RANGEVAR, которые на практике необходимы, и предполагаю, что SX, SY, SZ и т. д. определены как переменные кортежа над множеством значений S; PX, PY, PZ и т. д. – переменные кортежа над множеством значений P; SPX, SPY, SPZ, и т. д. – переменные кортежа над множеством значений SP. Обращаю ваше внимание, что в общем случае приведенные ниже формулировки не являются единственно возможными. Оставляю в качестве упражнения для читателя формулирование эквивалентных запросов на SQL.

Пример 1. Получить все пары номеров поставщиков такие, что оба поставщика находятся в одном городе.

```
{ SX.SNO AS SA , SY.SNO AS SB } WHERE SX.CITY = SY.CITY
                                  AND SX.SNO < SY.SNO
```

Обратите внимание на использование фраз AS в этом примере.

Пример 2. Получить названия поставщиков, которые поставляют хотя бы одну красную деталь.

```
{ SX.SNAME } WHERE EXISTS SPX ( EXISTS PX ( SX.SNO = SPX.SNO AND
                                            SPX.PNO = PX.PNO AND
                                            PX.COLOR = 'Red' ) )
```

Пример 3. Получить названия поставщиков, которые поставляют хотя бы одну деталь, поставляемую поставщиком S2.

```
{ SX.SNAME } WHERE EXISTS SPX ( EXISTS SPY ( SX.SNO = SPX.SNO AND
                                             SPX.PNO = SPY.PNO AND
                                             SPY.SNO = 'S2' ) )
```

Пример 4. Получить названия поставщиков, которые не поставляют деталь P2.

```
{ SX.SNAME } WHERE NOT EXISTS SPX ( SPX.SNO = SX.SNO AND
                                    SPX.PNO = 'P2' )
```

На практике выражение, следующее за NOT, иногда приходится заключать в скобки.

Пример 5. Для каждой поставки получить всю информацию о поставке, а также ее общий вес.

```
{ SPX , PX.WEIGHT * SPX.QTY AS SHIPWT } WHERE PX.PNO = SPX.PNO
```

Обратите внимание на математическое выражение в прототипе кортежа. В алгебраическом варианте этого примера следовало бы воспользоваться оператором EXTEND.

Пример 6. Для каждой детали получить номер детали и общее поставленное количество.

```
{ PX.PNO , SUM ( SPX WHERE SPX.PNO = PX.PNO , QTY ) AS TOTQ }
```

В этом примере иллюстрируется вызов агрегатного оператора в прототипе кортежа (это также первый пример, в котором опущена фраза WHERE). Попутно отмечу, что следующее выражение, хотя оно синтаксически допустимо, не является правильной формулировкой того же запроса (почему?):

```
{ PX.PNO , SUM ( SPX.QTY WHERE SPX.PNO = PX.PNO ) AS TOTQ }
```

Ответ: Потому что перед вычислением суммы устраняются дубликаты, то есть одинаковые значения количества деталей в поставке.

Пример 7. Получить города, в которых находится более пяти красных деталей.

```
{ PX.CITY } WHERE
     COUNT ( PY WHERE PY.CITY = PX.CITY AND PY.COLOR = 'Red' ) > 5
```

Примеры ограничений

Теперь я хотел бы на примерах продемонстрировать применение реляционного исчисления для формулировки ограничений. Примеры пронумерованы так же, как в главе 8, и описывают те же самые ситуации. Предполагается, что переменные кортежа определены, как в предыдущем разделе. Еще раз замечу, что приведенные формулировки, вообще говоря, не являются единственно возможными.

Пример 1. Значения статуса должны принадлежать диапазону от 1 до 100 включительно.

```
CONSTRAINT CX1 FORALL SX ( SX.STATUS > 0 AND SX.STATUS < 101 ) ;
```

Примечание

SQL позволяет упрощать подобные ограничения за счет исключения явного использования переменной кортежа и, что важнее, явного квантора всеобщности. Точнее, мы можем следующим образом задать ограничение базовой таблицы (см. главу 8) в самом определении базовой таблицы S:

```
CONSTRAINT CX1 CHECK ( STATUS > 0 AND STATUS < 101 )
```

Аналогичное замечание применимо и к последующим примерам.

Пример 2. Поставщики из Лондона должны иметь статус 20.

```
CONSTRAINT CX2 FORALL SX ( IF SX.CITY = 'London'
                           THEN SX.STATUS = 20 ) ;
```

Пример 3. Ни в каких двух кортежах переменной-отношения S не должно быть одинакового номера поставщика.

```
CONSTRAINT CX3 FORALL SX ( FORALL SY ( IF SX.SNO = SY.SNO THEN
                                          SX.SNAME = SY.SNAME AND
                                          SX.STATUS = SY.STATUS AND
                                          SX.CITY = SY.CITY ) ) ;
```

Пример 4. Если в двух кортежах переменной-отношения S одинаковы номера поставщиков, то должны быть одинаковы и города.

```
CONSTRAINT CX4 FORALL SX ( FORALL SY ( IF SX.SNO = SY.SNO
                                       THEN SX.CITY = SY.CITY ) ) ;
```

Пример 5. Ни один поставщик со статусом меньше 20 не может поставлять деталь P6.

```
CONSTRAINT CX5 FORALL SX ( IF SX.STATUS < 20 THEN
                           NOT EXISTS SPX ( SPX.SNO = SX.SNO AND
                                            SPX.PNO = 'P6' ) ) ;
```

Пример 6. Любой номер поставщика, встречающийся в переменной-отношении SP, должен встречаться и в переменной-отношении S.

```
CONSTRAINT CX6 FORALL SPX ( EXISTS SX ( SX.SNO = SPX.SNO ) ) ;
```

Об этом примере у меня еще будет случай кое-что сказать в следующем разделе.

Пример 7. Ни один номер поставщика не может встречаться в обеих переменных-отношениях LS и NLS.

```
CONSTRAINT CX7 FORALL LX ( FORALL NX ( LX.SNO ≠ NX.SNO ) ) ;
```

Множествами значений LX и NX являются LS и NLS соответственно.

Пример 8. Поставщик S1 и деталь P1 не могут находиться в разных городах.

```
CONSTRAINT CX8 NOT EXISTS SX ( EXISTS PX ( SX.SNO = 'S1' AND
                                           PX.PNO = 'P1' AND
                                           SX.CITY ≠ PX.CITY ) ) ;
```

Пример 9. Должен существовать хотя бы один поставщик. (В главе 8 нет соответствующего примера.)

```
CONSTRAINT CX9 EXISTS SX ( TRUE ) ;
```

Выражение EXISTS SX (TRUE) принимает значение FALSE тогда и только тогда, когда множеством значений SX является пустое отношение.

Еще о квантификации

Я бы хотел обсудить еще ряд вопросов, относящихся к квантификации.

Нам не нужны оба квантора

На практике нам не нужны оба квантора EXISTS и FORALL, поскольку любой предикат, выражаемый в терминах EXISTS, можно выразить

в терминах FORALL и наоборот. В качестве примера еще раз рассмотрим следующий предикат:

```
EXISTS x ( x выше, чем Стив )
```

(«Кто-то выше, чем Стив»). По-другому то же самое можно сказать следующим образом:

```
NOT ( FORALL x ( NOT ( x выше, чем Steve ) ) )
```

(«Неверно, что нет никого выше Стива»). Более общо, предикат

```
EXISTS x ( p ( x ) )
```

логически эквивалентен предикату

```
NOT ( FORALL x ( NOT ( p ( x ) ) )
```

(где предикат *p* может иметь и другие параметры, помимо *x*). Аналогично предикат

```
FORALL x ( p ( x ) )
```

логически эквивалентен предикату

```
NOT ( EXISTS x ( NOT ( p ( x ) ) )
```

(где опять-таки предикат *p* может иметь и другие параметры, помимо *x*).

Отсюда следует, что формальный язык не обязан явно поддерживать оба квантора – EXISTS и FORALL. Но на практике их поддержка крайне желательна. Причина в том, что некоторые задачи «более естественно» формулируются в терминах EXISTS, тогда как другие – в терминах FORALL. Например, как мы знаем, SQL поддерживает EXISTS, но не FORALL; поэтому формулировка некоторых запросов выглядит в SQL неуклюже. Рассмотрим еще раз запрос «Получить поставщиков, которые поставляют все детали», который в реляционном исчислении выражается совсем просто:

```
{ SX } WHERE FORALL PX ( EXISTS SPX
                         ( SPX.SNO = SX.SNO AND SPX.PNO = PX.PNO ) )
```

А в SQL этот запрос приходится записывать следующим образом:

```
SELECT SX.*
FROM   S AS SX
WHERE  NOT EXISTS
     ( SELECT *
       FROM   P AS PX
       WHERE  NOT EXISTS
            ( SELECT *
              FROM   SP AS SPX
              WHERE  SX.SNO = SPX.SNO
              AND    SPX.PNO = PX.PNO ) )
```

(«Получить поставщиков SX таких, что не существует детали PX такой, что не существует поставки SPX, связывающей поставщика SX с дета-

лью РХ»). Уже одно отрицание – это плохо (многие пользователи испытывают при этом трудности), а уж двойное отрицание, как в этом запросе, многократно хуже.

Пустые множества значений

Снова обратимся к тому факту, что предикаты

```
EXISTS x ( p ( x ) )
```

и

```
NOT ( FORALL x ( NOT ( p ( x ) ) ) )
```

логически эквивалентны. Как мы знаем, связанная переменная *x* в каждом из этих предикатов должна принимать значения из некоторого множества допустимых значений. Предположим теперь, что это множество пусто; например, оно может состоять из людей ростом выше пятнадцати метров (в контексте баз данных было бы уместнее говорить о множестве кортежей переменной-отношения, которое в данный момент пусто). Тогда:

- Выражение EXISTS *x* (*p*(*x*)) принимает значение FALSE, потому что «не существует ни одного *x*», то есть не существует ни одного значения, которое можно было бы подставить вместо *x*, чтобы сделать это выражение истинным. Отмечу, что это замечание справедливо *вне зависимости от того, что представляет собой p(x)*. Например, высказывание «существует человек ростом выше пятнадцати метров, работающий в IBM» ложно (что и не удивительно).
- Отсюда следует, что отрицание NOT EXISTS *x* (*p*(*x*)) принимает значение TRUE, –опять-таки вне зависимости от того, что представляет собой *p*(*x*). Например, высказывание «не существует человека ростом выше пятнадцати метров, работающего в IBM» (или в более разговорной форме «в IBM не работают люди ростом выше пятнадцати метров») истинно (и снова не удивительно).
- Но выражение NOT EXISTS *x* (*p*(*x*)) эквивалентно FORALL *x* (NOT (*p*(*x*)), и, значит, это последнее тоже принимает значение TRUE – и снова вне зависимости от природы *p*(*x*).
- Но если предикат *p*(*x*) произволен, то произволен и предикат NOT (*p*(*x*)). Поэтому мы получаем следующий, быть может, удивительный результат; суждение FORALL *x* (...) принимает значение TRUE, если никаких допустимых *x* не существует, *вне зависимости от того, что находится в скобках*. Например, суждение «Все люди ростом выше пятнадцати метров работают в IBM» также истинно, потому что, повторюсь, людей выше пятнадцати метров не существует.

Следствием всего вышеизложенного является тот факт, что некоторые запросы могут порождать неожиданные результаты (для тех, кто незнаком с формальной логикой). Например, рассмотренный выше запрос

```
{ SX } WHERE FORALL PX ( EXISTS SPX ( SPX.SNO = SX.SNO AND
                                     SPX.PNO = PX.PNO ) )
```

(«Получить поставщиков, которые поставляют все детали») вернет всех поставщиков, если не одной детали не существует.

Определение EXISTS и FORALL

Как вы, наверное, поняли, кванторы EXISTS и FORALL можно определить в терминах *повторяющихся* OR и AND соответственно. Сначала рассмотрим EXISTS. Пусть $p(x)$ – предикат с параметром x, и пусть x пробегает множество значений $X = \{x1, x2, \ldots, xn\}$. Тогда

```
EXISTS x ( p ( x ) )
```

является предикатом, который, по определению, эквивалентен (и потому является сокращенной записью) предикату

```
p ( x1 ) OR p ( x2 ) OR ... OR p ( xn ) OR FALSE
```

Обратите внимание, что это выражение обращается в FALSE, если множество X пусто (как мы уже знаем). Например, пусть $p(x)$ имеет вид «у x есть спутник», и пусть X – множество {Меркурий, Венера, Земля, Марс}. Тогда предикат EXISTS x ($p(x)$) принимает вид «EXISTS x (у x есть спутник)» и является сокращенной записью для

```
( у Меркурия есть спутник ) OR ( у Венеры есть спутник ) OR
( у Земли есть спутник )    OR ( у Марса есть спутник  ) OR FALSE
```

и равен TRUE, например, потому что высказывание «у Марса есть спутник» истинно. Аналогично

```
FORALL x ( p ( x ) )
```

является предикатом, который, по определению, эквивалентен (и потому является сокращенной записью) предикату

```
p ( x1 ) AND p ( x2 ) AND ... AND p ( xn ) AND TRUE
```

И это выражение (как мы уже знаем) обращается в TRUE, если множество X пусто. Например, пусть $p(x)$ и X такие же, как определено выше для квантора EXISTS. Тогда предикат FORALL x ($p(x)$) принимает вид «FORALL x (у x есть спутник)» и является сокращенной записью для

```
( у Меркурия есть спутник ) AND ( у Венеры есть спутник ) AND
( у Земли есть спутник )    AND ( у Марса есть спутник  ) AND TRUE
```

Это выражение принимает значение FALSE, например, потому что высказывание «у Венеры есть спутник» ложно.

Попутно отмечу, что, как видно из этих примеров, возможность определения EXISTS и FORALL в терминах повторяющихся OR и AND, соответственно, означает, что всякий предикат с кванторами эквивалентен некоторому предикату без кванторов. Но тогда возникает не лишен-

ный основания вопрос, а зачем вообще нужна вся эта возня с кванторами? Ответ таков: мы можем определить EXISTS и FORALL в терминах повторяющихся OR и AND только потому, что *множества, с которыми нам приходится иметь дело, к счастью, конечны* (так как мы работаем с компьютерами, а те по природе своей конечны). В чистой логике, где такого ограничения нет, эти определения уже не действуют.[1]

Быть может, следует добавить, что пусть даже мы всегда имеем дело с конечными множествами, а EXISTS и FORALL – просто сокращения, но зато какие полезные сокращения! Лично я точно не хотел бы формулировать запросы исключительно в терминах AND и OR, не прибегая к кванторам. А если спуститься с небес на землю, то кванторы позволяют формулировать запросы, не зная точного содержимого базы данных в каждый момент времени (что было бы невозможно в случае явного повторения OR или AND).

Другие виды кванторов

Хотя кванторы EXISTS и FORALL, безусловно, самые важные на практике, кроме них есть и другие. Нет никаких причин, запрещающих, например, кванторы вида

существуют по меньшей мере три x, такие что

или

большинство x таковы, что

или

нечетное количество x таковы, что

(и так далее). Один довольно важный частный случай – *существует ровно одно x, такое что*. Для него я буду использовать ключевое слово UNIQUE. Вот несколько примеров:

```
UNIQUE x ( x выше, чем Арнольд )
```

Семантика: существует ровно один человек, который выше, чем Арнольд; скорее всего, FALSE.

```
UNIQUE x ( x имеет номер социального страхования y )
```

---

[1] Поясню: рассмотрим, к примеру, высказывание EXISTS *x* (*p*), где *p* – предикат с единственным параметром *x*. Если *x* пробегает бесконечное множество, то любая попытка сформулировать алгоритм «с повторяющимися OR» для вычисления значения такого высказывания обречена на провал, так как вычисление может никогда не закончиться (не найдется такое значение *x*, которое удовлетворяло бы *p*). Аналогично любая попытка сформулировать алгоритм «с повторяющимися AND» для FORALL *x* (*p*) также обречена на провал, так как вычисление может никогда не закончиться (не найдется такое значение *x*, которое не удовлетворяло бы *p*).

Семантика: Ровно один человек имеет номер социального страхования *y* (*y* – параметр). Мы не можем сопоставить этому суждению значение истинности, потому что это (одноместный) предикат, а не высказывание.

```
FORALL y ( UNIQUE x ( x имеет номер социального страхования y ) )
```

Семантика: Каждый человек имеет уникальный номер социального страхования. (Я предполагаю, что *y* пробегает множество всех реально присвоенных, а не потенциально возможных номеров социального страхования. *Упражнение*: Верно ли, что этот предикат, а фактически высказывание, принимает значение TRUE?)

Еще одно упражнение: что означает следующий предикат?

```
FORALL x ( UNIQUE y ( x имеет номер социального страхования y ) )
```

Теперь вспомним следующее ограничение: «Любой номер поставщика, встречающийся в переменной-отношении SP, должен встречаться в переменной-отношении S». Вот как я формулировал его раньше:

```
CONSTRAINT CX6 FORALL SPX ( EXISTS SX ( SX.SNO = SPX.SNO ) ) ;
```

Но я полагаю, вы согласитесь, что более точна такая формулировка:

```
CONSTRAINT CX6 FORALL SPX ( UNIQUE SX ( SX.SNO = SPX.SNO ) ) ;
```

Другими словами, мы хотим, чтобы для любого кортежа переменной-отношения SP существовал не хотя бы один (EXISTS), а ровно один (UNIQUE) соответствующий ему кортеж переменной-отношения S. Предыдущая формулировка «работает», потому что имеется дополнительное ограничение, состоящее в том, что {SNO} – ключ переменной-отношения S. Но пересмотренная формулировка ближе к тому, что мы хотим сказать в действительности.

SQL поддерживает UNIQUE (в какой-то мере), хотя поддержка и не такая прямая, как для EXISTS. Точнее, пусть *sq* – подзапрос; тогда UNIQUE *sq* – булево выражение, которое принимает значение FALSE, если таблица, обозначенная *sq*, содержит строки-дубликаты, и TRUE в противном случае. Таким образом, если логическое выражение

```
UNIQUE x ( p ( x ) )
```

означает «Существует ровно одно значение аргумента *a*, соответствующее параметру *x*, при котором *p*(*a*) равно TRUE», то его SQL-аналог (очень приблизительный!)

```
UNIQUE ( SELECT * FROM T WHERE p ( x ) )
```

означает «При данном значении аргумента *a*, соответствующего параметру *x*, в таблице *T* существует не более одной строки такой, что *p*(*a*) равно TRUE». В частности, если выражение, являющееся аргументом UNIQUE, – не произвольный, а однострочный подзапрос (см. главу 12), то вызов UNIQUE возвращает TRUE, если обозначенная этим подзапросом таблица содержит только одну *или не содержит ни одной строки*.

Вот несколько натянутый пример SQL-запроса с ключевым словом UNIQUE («Получить названия поставщиков, которые поставляют по меньшей мере две различных детали в одном и том же количестве»):

```
SELECT DISTINCT SNAME
FROM   S
WHERE  NOT UNIQUE ( SELECT QTY
                    FROM   SP
                    WHERE  SP.SNO = S.SNO )
```

Повторю, что это натянутый пример; точнее, в нем используется тот факт, что SQL сохраняет дубликаты в результате выражения SELECT, если отсутствует DISTINCT.[1] Однако, как вы помните, в главе 4 я говорил, что слово DISTINCT следует употреблять «всегда». Поэтому, пусть даже этот пример близок к реальности, не делайте отсюда вывод, что система должна допускать строки-дубликаты во имя поддержки таких запросов. Вот формулировка того же запроса, в которой UNIQUE не используется:

```
SELECT DISTINCT SNAME
FROM   S
WHERE  ( SELECT COUNT ( DISTINCT QTY )
         FROM   SP
         WHERE  SP.SNO = S.SNO ) <
       ( SELECT COUNT ( * )
         FROM   SP
         WHERE  SP.SNO = S.SNO )
```

В SQL ключевое слово UNIQUE применяется также в определениях ключа. Например, в предложении CREATE TABLE для таблицы S есть такая спецификация:

```
UNIQUE ( SNO )
```

Но на самом деле эта спецификация, по определению, является сокращенной записью следующей (которая могла бы быть частью более общего ограничения таблицы или предложения CREATE ASSERTION):

```
CHECK ( UNIQUE ( SELECT SNO FROM S ) )
```

Заметим, что выражение SELECT в этом ограничении CHECK точно не должно содержать ключевого слова DISTINCT! (Почему?)

Кроме того, ключевое слово UNIQUE используется в SQL и в выражениях MATCH. Вот пример («Получить поставщиков, которые поставляют ровно одну деталь»):

```
SELECT SNO , SNAME , STATUS , CITY
```

[1] Это также первый пример в этой главе, где используются неявные переменные кортежа (хотя в предыдущих главах таких примеров было сколько угодно). Если вам требуется формальное объяснение происходящего, см. главу 12.

```
FROM   S
WHERE  SNO MATCH UNIQUE ( SELECT SNO FROM SP )
```

Однако и это не более чем сокращенная запись. Так, только что приведенный пример эквивалентен следующему:

```
SELECT SNO , SNAME , STATUS , CITY
FROM   S
WHERE  EXISTS ( SELECT *  /* существует по меньшей мере один ... */
                FROM   SP
                WHERE  SP.SNO = S.SNO )
AND    UNIQUE ( SELECT *    /* ... и не существует двух          */
                FROM   SP
                WHERE  SP.SNO = S.SNO )
```

который, в свою очередь, эквивалентен такому:

```
SELECT SNO , SNAME , STATUS , CITY
FROM   S
WHERE  ( SELECT COUNT ( * )
         FROM   SP
         WHERE  SP.SNO = S.SNO ) = 1
```

Некоторые эквиваленции

Я закончу эту главу несколькими замечаниями об эквиваленциях, которые, возможно, вам уже приходили в голову. Прежде всего, вспомним оператор IS_EMPTY, с которым мы познакомились в главе 3 и активно использовали в главе 8. Если система поддерживает этот оператор, то отпадает логическая необходимость в поддержке кванторов благодаря следующим эквиваленциям:

```
EXISTS x ( p )  ≡  NOT ( IS_EMPTY ( X WHERE p ) )
```

и

```
FORALL x ( p )  ≡  IS_EMPTY ( X WHERE NOT ( p ) )
```

(Я предполагаю, что переменная x пробегает множество X.)

На самом деле, имеющаяся в SQL поддержка EXISTS (и FORALL в том виде, в котором она существует) основана именно на этих эквиваленциях. Факт в том, что EXISTS в SQL вообще не является квантором как таковым, потому что отсутствуют связанные переменные. Это оператор в общепринятом смысле слова: одноместный оператор типа BOOLEAN, если быть точным. Как и для любого одноместного оператора, вызов оператора EXISTS состоит из двух этапов: сначала вычисляется выражение, обозначающее его единственный аргумент, а потом сам оператор применяется к результату вычисления. Таким образом, видя выражение EXISTS (tx), где tx – табличное выражение, система сначала вычисляет tx, чтобы получить таблицу t, а затем применяет к t оператор EXISTS, который возвращает TRUE, если t не пуста, и FALSE в против-

ном случае. (Так, по крайней мере, алгоритм выглядит концептуально; возможны многочисленные оптимизации, но они к теме данного обсуждения не имеют отношения.)

А теперь я объясню, почему SQL не поддерживает квантор FORALL. Причина состоит в том, что представление квантора всеобщности с помощью оператора, имеющего синтаксис FORALL(*tx*), где *tx* – табличное выражение, не имеет никакого смысла. Рассмотрим, к примеру, гипотетическое выражение FORALL (SELECT * FROM S WHERE CITY = 'Paris'). Что оно могло бы означать? Очевидно, что оно не могло бы означать «Все поставщики в Париже», потому что – говоря нестрого – аргумент, к которому применяется этот гипотетический оператор, – не все поставщики вообще, а только поставщики в Париже.

Но на самом деле нам вообще не нужны кванторы, если система поддерживает агрегатный оператор COUNT, благодаря следующим эквиваленциям:

- `EXISTS` *x* `(` *p* `)` ≡ `COUNT (` *X* `WHERE` *p* `) > 0`
- `FORALL` *x* `(` *p* `)` ≡ `COUNT (` *X* `WHERE` *p* `) = COUNT (` *X* `)`
- `UNIQUE` *x* `(` *p* `)` ≡ `COUNT (` *X* `WHERE` *p* `) = 1`

Конечно, я не призываю заменять все выражения с кванторами выражениями, содержащими вызовы COUNT, – хотя иногда приходится, если мы хотим оставаться в рамках чистой алгебры, – но было бы неправильно с моей стороны не упомянуть о такой возможности.

Отступление

В этой книге речь редко заходит о производительности, но я должен хотя бы отметить, что приведенные выше эквиваленции могут порождать проблемы с производительностью. Рассмотрим, к примеру, следующее выражение, которое выражает на SQL запрос «Получить поставщиков, которые поставляют по меньшей мере одну деталь»:

```
SELECT *
FROM   S
WHERE  EXISTS
     ( SELECT *
       FROM   SP
       WHERE  SP.SNO = S.SNO )
```

Вот другая формулировка, логически эквивалентная предыдущей:

```
SELECT *
FROM   S
WHERE  ( SELECT COUNT ( * )
         FROM   SP
         WHERE  SP.SNO = S.SNO ) > 0
```

Но нам совершенно ни к чему, чтобы система вычисляла истинное значение счетчика, который здесь упоминается, только чтобы потом проверить, боль-

ше он единицы или нет; мы хотим, чтобы вычисление прекратилось, как только будет найдена вторая поставка. Другими словами, нам нужна некоторая оптимизация. Но писать код, который зависит от наличия такой оптимизации, было бы неразумно! Поэтому относитесь к оператору COUNT с осторожностью и не используйте его там, где логически правильнее использовать EXISTS.

Реляционная полнота

У каждого оператора реляционной алгебры есть точное определение в терминах формальной логики. (Я не останавливался на этом раньше, но легко видеть, что данные в главах 6 и 7 определения соединения и других операторов легко переформулировать в терминах логики, как описано в этой главе.) Отсюда сразу следует, что для каждого выражения реляционной алгебры существует логически эквивалентное (то есть обладающее той же семантикой) ему выражение реляционного исчисления. Иными словами, реляционное исчисление по меньшей мере столь же «мощно» (или, точнее, столь же *выразительно*), сколь и реляционная алгебра: любая задача, выразимая средствами алгебры, может быть выражена и средствами исчисления.

Обратное также верно, хотя это и не столь очевидно, – реляционная алгебра по меньшей мере столь же выразительна, сколь и реляционное исчисление. Другими словами, оба формализма логически эквивалентны, и оба, как говорят, *реляционно полные*. В приложении А я еще скажу несколько слов об этом понятии.

Важность непротиворечивости

Мне необходимо поставить точку в одном незаконченном деле. Если помните, в главе 8 я заявил, что в противоречивой системе можно доказать истинность любого высказывания (даже очевидно ложного). Поговорим об этом утверждении более подробно.

Начну с простого примера. Предположим, что (а) переменная-отношение S в данный момент не пуста; (б) имеется ограничение, требующее, чтобы всегда существовала хотя бы одна деталь, но в данный момент переменная-отношение P пуста (налицо противоречие). Теперь рассмотрим такой запрос реляционного исчисления:

```
SX WHERE EXISTS PX ( TRUE )
```

Или, если вы предпочитаете SQL:

```
SELECT *
FROM   S
WHERE  EXISTS
     ( SELECT *
       FROM   P )
```

Если вычислять это выражение непосредственно, то получится пустой результат. Альтернативно, если система (или пользователь) замечает, что существует ограничение, которое говорит, что EXISTS PX (TRUE) всегда должно иметь значение TRUE, то фразу WHERE можно свести просто к WHERE TRUE, и тогда результатом будет все отношение «поставщики». По крайней мере один из этих результатов обязан быть ложным! Фактически в некотором смысле ложны оба; в противоречивой базе данных просто не существует – и не может существовать – корректно определенного понятия правильности, и любой ответ столь же хорош (или плох), как любой другой. Собственно, это и так очевидно: если я говорю вам, что высказывание *p* одновременно истинно и ложно, а затем спрашиваю, истинно ли *p*, то вы просто не можете дать правильный ответ.

Если я вас еще не убедил, давайте рассмотрим следующий чуть более реалистичный пример на SQL (в тех же предположениях):

```
SELECT DISTINCT
       CASE WHEN EXISTS ( SELECT * FROM P ) THEN x ELSE y END
FROM   S
```

Это выражение вернет либо *x*, либо *y* – точнее, таблицу, содержащую либо *x*, либо *y* – в зависимости от того, будет или не будет вызов EXISTS заменен простым TRUE. А теперь примите во внимание, что *x* и *y* могут быть чем угодно... Например, *x* может быть SQL-выражением, обозначающим суммарный вес всех деталей, а *y* – литералом 0. И в таком случае выполнение этого запроса легко может привести к неверному выводу о том, суммарный вес деталей равен null, а не 0.

Заключительные замечания

Я глубоко убежден, что профессионалы в области баз данных вообще и специалисты, применяющие SQL на практике, в частности, должны быть знакомы хотя бы с основами логики предикатов (или реляционного исчисления – что в конечном итоге одно и то же). Я хотел бы завершить эту главу обоснованием своей точки зрения.

Моя основная мысль заключается в том, что знание формальной логики приучает точно мыслить (а в нашей области точность мышления имеет особенно большое значение). В частности, оно заставляет ценить значимость надлежащей квантификации. Естественный язык часто бывает неточен, но тщательный анализ того, какая именно квантификация нужна, позволяет ухватить смысл высказывания, которое на естественном языке звучало бы очень расплывчато. Например, можете поразмыслить над тем, что хотел сказать Авраам Линкольн – или мог бы хотеть, или думал, что хочет сказать, или мог бы думать, что хочет сказать – в своем знаменитом афоризме: «Можно всё время дурачить

некоторых, можно некоторое время дурачить всех, но нельзя всё время дурачить всех».

Я прекрасно осознаю, что многие со мной не согласятся, то есть найдется немало людей, считающих, что простым смертным нет необходимости вгрызаться в такой неподатливый предмет, каким может показаться формальная логика. По сути дела, они заявляют, что логика слишком трудна для большинства людей. Возможно, в целом это и так (логика – обширный предмет). Но для наших узких целей вовсе не обязательно разбираться во всей формальной логике; я даже сомневаюсь, что потребуется существенно больше того, что уже изложено в этой главе. Но сколь велика награда! Я уже высказывал эту мысль в другой своей книге «*Logic and Databases: The Roots of Relational Theory*» (Trafford, 2007) и хотел бы процитировать слова из нее:

> Безусловно, имеет смысл с самого начала потратить определенные усилия на ознакомление с материалом, изложенным в этой главе, чтобы впоследствии избежать проблем с неоднозначно трактуемыми бизнес-правилами. Неоднозначность бизнес-правил ведет к задержкам реализации в лучшем случае и к ошибкам реализации – в худшем (или к тому и другому сразу). А такие задержки и ошибки, конечно, стоят денег, возможно, куда больших, чем пришлось бы потратить на начальное обучение. Другими словами, правильное оформление бизнес-правил – серьезное дело, требующее определенного уровня технической грамотности.

Как видите, эти замечания высказаны в узком контексте бизнес-правил, но думаю, что область их применимости гораздо шире, что я и попытаюсь продемонстрировать в следующей главе.

Упражнения

Упражнение 10.1. В тексте главы я сказал, что существует ровно 16 двуместных связок. Выпишите все соответствующие таблицы истинности. Сколько существует одноместных связок?

Упражнение 10.2. (Уже приводилось в тексте главы, но здесь сформулировано по-другому). (а) Докажите, что все одноместные и двуместные связки можно выразить в терминах подходящих комбинаций NOT и либо AND, либо OR; (б) докажите, что все связки можно выразить в терминах одной-единственной связки.

Упражнение 10.3. Рассмотрим предикат «*x* – звезда». Если подставить вместо *x Солнце*, станет ли этот предикат высказыванием? Если нет, то почему? А если в качестве аргумента берется *Луна*?

Упражнение 10.4. Снова рассмотрим предикат «*x* – звезда». Если подставить вместо *x Солнце*, будет ли удовлетворяться этот предикат? Если нет, то почему? А если в качестве аргумента берется *Луна*?

Упражнение 10.5. Еще раз приведем ограничение CX1 из главы 8:

```
CONSTRAINT CX1 IS_EMPTY ( S WHERE STATUS < 1 OR STATUS > 100 ) ;
```

Здесь выражение IS_EMPTY (...), очевидно, является предикатом. А в главе 8 я говорил, что имя переменной-отношения «S» в этом предикате играет роль *обозначения*. Но разве это не параметр? Если нет, то в чем разница?

Упражнение 10.6. (Уже приводилось в тексте главы.) Что означает следующее выражение:

```
{ SX.SNAME } WHERE EXISTS SPX ( FORALL PX ( SPX.SNO = SX.SNO AND
                                            SPX.PNO = PX.PNO ) )
```

Упражнение 10.7. (Уже приводилось в тексте главы.) Приведите SQL-аналоги выражений реляционного исчисления из подраздела «Дополнительные примеры запросов».

Упражнение 10.8. Докажите, что операции AND и OR ассоциативны.

Упражнение 10.9. Пусть $p(x)$ и q – предикаты, в которых x соответственно встречается и не встречается в качестве свободной переменной. Какие из следующих утверждений верны? (Напоминаю, что символом «⇒» обозначается импликация, а символом «≡» – эквиваленция. Отмечу также, что одновременная истинность A ⇒ B и B ⇒ A – то же самое, что A ≡ B.)

a. `EXISTS x ( q ) ≡ q`

b. `FORALL x ( q ) ≡ q`

c. `EXISTS x ( p(x) AND q ) ≡ EXISTS x ( p(x) ) AND q`

d. `FORALL x ( p(x) AND q ) ≡ FORALL x ( p(x) ) AND q`

e. `FORALL x ( p(x) ) ⇒ EXISTS x ( p(x) )`

f. `EXISTS x ( TRUE ) ≡ TRUE`

g. `FORALL x ( FALSE ) ≡ FALSE`

h. `UNIQUE x ( p(x) ) ⇒ EXISTS x ( p(x) )`

i. `UNIQUE x ( p(x) ) ⇒ FORALL x ( p(x) )`

j. `FORALL x ( p(x) ) AND EXISTS x ( p(x) ) ⇒ UNIQUE x ( p(x) )`

k. `FORALL x ( p(x) ) AND UNIQUE x ( p(x) ) ⇒ EXISTS x ( p(x) )`

Упражнение 10.10. Пусть $p(x,y)$ – предикат со свободными переменными x и y. Какие из следующих утверждений верны?

a. `EXISTS x EXISTS y ( p(x,y) ) ≡ EXISTS y EXISTS x ( p(x,y) )`

b. `FORALL x FORALL y ( p(x,y) ) ≡ FORALL y FORALL x ( p(x,y) )`

c. `FORALL x ( p(x,y) ) ≡ NOT EXISTS x ( NOT p(x,y) )`

d. `EXISTS x ( p(x,y) ) ≡ NOT FORALL x ( NOT p(x,y) )`

e. `EXISTS x FORALL y ( p(x,y) ) ≡ FORALL y EXISTS x ( p(x,y) )`

f. `EXISTS y FORALL x ( p(x,y) ) ⇒ FORALL x EXISTS y ( p(x,y) )`

Упражнение 10.11. Пусть $p(x)$ и $q(y)$ – предикаты со свободными переменными x и y соответственно. Какие из следующих утверждений верны?

a.
```
EXISTS x ( p(x) ) AND EXISTS y ( q(y) ) ≡
EXISTS x EXISTS y ( p(x) AND q(y) )
```

b.
```
EXISTS x ( IF p(x) THEN q(x) ) ≡
IF FORALL x ( p(x) ) THEN EXISTS x ( q(x) )
```

Упражнение 10.12. Там, где это возможно и разумно, дайте решения упражнений из глав 8–9 в терминах реляционного исчисления.

Упражнение 10.13. Рассмотрим запрос: «Получить города, в которых находится либо какой-то поставщик, либо какая-то деталь». Можно ли выразить этот запрос в терминах реляционного исчисления? Если нет, то почему?

Упражнение 10.14. Покажите, что язык SQL реляционно полный (то есть докажите, что для каждого выражения реляционной алгебры или реляционного исчисления существует семантически эквивалентная формулировка на SQL).

Упражнение 10.15. Ниже приведена цитата из главы 8:

- Если база данных содержит только истинные высказывания, то она непротиворечива, но обратное может быть неверно.
- Если база данных противоречива, то она содержит хотя бы одно ложное утверждение, но обратное опять-таки неверно.

Верно ли, что эти два утверждения логически эквивалентны? Другими словами, нет ли здесь дублирования?

Упражнение 10.16. Всегда ли достижима предваренная нормальная форма?

11

Использование формальной логики для формулирования SQL-выражений

В главе 6 я описал процедуру трансформации выражений, применяемую к выражениям реляционной алгебры; я показал, как одно выражение можно трансформировать в другое, логически эквивалентное, применяя различные правила трансформации. Среди рассмотренных правил были такие:

a. Дистрибутивность ограничения относительно объединения, пересечения и разности

b. Дистрибутивность проекции относительно объединения, но не пересечения и разности

и ряд других. (Как вы, наверное, догадываетесь, аналогичные правила применимы и к выражениям реляционного исчисления, хотя в главе 10 я о них не говорил.)

Я отметил, что цель таких трансформаций – в основном оптимизация; задача состоит в том, чтобы найти выражение с такой же семантикой, как у исходного, но с лучшими показателями производительности. Однако сама идея трансформации выражений – или *переписывания запросов*, как ее иногда (не совсем правильно) называют, – находит применение и в других областях. В частности, что очень важно, ею можно воспользоваться для трансформации точных логических выражений, представляющих запросы, в SQL-эквиваленты. Именно этому и посвящена настоящая глава: будет показано, как, имея формулировку некоторого запроса или ограничения (к примеру) в терминах формальной логики или реляционного исчисления, применить систематическую процедуру преобразования ее в эквивалент на языке SQL. И хотя полученная таким образом формулировка на SQL иногда с трудом поддается пониманию, мы можем быть уверены в ее правильности благодаря фор-

мализованному способу получения. Отсюда и подзаголовок книги: *Как грамотно писать код на SQL*.

Некоторые правила трансформации

Правила трансформации, подобные упомянутым выше, известны и под другими названиями:

- *Эквиваленции*, потому что в общем виде они записываются так: *exp1* ≡ *exp2* (напомню, что символ «≡» означает «эквивалентно»).
- *Тождества*, потому что правило вида *exp1* ≡ *exp2* можно произнести так: *exp1* и *exp2* «тождественно равны», то есть обладают одинаковой семантикой.
- *Правила перезаписи*, потому из правила вида *exp1* ≡ *exp2* следует, что выражение, в которое входит *exp1*, можно переписать в виде выражения, в которое входит *exp2*, не изменяя смысла.

О последнем пункте я хотел бы поговорить подробнее, так как он особенно важен для того, чем мы займемся в этой главе. Итак, пусть *X1* – выражение, содержащее вхождение *x1* в качестве подвыражения, пусть *x2* эквивалентно *x1*, и пусть *X2* – выражение, полученное подстановкой *x2* вместо *x1* в *X1*. Тогда *X1* и *X2* логически и семантически эквивалентны, следовательно, *X1* можно переписать как *X2*.

Вот простой пример. Рассмотрим следующее SQL-выражение

```
SELECT  SNO
FROM    S
WHERE ( STATUS > 10 AND CITY = 'London' )
OR    ( STATUS > 10 AND CITY = 'Athens' )
```

Булево выражение во фразе WHERE, очевидно, эквивалентно такому:

```
STATUS > 10 AND ( CITY = 'London' OR CITY = 'Athens' )
```

Поэтому все выражение можно переписать в виде

```
SELECT    SNO
FROM      S
WHERE     STATUS > 10
AND     ( CITY = 'London' OR CITY = 'Athens' )
```

Перечислю некоторые правила трансформации, которыми мы будем пользоваться в этой главе.

- *Правило импликации*

  ```
  IF p THEN q  ≡ ( NOT p ) OR q
  ```

  Я сформулировал это правило в главе 10, но не привел примеров. Потратьте некоторое время (если вам это требуется) на то, чтобы проверить таблицы истинности и убедиться, что это правило действительно справедливо. *Примечание*: Символами *p* и *q* обозначают-

ся произвольные булевы выражения, или предикаты. В этой главе я буду употреблять термин *булево выражение*, а не *предикат*, потому что упор делается именно на выражения, то есть фрагменты текста программы, а не на формальную логику. Логика вообще и предикаты в частности находятся на более абстрактном уровне, чем фрагменты программы (по крайней мере, такое заявление не лишено оснований). В предыдущей главе мы отмечали, что любое булево выражение можно рассматривать как конкретное представление некоторого предиката.

- *Закон двойного отрицания*

```
NOT ( NOT p ) ≡ p
```

Это правило очевидно (но тем не менее важно).

- *Законы де Моргана*

```
NOT ( p AND q ) ≡ ( NOT p ) OR  ( NOT q )
NOT ( p OR  q ) ≡ ( NOT p ) AND ( NOT q )
```

Я ничего не говорил об этих законах в предыдущей главе, но полагаю, что интуитивно они понятны. Например, первый утверждает следующее: если неверно, что *p* и *q* одновременно истинны, значит, либо не истинно *p*, либо не истинно *q* (либо оба вместе). Так или иначе, справедливость обоих законов без труда проверяется по таблицам истинности. Вот, например, таблица истинности, соответствующая первому закону:

p *q*	*p* AND *q*	NOT (*p* AND *q*)	(NOT *p*) OR (NOT *q*)
T T	T	F	F
T F	F	T	T
F T	F	T	T
F F	F	T	T

Справедливость первого закона де Моргана следует из того, что столбцы NOT (*p* AND *q*) и (NOT *p*) OR (NOT *q*) одинаковы. Доказательство второго закона аналогично.

- *Дистрибутивные законы*

```
p AND ( q OR  r ) ≡ ( p AND q ) OR  ( p AND r )
p OR  ( q AND r ) ≡ ( p OR  q ) AND ( p OR  r )
```

Оставляю доказательство справедливости этих законов читателю. Отмечу, однако, что воспользовался первым из них в примере трансформации SQL-запроса в начале этой главы. Возможно, вы обратили внимание на то, что в некотором смысле эти дистрибутивные законы носят более общий характер, чем те, что мы видели в главе 6. Там мы рассматривали примеры дистрибутивности одноместного оператора, скажем ограничения, относительно двуместного (объединения); здесь же речь идет о дистрибутивности двуместных операторов (AND и OR) относительно друг друга.

- *Правило квантификации*

```
FORALL x ( p ( x ) )  ≡  NOT EXISTS x ( NOT p ( x ) )
```

  Это правило я тоже обсуждал в предыдущей главе. Надеюсь, вы заметили, что в действительности это просто применение законов де Моргана к выражениям EXISTS и FORALL (напомню, что в предыдущей главе было показано, что EXISTS и FORALL – это, по существу, последовательность повторяющихся OR и AND соответственно).

Еще одно замечание об этих законах: поскольку законы де Моргана часто применяются к результату применения правила импликации, то удобно хотя бы первый из них переписать в таком виде (где вместо *q* подставлено NOT *q* и молчаливо применен закон двойного отрицания):

```
NOT ( p AND NOT q )  ≡  ( NOT p ) OR q
```

Или в таком (логически эквивалентном):

```
( NOT p ) OR q  ≡  NOT ( p AND NOT q )
```

Или, что то же самое:

```
IF p THEN q  ≡  NOT ( p AND NOT q )
```

По большей части ссылки на законы де Моргана в данной главе будут относиться к этой последней формулировке.

Далее в этой главе будут предложены практические рекомендации по применению этих законов к формулировке «сложных» SQL-выражений. Я начну с простых примеров и буду двигаться в сторону усложнения.

Пример 1. Логическая импликация

Рассмотрим еще раз упомянутое в предыдущей главе ограничение, требующее, чтобы все красные детали хранились в Лондоне. Для конкретной детали этому ограничению соответствует бизнес-правило, формулируемое (более-менее формально) следующим образом:

```
IF COLOR = 'Red' THEN CITY = 'London'
```

Иначе говоря, это логическая импликация. Правда, в SQL логическая импликация в чистом виде не поддерживается, но правило импликации позволяет трансформировать это выражение в такое:

```
( NOT ( COLOR = 'Red' ) ) OR CITY = 'London'
```

(Скобки я добавил для ясности.) А в этом выражении встречаются только операторы, поддерживаемые SQL, поэтому его уже можно сформулировать как ограничение базовой таблицы:

```
CONSTRAINT BTCX1 CHECK ( NOT ( COLOR = 'Red' ) OR CITY = 'London' )
```

Или, быть может, несколько более естественно (учитывая тот факт, что NOT ($a = b$) можно трансформировать в $a \neq b$ (в нотации SQL $a <> b$) и опустить некоторые скобки):

```
CONSTRAINT BTCX1 CHECK ( COLOR <> 'Red' OR CITY = 'London' )
```

Пример 2. Добавление квантора всеобщности

Сознаюсь, что в примере 1 я немного схитрил, притворившись, будто понятно, к какой конкретной детали применяется ограничение. Но природа ограничений базовых таблиц в SQL такова, что они применяются к каждой строке базовой таблицы, в определении которой встречаются. Предположим, однако, что мы хотим явно сформулировать, что ограничение должно применяться к каждой детали, которая присутствует в таблице P. Иными словами, для любой детали PX, если цвет PX красный, городом PX является Лондон:

```
FORALL PX ( IF PX.COLOR = 'Red' THEN PX.CITY = 'London' )
```

Примечание

Имя PX и подобные ему в этой главе специально выбраны так, чтобы напоминать о переменных кортежа, которые использовались в предыдущей главе. Фактически, начиная с этого места, я буду предполагать, что имена вида PX, PY, PZ и т. д. обозначают переменные, областью значений которых является таблица P; для переменных с именами вида SX, SY, SZ и т. д. областью значений является таблица S и так далее. Детали того, как эти переменные определены (в формальной логике, а не в SQL), сейчас для нас несущественны и потому опускаются. В SQL они определяются с помощью фразы AS, как я продемонстрирую, когда мы дойдем до SQL-формулировок.

В SQL квантор FORALL не поддерживается, однако правила квантификации позволяют трансформировать приведенное выше выражение в такое:

```
NOT EXISTS PX ( NOT ( IF PX.COLOR = 'Red'
                      THEN PX.CITY = 'London' ) )
```

(И снова я для ясности добавил скобки. Начиная с этого места, я буду добавлять скобки или убирать уже имеющиеся без специальных комментариев.) Теперь применим правило импликации:

```
NOT EXISTS PX ( NOT ( NOT ( PX.COLOR = 'Red' )
                      OR PX.CITY = 'London' ) )
```

Это выражение уже можно было бы отобразить на SQL, но, пожалуй, сначала стоит еще немного поработать над ним. Применяем закон де Моргана:

```
NOT EXISTS PX ( NOT ( NOT ( ( PX.COLOR = 'Red' )
                AND NOT ( PX.CITY = 'London' ) ) ) )
```

Применяем закон двойного отрицания и опускаем некоторые скобки:

```
NOT EXISTS PX ( PX.COLOR = 'Red' AND NOT ( PX.CITY = 'London' ) )
```

И окончательно

```
NOT EXISTS PX ( PX.COLOR = 'Red' AND PX.CITY ≠ 'London' )
```

Все примененные до сих пор трансформации были очень простыми, вы, наверное, даже заскучали. Однако отображение последнего логического выражения на SQL не так уж тривиально. Опишу подробности этого отображения:

- Во-первых (что не удивительно), NOT отображается на NOT.
- Во-вторых, «EXISTS PX (*bx*)» отображается на «EXISTS (SELECT * FROM P AS PX WHERE (*sbx*))», где *sbx* – SQL-аналог булевого выражения *bx*.
- В-третьих, скобки, окружающие *sbx*, можно опустить, хотя это и не обязательно.
- И наконец, все выражение целиком следует погрузить в подходящее предложение CREATE ASSERTION.

Вот как выглядит окончательный результат:

```
CREATE ASSERTION ... CHECK
     ( NOT EXISTS ( SELECT *
                    FROM   P AS PX
                    WHERE  PX.COLOR = 'Red'
                    AND    PX.CITY <> 'London' ) ) ;
```

Пример 3. Импликация и квантор всеобщности

На этот раз рассмотрим запрос «Получить названия деталей, для которых вес отличается от веса любой детали, находящейся в Париже». Вот как выглядит прямолинейная формулировка в терминах формальной логики (то есть реляционного исчисления):

```
{ PX.PNAME } WHERE FORALL PY ( IF PY.CITY = 'Paris'
                               THEN PY.WEIGHT ≠ PX.WEIGHT )
```

Это выражение можно интерпретировать следующим образом: «получить значения PNAME для деталей PX таких, что для всех деталей PY, если PY находится в Париже, то веса PY и PX различны». Обратите внимание, что в подобных интерпретациях на естественном языке я считаю обороты *где* и *такой что* синонимами и употребляю тот, что лучше звучит в конкретном контексте.

В качестве первой трансформации применим правило квантификации:

```
{ PX.PNAME } WHERE NOT EXISTS PY ( NOT ( IF PY.CITY = 'Paris'
                                         THEN PY.WEIGHT ≠ PX.WEIGHT ) )
```

Затем – правило импликации:

```
{ PX.PNAME } WHERE
             NOT EXISTS PY ( NOT ( NOT ( PY.CITY = 'Paris' )
                                 OR ( PY.WEIGHT ≠ PX.WEIGHT ) ) )
```

Теперь – закон де Моргана:

```
{ PX.PNAME } WHERE
             NOT EXISTS PY ( NOT ( NOT ( ( PY.CITY = 'Paris' )
                                 AND NOT ( PY.WEIGHT ≠ PX.WEIGHT ) ) ) )
```

Наведем красоту, воспользовавшись законом двойного отрицания и тем, что NOT ($a \neq b$) эквивалентно $a = b$:

```
{ PX.PNAME } WHERE NOT EXISTS PY ( PY.CITY = 'Paris' AND
                                   PY.WEIGHT = PX.WEIGHT )
```

Отображаем на SQL:

```
SELECT DISTINCT PX.PNAME
FROM   P AS PX
WHERE  NOT EXISTS
     ( SELECT *
       FROM   P AS PY
       WHERE  PY.CITY = 'Paris'
       AND    PY.WEIGHT = PX.WEIGHT )
```

Кстати, слово DISTINCT в первой фразе SELECT необходимо! Вот что получается в результате:[1]

PNAME
Screw
Cog

К сожалению, в эту бочку меда попала ложка дегтя. Предположим, что в Париже есть хотя бы одна деталь, но у всех таких деталей вес равен null. Тогда мы попросту не знаем – не можем сказать, – существуют ли какие-нибудь детали, вес которых отличается от веса любой детали, находящейся в Париже (то есть, на исходный вопрос, строго говоря, нельзя дать ответ). Однако в SQL подзапрос, следующий за ключевым словом EXISTS, возвращает пустую таблицу для любой детали PX, представленной в P, поэтому NOT EXISTS будет принимать значение TRUE для любой такой детали PX, и выражение в целом вернет названия всех деталей в таблице P, что совершенно неправильно.

[1] Результаты всех запросов в этой главе основаны на наших привычных тестовых данных. *Примечание*: Рецензенты сообщили, что по меньшей мере две разных SQL-системы дают один и тот же результат вне зависимости от того, задано слово DISTINCT или нет. Если это так, то в этих системах, по-видимому, имеется ошибка.

Отступление

Это, пожалуй, самая серьезная практическая проблема, связанная с null-значениями, – они ведут к неверным ответам. Еще хуже то, что в общем случае мы даже не знаем, какие ответы верны, а какие – нет! Более подробное освещение этой темы см. в статье «Почему не работает трех- и четырехзначная логика», упомянутой в приложении D.

Скажу больше – неправилен не только результат предыдущего SQL-запроса, но и любой определенный результат следовало бы рассматривать как ложь со стороны системы. Еще раз повторю, единственный логически правильный результат в данном случае – «Я не знаю», или, если быть более точным и честным: «У системы нет достаточной информации, чтобы дать определенный ответ на этот запрос».

Еще более отягчает ситуацию тот факт, что при тех же самых условиях (в Париже есть хотя бы одна деталь, и все такие детали имеют вес null) SQL-выражение

```
SELECT DISTINCT PX.PNAME
FROM   P AS PX
WHERE  PX.WEIGHT NOT IN ( SELECT PY.WEIGHT
                          FROM   P AS PY
                          WHERE  PY.CITY = 'Paris' )
```

которое вроде бы должно быть эквивалентно показанному выше (и так оно и есть в отсутствие null-значений), возвращает пустой результат: иной, хотя и столь же неправильный, ответ.

Мораль очевидна: избегайте null-значений! И тогда все трансформации будут работать корректно.

Пример 4. Коррелированные подзапросы

Рассмотрим запрос «Получить названия поставщиков, которые поставляют одновременно детали P1 и P2». Вот его формально-логическая формулировка:

```
{ SX.SNAME } WHERE
             EXISTS SPX ( SPX.SNO = SX.SNO AND SPX.PNO = 'P1' ) AND
             EXISTS SPX ( SPX.SNO = SX.SNO AND SPX.PNO = 'P2' )
```

Эквивалентная SQL-формулировка записывается без труда:

```
SELECT DISTINCT SX.SNAME
FROM   S AS SX
WHERE  EXISTS ( SELECT *
                FROM   SP AS SPX
                WHERE  SPX.SNO = SX.SNO
                AND    SPX.PNO = 'P1' )
AND    EXISTS ( SELECT *
                FROM   SP AS SPX
```

```
WHERE  SPX.SNO = SX.SNO
AND    SPX.PNO = 'P2' )
```

И приводит к такому результату:

SNAME
Smith
Jones

Однако легко видеть, что SQL-выражение содержит два коррелированных подзапроса. (На самом деле, в примере 3 тоже встречался коррелированный подзапрос. Дополнительные сведения см. в главе 12.) Но коррелированные подзапросы часто противопоказаны с точки зрения производительности, потому что (во всяком случае концептуально) их необходимо повторно вычислять для каждой строки внешней таблицы, а не раз и навсегда. Поэтому стоит подумать, нельзя ли их устранить. В рассматриваемом случае (когда коррелированные подзапросы встречаются внутри вызова EXISTS) существует простое преобразование, позволяющее добиться указанной цели. В результате получается такое выражение:

```
SELECT DISTINCT SX.SNAME
FROM   S AS SX
WHERE  SX.SNO IN ( SELECT SPX.SNO
                   FROM   SP AS SPX
                   WHERE  SPX.PNO = 'P1' )
AND    SX.SNO IN ( SELECT SPX.SNO
                   FROM   SP AS SPX
                   WHERE  SPX.PNO = 'P2' )
```

Более общо, SQL-выражение

```
SELECT sic  /* «SELECT item commalist» - список элементов SELECT */
FROM   T1
WHERE  [ NOT ] EXISTS ( SELECT *
                        FROM   T2
                        WHERE  T2.C = T1.C
                        AND    bx )
```

можно трансформировать в

```
SELECT sic
FROM   T1
WHERE  T1.C [ NOT ] IN ( SELECT T2.C
                         FROM   T2
                         WHERE  bx )
```

На практике такую трансформацию имеет смысл применять всюду, где возможно. (Конечно, было бы лучше, если бы оптимизатор умел делать это автоматически, но, к сожалению, мы не можем рассчитывать на то, что оптимизатор всегда примет наилучшее решение.) Однако есть много

случаев, когда эта трансформация попросту неприменима. Как показывает пример 3, одной из причин является наличие null-значений (кстати, а являются ли null-значения проблемой в примере 4?), но есть ситуации, когда ее не удается применить, даже если никаких null-значений нет и в помине. В качестве упражнения можете попытаться понять, к каким из следующих далее примеров эта трансформация применима.

Пример 5. Именование подвыражений

Рассмотрим запрос «Получить всю информацию о поставщиках, которые поставляют все фиолетовые детали». *Примечание*: такой или похожий запрос часто служит для демонстрации изъяна в той форме реляционного оператора деления, которая была предложена изначально. См. замечание по этому поводу в конце раздела.

Вот первая формулировка в терминах логики:

```
{ SX } WHERE FORALL PX ( IF PX.COLOR = 'Purple' THEN
             EXISTS SPX ( SPX.SNO = SX.SNO AND SPX.PNO = PX.PNO ) )
```

(«названия поставщиков SX таких, что для всех деталей PX, если PX фиолетовая, то существует поставка SPX, в которой SNO равно номеру поставщика SX, а PNO равно номеру детали PX»). Сначала применим правило импликации:

```
{ SX } WHERE FORALL PX ( NOT ( PX.COLOR = 'Purple' ) OR
             EXISTS SPX ( SPX.SNO = SX.SNO AND SPX.PNO = PX.PNO ) )
```

Затем – закон де Моргана:

```
{ SX } WHERE
       FORALL PX ( NOT ( ( PX.COLOR = 'Purple' ) AND
       NOT EXISTS SPX ( SPX.SNO = SX.SNO AND SPX.PNO = PX.PNO ) ) )
```

Теперь – правило квантификации:

```
{ SX } WHERE
       NOT EXISTS PX ( NOT ( NOT ( ( PX.COLOR = 'Purple' ) AND
       NOT EXISTS SPX ( SPX.SNO = SX.SNO AND SPX.PNO = PX.PNO ) ) ) )
```

Правило двойного отрицания:

```
{ SX } WHERE
       NOT EXISTS PX ( ( PX.COLOR = 'Purple' ) AND
       NOT EXISTS SPX ( SPX.SNO = SX.SNO AND SPX.PNO = PX.PNO ) )
```

Опускаем лишние скобки и отображаем на SQL:

```
SELECT *
FROM   S AS SX
WHERE  NOT EXISTS
     ( SELECT *
       FROM   P AS PX
```

```
WHERE  PX.COLOR = 'Purple'
AND    NOT EXISTS
     ( SELECT *
       FROM   SP AS SPX
       WHERE  SPX.SNO = SX.SNO
       AND    SPX.PNO = PX.PNO ) )
```

Результатом является все отношение «поставщики»:[1]

SNO	SNAME	STATUS	CITY
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

Возможно, вы не совсем понимаете, как были выполнены трансформации в этом примере и почему получился такой результат. Когда выражения становятся слишком сложными, имеет смысл применить полезный прием, заключающийся в том, чтобы ввести символические имена для подвыражений. Обозначим *exp1* подвыражение

```
PX.COLOR = 'Purple'
```

а *exp2* – такое подвыражение:

```
EXISTS SPX ( SPX.SNO = SX.SNO AND SPX.PNO = PX.PNO )
```

(отметим, что оба подвыражения можно более или менее непосредственно представить на SQL). Тогда исходное выражение реляционного исчисления принимает вид:

```
{ SX } WHERE FORALL PX ( IF exp1 THEN exp2 )
```

Теперь за деревьями стал виден лес, и мы можем приступить к применению обычных трансформаций, хотя сейчас представляется более разумным применить их в другом порядке, именно потому что мы лучше ухватываем всю картину в целом. Итак, сначала правило квантификации:

```
{ SX } WHERE NOT EXISTS PX ( NOT ( IF exp1 THEN exp2 ) )
```

Затем правило импликации:

```
{ SX } WHERE NOT EXISTS PX ( NOT ( NOT ( exp1 ) OR exp2 ) )
```

Закон де Моргана:

```
{ SX } WHERE NOT EXISTS PX ( NOT ( NOT ( exp1 AND NOT ( exp2 ) ) ) )
```

[1] Напомню (см. главу 7), что, поскольку фиолетовых деталей нет вообще, то каждый поставщик поставляет их все, – даже поставщик S5, который не поставляет никаких деталей. См. обсуждение пустых множеств значений в главе 10.

Двойное отрицание:

```
{ SX } WHERE NOT EXISTS PX ( exp1 AND NOT ( exp2 ) )
```

И наконец, разворачиваем *exp1* и *exp2* и отображаем все выражение на SQL:

```
SELECT *
FROM   S AS SX
WHERE  NOT EXISTS
     ( SELECT *
       FROM   P AS PX
       WHERE  PX.COLOR = 'Purple'
       AND    NOT EXISTS
            ( SELECT *
              FROM   SP AS SPX
              WHERE  SPX.SNO = SX.SNO
              AND    SPX.PNO = PX.PNO ) )
```

Я полагаю, этот пример убедительно демонстрирует, что SQL-выражения, получаемые с помощью обсуждаемых методов, часто оказываются сложны для восприятия, но, как я уже говорил, они заведомо корректны, потому что для их вывода систематически применялись строгие правила.

Не могу противиться искушению привести версию этого примера на языке **Tutorial D** – просто для сравнения:

```
S WHERE ( !!SP ) { PNO } ⊇ ( P WHERE COLOR = 'Purple' ) { PNO }
```

А теперь я хотел бы вернуться к замечанию об операторе деления, сделанному в начале этого раздела. Для простоты обозначим символом PP оператор ограничения P WHERE COLOR = 'Purple'. Кроме того, упростим запрос – «Получить всю информацию о поставщиках, которые поставляют все фиолетовые детали» – так, чтобы запрашивалась не вся информация, а только номера поставщиков. Тогда можно подумать, что этот запрос представляется следующим выражением:

```
SP { SNO , PNO } DIVIDEBY PP { PNO }
```

Примечание

Здесь DIVIDEBY представляет оператор деления, определенный в оригинальной модели. Если вам нужны пояснения по этому поводу, обратитесь к главе 7.

Однако на наших тестовых данных отношение PP, а, стало быть, и проекция PP на {PNO}, пусты (поскольку фиолетовых деталей не существует), и это выражение возвращает номера поставщиков S1, S2, S3, S4. Но если фиолетовых деталей нет, то любой поставщик поставляет их все (см. обсуждение пустых множеств значений в предыдущей главе) – *даже поставщик S5*, который не поставляет вообще никаких деталей. В то же время приведенная выше операция деления не может вернуть номер поставщика S5, так как номера поставщиков извлекаются из SP,

а не из S, а поставщик S5 в SP не представлен. Поэтому неформальное описание такой операции деления как «Получить номера поставщиков, которые поставляют все фиолетовые детали» некорректно, правильнее было бы сказать «Получить номера поставщиков, которые поставляют *хотя бы одну фиолетовую деталь и при этом* поставляют все такие детали». Поэтому, как показывает этот пример (и повторяя сказанное в главе 7), оператор деления не решает задачу, для решения которой задумывался.

Пример 6. Еще об именовании подвыражений

Я приведу еще один пример, иллюстрирующий полезность введения символических имен для подвыражений. Мы рассмотрим запрос «Получить поставщиков таких, что каждая поставляемая ими деталь находится в том же городе, где и сам поставщик». Вот его формулировка в терминах формальной логики:

```
{ SX } WHERE FORALL PX
     ( IF EXISTS SPX ( SPX.SNO = SX.SNO AND SPX.PNO = PX.PNO )
       THEN PX.CITY = SX.CITY )
```

(«поставщики SX такие, что для любой детали PX, если существует поставка PX поставщиком SX, то PX.CITY = SX.CITY»).

На этот раз я покажу трансформации, не называя применяемых на каждом шаге правил (оставляю это в качестве упражнения для читателя):

```
{ SX } WHERE FORALL PX ( IF exp1 THEN exp2 )

{ SX } WHERE NOT EXISTS PX ( NOT ( IF exp1 THEN exp2 ) )

{ SX } WHERE NOT EXISTS PX ( NOT ( NOT ( exp1 ) OR exp2 ) )

{ SX } WHERE NOT EXISTS PX ( NOT ( NOT ( exp1 AND NOT ( exp2 ) ) ))

{ SX } WHERE NOT EXISTS PX ( exp1 AND NOT ( exp2 ) )
```

Теперь разворачиваем *exp1* и *exp2* и отображаем на SQL:

```
SELECT *
FROM   S AS SX
WHERE  NOT EXISTS
     ( SELECT *
       FROM   P AS PX
       WHERE  EXISTS
            ( SELECT *
              FROM   SP AS SPX
              WHERE  SPX.SNO = SX.SNO
              AND    SPX.PNO = PX.PNO )
              AND    PX.CITY <> SX.CITY )
```

Результат:

SNO	SNAME	STATUS	CITY
S3	Blake	30	Paris
S5	Adams	30	Athens

Кстати, если этот результат кажется вам странным, обратите внимание, что поставщик S3 поставляет только одну деталь P2, а поставщик S5 не поставляет никаких деталей, поэтому, с точки зрения логики, оба эти поставщика удовлетворяют условию, что «каждая поставляемая ими деталь» находится в том же городе, где и сам поставщик.

Для интересующихся приведу версию этого примера на языке **Tutorial D**:

```
S WHERE RELATION { TUPLE { CITY CITY } } = ( ( !!SP ) JOIN P ) { CITY }
```

Пример 7. Устранение неоднозначности

Естественный язык часто неоднозначен. Рассмотрим, к примеру, следующий запрос: «Получить поставщиков таких, что каждая поставляемая ими деталь находится в том же городе». Во-первых, обратите внимание на тонкое (?) различие между этим примером и предыдущим. Во-вторых, и это важнее, заметьте, что эта формулировка на естественном языке действительно неоднозначна! Для определенности буду предполагать, что она означает следующее:

> Получить поставщиков SX таких, что для любых деталей PX и PY, если SX поставляет обе детали, то PX.CITY = PY.CITY.

Отметим, что поставщик, который поставляет всего одну деталь, удовлетворяет этой интерпретации условия. (Как, кстати, и поставщик, не поставляющий ни одной детали.) Альтернативно запрос мог бы означать следующее:

> Получить поставщиков SX таких, что для любых деталей PX и PY, если SX поставляет обе детали *и они различны*, то PX.CITY = PY.CITY.

Такому условию поставщик, который поставляет всего одну деталь или не поставляет ни одной детали, уже не удовлетворяет.

Я уже сказал, что для определенности остановлюсь на первой интерпретации. Но замечу, что такого рода неоднозначности – обычное дело в сложных запросах и в сложных бизнес-правилах, и одно из достоинств формальной логики в том и заключается, что она помогает выявлять и разрешать их.

Вот формулировка первой интерпретации в терминах логики:

```
{ SX } WHERE FORALL PX ( FORALL PY
     ( IF EXISTS SPX ( SPX.SNO = SX.SNO AND SPX.PNO = PX.PNO )
```

```
        AND EXISTS SPY ( SPY.SNO = SX.SNO AND SPY.PNO = PY.PNO )
        THEN PX.CITY = PY.CITY ) )
```

А вот и трансформации (снова оставляю читателю выяснение того, какое правило применено на каждом шаге):

```
{ SX } WHERE FORALL PX ( FORALL PY
           ( IF exp1 AND exp2 THEN exp3 ) )

{ SX } WHERE NOT EXISTS PX ( NOT FORALL PY
           ( IF exp1 AND exp2 THEN exp3 ) )

{ SX } WHERE NOT EXISTS PX ( NOT ( NOT EXISTS PY ( NOT
           ( IF exp1 AND exp2 THEN exp3 ) ) ) )

{ SX } WHERE NOT EXISTS PX ( EXISTS PY ( NOT
           ( IF exp1 AND exp2 THEN exp3 ) ) )

{ SX } WHERE NOT EXISTS PX ( EXISTS PY ( NOT
           ( NOT ( exp1 AND exp2 ) OR exp3 ) ) )

{ SX } WHERE NOT EXISTS PX ( EXISTS PY ( NOT
           ( NOT ( exp1 ) OR NOT ( exp2 ) OR exp3 ) ) )

{ SX } WHERE NOT EXISTS PX ( EXISTS PY (
           ( exp1 AND exp2 AND NOT ( exp3 ) ) ) )
```

SQL-эквивалент:

```
SELECT *
FROM   S AS SX
WHERE  NOT EXISTS
     ( SELECT *
       FROM   P AS PX
       WHERE  EXISTS
            ( SELECT *
              FROM   P AS PY
              WHERE  EXISTS
                   ( SELECT *
                     FROM   SP AS SPX
                     WHERE  SPX.SNO = SX.SNO
                     AND    SPX.PNO = PX.PNO )
              AND    EXISTS
                   ( SELECT *
                     FROM   SP AS SPY
                     WHERE  SPY.SNO = SX.SNO
                     AND    SPY.PNO = PY.PNO )
              AND    PX.CITY <> PY.CITY ) )
```

Кстати, в этом примере я использовал две переменных кортежа SPX и SPY, обе с множеством значений SP, просто для ясности; я вполне мог бы дважды воспользоваться одной и той же переменной (скажем,

SPX) – никакого логического различия тут нет! Вот какой получается результат:

SNO	SNAME	STATUS	CITY
S3	Blake	30	Paris
S5	Adams	30	Athens

В этом месте я хотел бы упомянуть еще об одном правиле трансформации, которое иногда бывает полезно: правиле контрапозиции. Рассмотрим импликацию IF NOT *q* THEN NOT *p*. По определению, это выражение эквивалентно NOT (NOT *q*) OR NOT *p*, то есть *q* OR NOT *p*, то есть NOT *p* OR *q*, то есть IF *p* THEN *q*. Таким образом, имеем:

```
IF p THEN q  ≡  IF NOT q THEN NOT p
```

Отметим, что этот закон интуитивно очевиден: если из истинности *p* следует истинность *q*, то из ложности *q* должна следовать ложность *p*. Например, если из утверждения «Идет дождь» следует, что «Улицы мокрые», то из утверждения «Улицы не мокрые» должно следовать, что «Дождь не идет».

Значит, в рассматриваемом примере мы можем переформулировать ранее принятую интерпретацию (Получить поставщиков SX таких, что для любых деталей PX и PY, если SX поставляет обе детали, то PX.CITY = PY.CITY) следующим образом:

> Получить поставщиков SX таких, что для любых деталей PX и PY, если PX.CITY ≠ PY.CITY, то SX не поставляет обе детали. (Кстати, так ли очевидно, что этот вариант эквивалентен предыдущему?)

Такая интерпретация запроса вполне может привести к другой (хотя и логически эквивалентной) формулировке его на SQL. Детали оставляю читателю в качестве упражнения.

Пример 8. Использование COUNT

О предыдущем примере сказано еще не все. Повторю сам запрос: «Получить поставщиков таких, что каждая поставляемая ими деталь находится в том же городе». Вот еще одна возможная его интерпретация на естественном языке:

> Получить поставщиков SX таких, что количество городов для деталей, поставляемых SX, меньше или равно единице.

Кстати, обратите внимание на слова «меньше или равно», если бы я написал просто «равно», то получил бы другую интерпретацию запроса. Вот формулировка в терминах логики:

```
{ SX } WHERE COUNT ( PX.CITY WHERE EXISTS SPX
                   ( SPX.SNO = SX.SNO AND SPX.PNO = PX.PNO ) ) ≤ 1
```

Это первый пример в этой главе, где встречается агрегатный оператор. Но, как вы наверняка понимаете, отображение не вызывает сложностей. Эквивалентная формулировка на SQL выглядит так:

```
SELECT *
FROM   S AS SX
WHERE  ( SELECT COUNT ( DISTINCT PX.CITY )
         FROM   P AS PX
         WHERE  EXISTS ( SELECT *
                         FROM   SP AS SPX
                         WHERE  SPX.SNO = SX.SNO
                         AND    SPX.PNO = PX.PNO ) ) <= 1
```

Результат такой же, как в примере 7. Однако хочу напомнить сказанное в предыдущей главе – вообще говоря, из соображений производительности не стоит прибегать к использованию оператора COUNT, в частности, не нужно применять его там, где EXISTS выглядит логически более правильным.

Вот несколько вопросов для читателя. Во-первых, действительно ли необходимо ключевое слово DISTINCT в вызове COUNT при такой формулировке SQL-запроса? Во-вторых, попытайтесь сформулировать этот запрос в терминах GROUP BY и HAVING. Если получится, то опишите логические шаги, с помощью которых вы перешли к такой формулировке. (Дальнейшее обсуждение GROUP BY и HAVING см. в примере 12.)

Пример 9. Запросы с соединением

На этот раз я дам вам возможность попрактиковаться: приведу только сам запрос и его формулировку на SQL, а логическую формулировку и процесс вывода одного из другого оставлю вам. Запрос звучит так: «Получить поставщиков таких, что каждая поставляемая ими деталь находится в том же самом городе (как в примерах 7 и 8), а также включить в результат сам город». А вот его запись на SQL:

```
SELECT DISTINCT SX.* , PX.CITY
FROM   S AS SX , P AS PX
WHERE  EXISTS
     ( SELECT *
       FROM   SP AS SPX
       WHERE  SPX.SNO = SX.SNO
       AND    NOT EXISTS
            ( SELECT *
              FROM   SP AS SPY
              WHERE  SPY.SNO = SPX.SNO
              AND    EXISTS
                   ( SELECT *
                     FROM   P AS PY
                     WHERE  PY.PNO = SPY.PNO
                     AND    PY.CITY <> PX.CITY ) ) )
```

Результат:

SNO	SNAME	STATUS	CITY
S3	Blake	30	Paris

Упражнение: Необходимо ли ключевое слово DISTINCT в этом примере? И почему этот раздел называется «Запросы с соединением»?

Пример 10. Квантор UNIQUE

Вспомним пример из главы 10:

```
CONSTRAINT CX6 FORALL SPX ( UNIQUE SX ( SX.SNO = SPX.SNO ) ) ;
```

Это формально-логическая формулировка ограничения, согласно которому для каждой поставки должен существовать ровно один поставщик.

Теперь вспомним, что логическое выражение «EXISTS SX (*bx*)» отображается на SQL-выражение «EXISTS (SELECT * FROM S AS SX WHERE (*sbx*))», где *sbx* – SQL-аналог булевого выражения *bx*. Однако логическое выражение «UNIQUE SX (*bx*)» *не* отображается на SQL-выражение «UNIQUE (SELECT * FROM S AS SX WHERE (*sbx*))»! (Это ловушка для непосвященных.) А вот на какое выражение оно отображается:

```
UNIQUE ( SELECT * FROM S AS SX WHERE ( sbx ) )
AND
EXISTS ( SELECT * FROM S AS SX WHERE ( sbx ) )
```

Таким образом, булевая часть ограничения CX6 отображается на:

```
NOT EXISTS
  ( SELECT *
    FROM   SP AS SPX
    WHERE  NOT UNIQUE
         ( SELECT *
           FROM   S AS SX
           WHERE  SX.SNO = SPX.SNO ) )
    OR     NOT EXISTS
         ( SELECT *
           FROM   S AS SX
           WHERE  SX.SNO = SPX.SNO ) )
```

Здесь есть еще одна эквиваленция, которой можно воспользоваться, – логическое выражение UNIQUE SX (*bx*), очевидно, эквивалентно такому:

```
COUNT ( SX WHERE ( bx ) ) = 1
```

В результате предыдущую формулировку можно свести к такой:

```
NOT EXISTS
  ( SELECT *
    FROM   SP AS SPX
```

```
WHERE  ( SELECT COUNT ( * )
         FROM   S AS SX
         WHERE  SX.SNO = SPX.SNO ) ) <> 1
```

Ради интереса приведу еще одну формулировку на SQL, в которой не используется ни UNIQUE, ни COUNT. Попытайтесь убедиться в ее правильности.

```
NOT EXISTS
  ( SELECT *
    FROM   SP AS SPX
    WHERE  NOT EXISTS
         ( SELECT *
           FROM   S AS SX
           WHERE  SX.SNO = SPX.SNO
           AND    NOT EXISTS
                ( SELECT *
                  FROM   S AS SY
                  WHERE  SY.SNO = SX.SNO
                  AND  ( SY.SNAME <> SX.SNAME OR
                         SY.STATUS <> SX.STATUS OR
                         SY.CITY <> SX.CITY ) ) ) )
```

Однако обратите особое внимание, что эта формулировка опирается на тот факт, что строки-дубликаты запрещены (в частности, в таблице S); в противном случае она не годится. Избегайте строк-дубликатов!

Пример 11. Сравнения с ALL или ANY

Возможно, вы знаете, что SQL поддерживает *сравнения с модификаторами ALL или ANY* (более формально они называются *квантифицированными сравнениями*, но я предпочитаю держаться подальше от этого термина из-за возможной путаницы с операторами EXISTS и UNIQUE в SQL). Сравнением с модификатором ALL или ANY называется выражение вида *rx* θ *sq*, где *rx* – строковое выражение, *sq* – подзапрос, а θ – любой из скалярных операторов сравнения, поддерживаемых в SQL («=», «<>», «<», «<=», «>», «>=»), за которым следует одно из ключевых слов ALL, ANY или SOME. (SOME – это синоним ANY, а табличные подзапросы будут подробнее рассматриваться в главе 12.) Семантика такого запроса описывается следующим образом.

- Сравнение с модификатором ALL возвращает TRUE, если такое же сравнение без ALL возвращает TRUE для любой строки таблицы, представленной подзапросом *sq*. (Если эта таблица пуста, сравнение возвращает TRUE.)
- Сравнение с модификатором ANY возвращает TRUE, если такое же сравнение без ANY возвращает TRUE хотя бы для одной строки таблицы, представленной подзапросом *sq*. (Если эта таблица пуста, сравнение возвращает FALSE.)

Вот пример («Получить названия деталей, вес которых больше веса каждой синей детали»).

```
SELECT DISTINCT PX.PNAME
FROM   P AS PX
WHERE  PX.WEIGHT >ALL ( SELECT PY.WEIGHT
                        FROM   P AS PY
                        WHERE  PY.COLOR = 'Blue' )
```

Результат:

PNAME
Bolt
Screw
Cog

Как видно из этого примера, «строковое выражение» *rx* в сравнении с модификатором ALL или ANY *rx* θ *sq* часто – на самом деле, почти всегда – представляет собой простое скалярное выражение, и в таком случае скалярное значение, обозначаемое этим выражением, приводится к типу строки, содержащей только это значение. (Кстати, отмечу, что даже если *rx* не сводится к простому скалярному выражению, а обозначает строку степени больше 1, то в качестве θ все равно могут выступать не только операторы «=» или «<>», хотя на практике это не рекомендуется. Более подробное обсуждение этого вопроса см. в главе 3.)

Рекомендация: Не пользуйтесь сравнениями с модификаторами ALL или ANY – они могут стать причиной ошибок, да и в любом случае того же результата можно достичь иными способами. В качестве иллюстрации первого положения примите во внимание тот факт, что в формулировке предыдущего запроса на естественном языке вполне можно было бы употребить слово *любой* вместо *каждой* – «Получить названия деталей, вес которых больше веса *любой* синей детали» – и тогда мы бы неправильно использовали сравнение >ANY вместо >ALL. А в качестве другого примера, иллюстрирующего сразу оба положения, рассмотрим такое SQL-выражение:

```
SELECT DISTINCT SNAME
FROM   S
WHERE  CITY <>ANY ( SELECT CITY FROM P )
```

Это выражение легко было бы прочитать так: «Получить названия поставщиков, для которых город местонахождения не равен городу местонахождения никакой детали», однако означает оно совсем не то. Логически это выражение эквивалентно[1] следующему запросу: «Получить названия поставщиков, для которых существует хотя бы одна деталь, находящаяся в другом городе»:

[1] А так ли это? Что если город, в котором находится поставщик или деталь, окажется равным null?

```
SELECT DISTINCT SNAME
FROM   S
WHERE  EXISTS ( SELECT *
                FROM   P
                WHERE  P.CITY <> S.CITY )
```

Результат:

SNAME
Smith
Jones
Blake
Clark
Adams

На самом деле, сравнение с модификатором ALL или ANY всегда можно преобразовать в эквивалентное выражение, содержащее EXISTS, как видно из предыдущего примера. Обычно такие сравнения можно также трансформировать в выражения, содержащие оператор MAX или MIN, потому что (к примеру) некоторое значение больше всех значений в множестве тогда и только тогда, когда оно больше максимального значения в этом множестве. А выражения с MAX и MIN часто проще для понимания, чем эквивалентные сравнения с ALL или ANY. В следующую таблицу сведены различные возможности в этом плане. (Чуть ниже я приведу пример.)

	ANY	ALL
=	IN	
<>		NOT IN
<	< MAX	< MIN
<=	<=MAX	<=MIN
>	> MIN	> MAX
>=	>=MIN	>=MAX

Замечу, в частности, что =ANY и <>ALL эквивалентны IN и NOT IN соответственно.[1] У конструкций =ALL и <>ANY нет аналогичных эквивалентов, но выражения, содержащие эти операторы, всегда можно заменить выражениями, содержащими EXISTS, как уже было сказано выше.

Предостережение: К сожалению, не гарантируется, что описанные выше трансформации (с использованием MAX и MIN) будут работать, если аргументом MAX или MIN окажется пустое множество. Причина в том, что в SQL MAX и MIN для пустого множества определены – неправиль-

[1] Так что здесь мы имеем два важных исключения из общей рекомендации избегать сравнений с модификаторами ALL или ANY; то есть допустимо на равных правах использовать =ANY и IN, а также <>ALL и NOT IN. (Лично мне кажется, что IN и NOT IN гораздо понятнее альтернатив, но выбор за вами.)

но – как null.[1] Приведу еще раз формулировку запроса «Получить названия деталей, вес которых больше веса каждой синей детали»:

```
SELECT DISTINCT PX.PNAME
FROM   P AS PX
WHERE  PX.WEIGHT >ALL ( SELECT PY.WEIGHT
                        FROM   P AS PY
                        WHERE  PY.COLOR = 'Blue' )
```

А вот ее трансформированный «эквивалент»:

```
SELECT DISTINCT PX.PNAME
FROM   P AS PX
WHERE  PX.WEIGHT > ( SELECT MAX ( PY.WEIGHT )
                     FROM   P AS PY
                     WHERE  PY.COLOR = 'Blue' )
```

Теперь предположим, что синих деталей нет. Тогда первое выражение вернет названия всех деталей в таблице P, а второе – пустой результат.[2]

Как бы то ни было, чтобы описанная в этом примере трансформация была корректной, воспользуйтесь оператором COALESCE, например, следующим образом:

```
SELECT DISTINCT PX.PNAME
FROM   P AS PX
WHERE  PX.WEIGHT > ( SELECT COALESCE ( MAX ( PY.WEIGHT ) , 0.0 )
                     FROM   P AS PY
                     WHERE  PY.COLOR = 'Blue' )
```

В качестве другого примера рассмотрим запрос «Получить названия деталей, вес которых меньше веса какой-нибудь детали, находящейся в Париже». Вот формулировка в терминах логики:

```
{ PX.PNAME } WHERE EXISTS PY ( PY.CITY = 'Paris' AND
                               PX.WEIGHT < PY.WEIGHT )
```

А вот соответствующая формулировка на SQL:

```
SELECT DISTINCT PX.PNAME
FROM   P AS PX
WHERE  EXISTS ( SELECT *
                FROM   P AS PY
                WHERE  PY.CITY = 'Paris'
                AND    PX.WEIGHT < PY.WEIGHT )
```

---

[1] Для справки: MAX для пустого множества должен быть равен минимально возможному значению, а MIN для пустого множества – максимально возможному значению (соответствующего типа).

[2] Отмечу, что в обоих случаях производится некоторое приведение типа. В качестве не вполне тривиального упражнения попробуйте разобраться в том, какие именно приведения выполняются в каждом случае.

Но этот запрос можно было бы выразить также в терминах сравнения с модификатором ANY:

```
SELECT DISTINCT PX.PNAME
FROM   P AS PX
WHERE  PX.WEIGHT <ANY ( SELECT PY.WEIGHT
                        FROM   P AS PY
                        WHERE  PY.CITY = 'Paris' )
```

Результат:

PNAME
Nut
Screw
Cam

Как видно из этого примера (и как уже было сказано ранее), выражение, содержащее сравнение с модификатором ALL или ANY, всегда можно трансформировать в эквивалентное выражение, содержащее EXISTS. Попробуйте ответить на следующие вопросы:

- Уверены ли вы, что «<ANY» – правильный оператор сравнения в данном примере? (Была ли в естественном языке употреблена фраза «меньше, чем любой»? А надо ли было сказать именно так? Вспомните также, что фраза «меньше, чем любой» отображается на «<ALL», верно?)
- Какая из представленных различных формулировок представляется вам наиболее «естественной»?
- Эквивалентны ли представленные различные формулировки в базе данных, где разрешены null-значения? А если разрешены дубликаты?

Пример 12. GROUP BY и HAVING

Как я и обещал, скажу еще несколько слов о фразах GROUP BY и HAVING. Рассмотрим такой запрос: «Для каждой детали, поставляемой не более, чем двумя поставщиками, получить номер детали, город, где она находится, и суммарное поставленное количество». Вот одна из возможных формулировок в терминах формальной логики (реляционного исчисления):

```
{ PX.PNO , PX.CITY ,
          SUM ( SPX.QTY WHERE SPX.PNO = PX.PNO , QTY ) AS TPQ }
                    WHERE COUNT ( SPY WHERE SPY.PNO = PX.PNO ) ≤ 2
```

Соответствующая формулировка на SQL:

```
SELECT PX.PNO , PX.CITY ,
              ( SELECT COALESCE ( SUM ( SPX.QTY ) , 0 )
                FROM   SP AS SPX
                WHERE  SPX.PNO = PX.PNO ) AS TPQ
```

```
FROM   P AS PX
WHERE  ( SELECT COUNT ( * )
         FROM   SP AS SPY
         WHERE  SPY.PNO = PX.PNO ) <= 2
```

Результат:

PNO	CITY	TPQ
P1	London	600
P3	Oslo	400
P4	London	500
P5	Paris	500
P6	London	100

Но, как следует из названия раздела, этот пример интересен тем, что может показаться, будто проще (и уж, точно, компактнее) выразить его с помощью фраз GROUP BY и HAVING:

```
SELECT PX.PNO , PX.CITY , COALESCE ( SUM ( SPX.QTY ) , 0 ) AS TPQ
FROM   P AS PX , SP AS SPX
WHERE  PX.PNO = SPX.PNO
GROUP  BY PX.PNO
HAVING COUNT ( * ) <= 2
```

Тогда возникают следующие вопросы:

- При такой формулировке допустимо ли появление PX.CITY в списке элементов SELECT?
- Действительно ли вам кажется, что формулировка с использованием GROUP BY / HAVING проще для понимания?
- Правильно ли работает этот запрос для деталей, которые не поставляются ни одним поставщиком (Нет, неправильно.)
- Эквивалентны ли обе формулировки, если в базе данных разрешены null-значения? А если разрешены строки-дубликаты?

В качестве дополнительного упражнения приведите формулировки на SQL, в которых (а) используются GROUP BY и HAVING, (б) не используются GROUP BY и HAVING, для следующих запросов:

- Получить номера поставщиков, которые поставляют N различных деталей для некоторого $N > 3$.
- Получить номера поставщиков, которые поставляют N различных деталей для некоторого $N < 4$.

Какие выводы вы можете сделать из этого упражнения?

Упражнения

Упражнение 11.1. Если вы еще этого не сделали, выполните упражнения, включенные в текст главы.

Упражнение 11.2. Еще раз взгляните на различные SQL-выражения в тексте главы. Сможете ли вы дать их интерпретацию на естественном языке, глядя только на SQL-формулировки (то есть не обращаясь к словесной постановке задачи)? Затем сравните свои ответы с постановкой задачи, приведенной в тексте.

Упражнение 11.3. Попытайтесь применить описанные в этой главе приемы к реальным SQL-запросам, с которыми вы сталкиваетесь на работе.

Упражнение 11.4. Некоторые примеры, обсуждавшиеся в этой главе (или очень похожие на них), встречались и в других главах, но тогда я приводил SQL-формулировки, по духу более близкие к «алгебре», нежели к «исчислению». Можете ли вы предложить правила трансформации, с помощью которых формулировки «на языке исчисления» можно было бы перевести в алгебраические и наоборот?

12

Различные вопросы, связанные с SQL

Эта последняя глава напоминает попурри; в ней обсуждается ряд аспектов SQL, которым по тем или иным причинам не нашлось места в предыдущих главах. Здесь же приводится для справки БНФ-грамматика табличных выражений SQL.

К тому же это подходящее место для определения двух терминов, к которым следует относиться с предельной настороженностью: *определяется реализацией* и *зависит от реализации*. Эти термины встречаются в стандарте SQL повсеместно. Вот их определения.

> **Определение:** Некоторое средство называется *определяемым реализацией*, если его семантика может быть разной в разных реализациях, но по крайней мере должна быть определена в каждой конкретной реализации. Иными словами, реализация свободна в выборе решения о том, как реализовать это средство, но выбранное решение должно быть документировано. Примером может служить максимальная длина строки символов.

> **Определение:** Некоторое средство называется *зависящим от реализации*, если его семантика может быть разной в разных реализациях и даже необязательно должна быть определена в конкретной реализации. Иными словами, правильнее было бы говорить о *неопределенном* поведении; реализация вправе решать, как реализовать данное средство, и выбранное решение может быть не документировано (оно может даже меняться от версии к версии). Примером может служить результат работы фразы ORDER BY, если заданная в ней спецификация не определяет полного упорядочения.

SELECT *

Применение формы «SELECT *» фразы SELECT в SQL приемлемо в ситуациях, когда ни сами столбцы, ни их порядок слева направо не существенны, например при вызове EXISTS. В других ситуациях это может быть рискованно из-за потенциального изменения смысла «*», например после добавления новых столбцов в таблицу. **Рекомендация**: будьте начеку и старайтесь избегать таких ситуаций. В частности, не используйте «SELECT *» на внешнем уровне определения курсора, задавайте имена столбцов явно. Аналогичное замечание относится и к определению представлений. *Примечание*: Если вы примете стратегию доступа к базе данных только через представления (см. обсуждение именования столбцов в главе 3), то можете без опаски использовать «SELECT *» всюду, кроме определения самих представлений.

Явные таблицы

«Явной таблицей» в SQL называется выражение вида TABLE *T*, где *T* – имя базовой таблицы или представления или «назначенное имя» (см. обсуждение WITH в главе 6). Логически оно эквивалентно выражению:

```
( SELECT * FROM T )
```

Вот довольно сложный пример использования явных таблиц («Получить все детали, но если названием города является Лондон, то показать вместо него Осло, а вес детали удвоить»):

```
WITH T1 AS ( SELECT PNO , PNAME , COLOR , WEIGHT , CITY
             FROM   P
             WHERE  CITY = 'London' ) ,
     T2 AS ( SELECT PNO , PNAME , COLOR , WEIGHT , CITY ,
                    2 * WEIGHT AS NEW_WEIGHT , 'Oslo' AS NEW_CITY
             FROM T1 ) ,
     T3 AS ( SELECT PNO , PNAME , COLOR ,
                    NEW_WEIGHT AS WEIGHT , NEW_CITY AS CITY
             FROM   T2 ) ,
     T4 AS ( TABLE P EXCEPT CORRESPONDING TABLE T1 )
TABLE T4 UNION CORRESPONDING TABLE T3
```

Квалификация имен

Имена столбцов в SQL обычно можно квалифицировать, поместив вначале имя переменной кортежа (см. следующий раздел) и точку. Однако во многих случаях SQL позволяет опускать квалификатор, и тогда по умолчанию подразумевается неявный квалификатор. Но:

- Правила SQL относительно неявной квалификации часто туманны. Поэтому не всегда очевидно, к чему относится конкретное неквалифицированное имя.

- Что однозначно сегодня, может стать неоднозначным завтра (например, если в существующую таблицу добавить новые столбцы).
- В главе 3 я настоятельно рекомендовал давать столбцам, представляющим однородную информацию, одинаковые имена. Если следовать этой рекомендации, то неквалифицированные имена часто будут оказываться неоднозначными, поэтому квалификация становится обязательной.

Поэтому примите за правило – если сомневаетесь, то квалифицируйте. К сожалению, бывают контексты, в которых квалификация не допускается. Говоря неформально, это такие контексты, в которых имя служит ссылкой на сам столбец, а не на содержащиеся в нем данные. Вот неполный перечень таких контекстов (обратите особое внимание на два последних пункта):

- Определение столбца в определении базовой таблицы
- Спецификация ключа или внешнего ключа
- Список имен столбцов (если он задан) в предложении CREATE VIEW
- Список имен столбцов (если он задан) после определения переменной кортежа (см. следующий раздел)
- «Ключи сортировки» во фразе ORDER BY
- Список имен столбцов во фразе JOIN ... USING
- Список имен столбцов (если он задан) в предложении INSERT
- Левая часть присваивания SET в предложении UPDATE

Переменные кортежа

В главе 10 мы видели, что переменная кортежа в реляционной модели – это переменная (в смысле формальной логики, а не в том смысле, который принят в языках программирования), которая «пробегает» множество строк некоторой таблицы (или, если быть точным, множество кортежей некоторого отношения). В SQL такие переменные определяются с помощью спецификации AS в контексте фразы FROM или JOIN, как показано в следующем примере:

```
SELECT SX.SNO
FROM   S AS SX
WHERE  SX.STATUS > 15
```

Здесь SX – переменная кортежа, пробегающая таблицу S; иными словами, ее допустимыми значениями являются строки таблицы S. Можно считать, что все выражение SELECT вычисляется следующим образом. Сначала переменная кортежа принимает какое-то допустимое значение, скажем, строку для поставщика S1. Значение статуса в этой строке больше 15? Если да, то номер поставщика S1 включается в результат.

Затем переменная кортежа смещается к следующей строке таблицы S, скажем, строке для поставщика S2, и снова проверяется значение статуса; если он больше 15, то номер соответствующего поставщика включается в результат. И так далее, пока переменная SX не примет последовательно каждое из допустимых значений.

Примечание

В SQL такие имена, как SX в примере выше, называются *корреляционными именами* (*correlation name*), но, похоже, что термин, обозначающий то, к чему относится такое имя, отсутствует; во всяком случае ничего, называющегося «корреляцией», в SQL нет. (Отметим, в частности, что этот термин не связан напрямую с коррелированными подзапросами, которые обсуждаются в следующем разделе.) Лично я предпочитаю термин *переменная кортежа*.

Фактически SQL требует, чтобы выражения SELECT всегда записывались в терминах переменных кортежа; если такие переменные не заданы явно, то неявно предполагается, что их имена совпадают с именами соответствующих таблиц. Например, выражение SELECT

```
SELECT SNO
FROM   S
WHERE  STATUS > 15
```

является, пожалуй, более «естественной» формулировкой приведенного выше примера и трактуется как сокращенная запись такого выражения:

```
SELECT S.SNO
FROM   S AS S
WHERE  S.STATUS > 15
```

В этой формулировке квалификатор «S» перед точкой и имя «S» в спецификации AS «AS S» обозначают не таблицу S, а переменную кортежа с именем S, которая пробегает таблицу с тем же именем.[1]

В стандарте SQL элементы списка во фразе FROM (следующего сразу же за самим ключевым словом FROM) называются *ссылками на таблицы*[2],

[1] Тут я должен признать (под нажимом), что с пониманием отношусь к замечанию своего старого друга, которое он однажды высказал как раз по этому поводу; «Вы, математики, все одинаковы – часами мучаетесь над вещами, которые всем остальным очевидны с первого взгляда».

[2] Этот термин не слишком удачен. В большинстве языков ссылка на переменную – частный случай выражения; синтаксически это просто имя переменной, обозначающее либо саму переменную, либо значение этой переменной, в зависимости от контекста. Но в SQL ссылка на таблицу не является табличным выражением – во всяком случае, не в том смысле, в котором этот термин употребляется на страницах данной книги, и (что, пожалуй, более существенно) не в смысле, в котором он используется в самом SQL.

как и табличные операнды в явном соединении JOIN (о последнем, но не о первом факте я уже упоминал в главе 6). Пусть *tr* – такая ссылка. Тогда, если табличное выражение в *tr* – это табличный подзапрос (см. следующий раздел), то *tr* также должна включать фразу AS – даже если переменная кортежа, определяемая этой фразой, больше нигде в выражении явно не упоминается. Например:

```
( SELECT SNO , CITY FROM S ) AS TEMP1
  NATURAL JOIN
( SELECT PNO , CITY FROM P ) AS TEMP2
```

Еще пример (уже приводился в главе 7):

```
SELECT PNO , GMWT
FROM ( SELECT PNO , WEIGHT * 454 AS GMWT
       FROM   P ) AS TEMP
WHERE  GMWT > 7000.0
```

Для интересующихся перепишу этот пример, сделав все неявные квалификаторы явными:

```
SELECT TEMP.PNO , TEMP.GMWT
FROM ( SELECT P.PNO , P.WEIGHT * 454 AS GMWT
       FROM   P ) AS TEMP
WHERE  TEMP.GMWT > 7000.0
```

Примечание

Определение переменной кортежа в SQL может включать необязательный список имен столбцов (определяющий имена столбцов таблицы, которую пробегает данная переменная кортежа), как в примере ниже:

```
SELECT *
FROM ( S JOIN P ON S.CITY > P.CITY ) AS TEMP
     ( A , B , C , D , E , F , G , H , I )
```

Здесь A, B, ..., I являются другими именами столбцов SNO, SNAME, STATUS, S.CITY, PNO, PNAME, COLOR, WEIGHT и P.CITY соответственно (объяснение конструкции JOIN ... ON см. в главе 6). Однако, если вы будете следовать рекомендациям, приведенным в этой книге, то часто прибегать к такой возможности не придется.

Рекомендация: Отдавайте предпочтение явному использованию переменных кортежа, особенно в «сложных» выражениях – это проясняет намерения, а иногда даже уменьшает количество ударов по клавишам. Однако помните, что правила областей видимости таких переменных в SQL понять непросто (впрочем, это в равной мере относится и к явным, и к неявным переменным).

Предостережение: Во многих учебниках по SQL имена переменных кортежа (или корреляционные имена) называются *псевдонимами* и описываются так, будто это просто альтернативные имена для таблиц, ко-

торые такие переменные пробегают. Но такое изложение неправильно представляет истинное положение вещей – и даже выдает серьезное недопонимание происходящего, – поэтому я его всячески осуждаю.

Подзапросы

Подзапросом в SQL называется табличное выражение, назовем его *tx*, заключенное в скобки; если таблицу, обозначенную выражением *tx*, назвать *t*, то таблица, обозначенная подзапросом, тоже будет *t*. Выражение *tx* не может быть явным выражением JOIN, поэтому «(SELECT * FROM A NATURAL JOIN B)» – допустимый подзапрос, а «(A NATURAL JOIN B)» – недопустимый.

Подзапросы можно разбить на три категории (хотя синтаксис во всех случаях одинаков):

- *Табличный подзапрос* – это любой подзапрос, не являющийся однострочным или скалярным.
- *Однострочный подзапрос* – это подзапрос, встречающийся в том месте, где ожидается строковое выражение. Пусть *rsq* – такой подзапрос; тогда *rsq* должен обозначать таблицу, содержащую всего одну строку. Назовем эту таблицу *t*, а ее единственную строку – *r*; тогда *rsq* ведет себя так, будто обозначает саму строку *r* (иными словами, *t* приводится к типу *r*).
- *Скалярный подзапрос* – это подзапрос, встречающийся в том месте, где ожидается скалярное выражение. Пусть *ssq* – такой подзапрос; тогда *ssq* должен обозначать таблицу, содержащую всего одну строку и всего один столбец. Назовем эту таблицу *t*, ее единственную строку – *r*, а единственное значение в этой строке – *v*; тогда *ssq* ведет себя так, будто обозначает это значение *v* (иными словами, *t* приводится к типу *r*, а затем *r* приводится к типу *v*).

В следующих примерах показаны табличный подзапрос, однострочный подзапрос и скалярный подзапрос (в этом порядке):

```
SELECT SNO
FROM   S
WHERE  CITY IN ( SELECT CITY
                 FROM   P
                 WHERE  COLOR = 'Red' )

UPDATE S
SET  ( STATUS , CITY ) = ( SELECT DISTINCT STATUS , CITY
                           FROM   S
                           WHERE  SNO = 'S1' )
WHERE  CITY = 'Paris' ;

SELECT SNO
FROM   S
```

```
WHERE  CITY = ( SELECT CITY
                FROM   P
                WHERE  PNO = 'P1' )
```

Далее, *коррелированный* подзапрос – это особый вид подзапроса (табличного, однострочного или скалярного), а именно такой, в котором есть ссылка на некоторую «внешнюю» таблицу. В следующем примере заключенное в скобки выражение после ключевого слова IN и есть коррелированный подзапрос, потому что включает ссылку на внешнюю таблицу S (запрос звучит так: «Получить названия поставщиков, которые поставляют деталь P1»).

```
SELECT DISTINCT S.SNAME
FROM   S
WHERE  'P1' IN ( SELECT PNO
                 FROM   SP
                 WHERE  SP.SNO = S.SNO )
```

В главе 11 отмечалось, что коррелированные подзапросы часто противопоказаны с точки зрения производительности (во всяком случае, концептуально), потому что их приходится вычислять для каждой строки внешней таблицы, а не раз и навсегда. (В примере выше, если выражение в целом вычисляется так, как сформулировано в запросе, то подзапрос будет вычисляться N раз, где N – количество строк в таблице S.) Поэтому лучше избегать коррелированных подзапросов, если удастся. В рассматриваемом случае этого легко добиться, переформулировав запрос следующим образом:

```
SELECT DISTINCT S.SNAME
FROM   S
WHERE  SNO IN ( SELECT SNO
                FROM   SP
                WHERE  PNO = 'P1' )
```

Наконец, «латеральный» подзапрос – это частный случай коррелированного подзапроса, а именно такой, который (а) находится внутри фразы FROM и (б) включает ссылку на «внешнюю» таблицу, которая определяется ссылкой на таблицу, встречающуюся ранее в той же фразе FROM. Рассмотрим, к примеру, запрос «Для каждого поставщика получить его номер и количество поставляемых им деталей». Вот одна из его возможных формулировок на SQL:

```
SELECT S.SNO , TEMP.PCT
FROM   S , LATERAL ( SELECT COUNT ( ALL PNO ) AS PCT
                     FROM   SP
                     WHERE  SP.SNO = S.SNO ) AS TEMP
```

Назначение ключевого слова LATERAL состоит в том, чтобы сообщить системе, что следующий за ним подзапрос коррелирован с чем-то, ранее упоминавшимся в той же фразе FROM (в данном примере подзапрос

возвращает ровно одно значение – счетчик деталей – для каждого значения SNO в таблице S). На наших тестовых данных получается такой результат:

SNO	PCT
S1	6
S2	2
S3	1
S4	3
S5	0

Кстати, здесь есть один тонкий момент, который может привести к путанице. Элементы в списке FROM – ссылки на таблицы, поэтому они обозначают таблицы. Но в примере выше ссылка на таблицу, начинающаяся с ключевого слова LATERAL, – точнее, то, что останется, если слово LATERAL убрать, – больше напоминает ссылку на скаляр или, точнее, на скалярный подзапрос; разумеется, ее можно было бы использовать в этом качестве, если бы контекст требовал такой интерпретации. Однако на самом деле это табличный подзапрос, пусть даже результатом его вычисления (для данного номера поставщика, посредством двойного приведения типа) является одно скалярное значение, которое затем вносит вклад в строку результата.

Вообще непонятно, зачем в данном случае нужен «латеральный» подзапрос. Конечно, его легко переписать в таком виде, чтобы избежать «необходимости» (?) в подобных изысках:

```
SELECT S.SNO , ( SELECT COUNT ( ALL PNO )
                 FROM   SP
                 WHERE  SP.SNO = S.SNO ) AS PCT
FROM   S
```

Подзапрос переместился из фразы FROM во фразу SELECT; он попрежнему ссылается еще на что-то в той же фразе (S.SNO, если быть точным), но ключевое слово LATERAL уже не нужно. Однако обратите внимание, что произошло со спецификацией AS PCT, которая раньше находилась внутри латерального подзапроса, а теперь вышла наружу.

И напоследок: я определил термин *подзапрос*; быть может, самое время определить заодно и термин *запрос*! – пусть даже он неоднократно использовался в предыдущих главах. Что ж, вот определение: *запросом* (*query*) называется команда выборки (retrieval request); иными словами, это табличное выражение – хотя такие выражения могут встречаться и в контекстах, не связанных с запросами, как таковыми, – или предложение, например, предложение SELECT в «прямом (то есть интерактивном) SQL», в котором требуется вычислить такое выражение. Иногда, хотя и не в этой книге, запросами называют также и команды обновления.

«Потенциально недетерминированные» выражения

Напомню (см. главу 2), что «потенциально недетерминированным» выражением в SQL называется табличное выражение, вычисление которого в разные моменты времени может давать разные значения, даже если в промежутке база данных не изменялась. Отдельно напомню, что такие выражения не разрешены в ограничениях целостности в SQL.

Опишу стандартные правила пометки табличного выражения как «потенциально недетерминированного». Пусть *tx* – табличное выражение. Тогда *tx* считается «потенциально недетерминированным», если выполнено хотя бы одно из следующих условий:

- *tx* – объединение (без ALL), пересечение или разность, и таблицы-операнды содержат столбец типа строки символов.
- *tx* – выражение SELECT, список элементов SELECT в этом выражении содержит элемент (назовем его *C*) типа строки символов, и справедливо хотя бы одно из следующих утверждений:
  - списку элементов SELECT предшествует ключевое слово DISTINCT.
  - *C* включает вызов MAX или MIN.
  - *tx* непосредственно включает фразу GROUP BY или *C* является одним из столбцов группировки.
- *tx* – выражение SELECT, которое непосредственно включает фразу HAVING, и булево выражение в этой фразе HAVING включает либо ссылку на столбец группировки типа строки символов, либо вызов MAX или MIN с аргументом типа строки символов.

Однако эти правила представляются неполными и недостаточно точными. Например:

- Объединение с ALL, безусловно, является «потенциально недетерминированным», если один из его операндов – «потенциально недетерминированное» выражение SELECT.
- Явное соединение NATURAL JOIN или JOIN USING имеет те же шансы оказаться потенциально недетерминированным, как, например, пересечение.
- Разность с ALL не может быть «недетерминированной», если таковым не является первый операнд.

Кроме того, в некоторых случаях эти правила строже, чем необходимо, например, когда в действующей схеме упорядочения задана спецификация NO PAD и в этой схеме нет символов, которые «равны, но различимы».

Пустые множества

Пустым называется множество, не содержащее ни одного элемента. Это понятие встречается в реляционной теории повсеместно и является крайне важным, но в SQL с ним связан целый ряд ошибок. К сожалению, поделать с ними вы ничего не можете, но хотя бы знать об этом нужно. Вот перечень таких ошибок (возможно, неполный):

- В выражении VALUES не допускается пустой список строковых выражений.
- Все «функции множества» в SQL возвращают null, если аргумент – пустое множество (за исключением функций COUNT(*) и COUNT, которые возвращают нуль).
- Если результатом вычисления скалярного подзапроса является пустая таблица, то она приводится к null.
- Если результатом вычисления однострочного подзапроса является пустая таблица, то она приводится к строке, состоящей из одних null-значений.
- Если множество столбцов группировки и группируемая таблица пусты, то GROUP BY порождает результат, содержащий всего одну, по необходимости пустую, группу, тогда как должен был бы порождаться результат, не содержащий групп вовсе.
- Пустое множество столбцов не может быть ключом (равно как и внешним ключом).
- Таблица не может иметь пустой заголовок.
- Список элементов SELECT не может быть пустым.
- Список элементов FROM не может быть пустым.
- Множество общих столбцов в операторах UNION, INTERSECT и EXCEPT не может быть пустым.
- Строка не может иметь пустое множество компонентов.

БНФ–грамматика табличных выражений SQL

Для справки представляется уместным завершить эту главу и основную часть книги БНФ-грамматикой табличных выражений SQL. По причинам, которые здесь не так уж важны, некоторые используемые мною термины отличаются от встречающихся в стандарте SQL; кроме того, конструкции, которые я ранее советовал не употреблять, сознательно опущены, равно как и некоторые экзотические особенности (например, рекурсия).[1] Используются следующие упрощающие сокращения:

[1] То есть грамматика несколько консервативна в том смысле, что некоторые допустимые согласно стандарту выражения не определены. Но я не думаю, что определены какие-то выражения, не допускаемые стандартом.

exp – *expression* (выражение)

ref – *reference* (ссылка)

spec – *specification* (спецификация)

Также предполагается, что следующие символы грамматики являются идентификаторами (<*identifier*>) и нигде далее не определяются:

<*table name*> (имя таблицы)

<*column name*> (имя столбца)

<*range variable name*> (имя переменной кортежа)

Неопределенными оставлены следующие символы (хотя стоит помнить о том, что одной из форм скалярного выражения является скалярный подзапрос):

<*scalar exp*>

<*boolean exp*>

Примечание

Вы можете заметить, что грамматика начинается с продукции <*top level table exp*>, хотя этот терм не упоминался в тексте книги. Я ввел эту синтаксическую категорию, чтобы отразить тот факт, что выражения соединения не могут встречаться иначе, как вложенными в какое-то другое табличное выражение[1], но это не означает, что терм <*table exp*>, определенный в грамматике, не на 100 процентов соответствует тому, что в книге называется табличным выражением! (Если быть точным, <*top level table exp*> – табличное выражение, но не <*table exp*>.) Приношу извинения, если такое положение дел кому-то покажется путаным, но это неизбежно, когда пытаешься определить грамматику языка, который по природе своей неортогонален.

```
<top level table exp>
   ::=   [ <with spec> ] <nonjoin exp>

<with spec>
   ::=   WITH <name intro commalist>

<name intro>
   ::=   <table name> AS <table subquery>

<table exp>
   ::=   [ <with spec> ] <nonjoin or join exp>

<nonjoin or join exp>
   ::=   <nonjoin exp> | <join exp>
```

[1] Это ограничение (которое упоминалось в разных местах книги без специальных пояснений) появилось в версии стандарта 2003 года; в версии 1999 года его не было.

```
<nonjoin exp>
   ::=   <nonjoin term>
       | <nonjoin exp> UNION
                 [ DISTINCT ] [ CORRESPONDING ] <table term>
       | <nonjoin exp> EXCEPT
                 [ DISTINCT ] [ CORRESPONDING ] <table term>

<table term>
   ::=   <nonjoin term>
       | <join exp>

<nonjoin term>
  ::=   <table term> INTERSECT
                 [ DISTINCT ] [ CORRESPONDING ] <table primary>
      | <nonjoin primary>

<table primary>
   ::=   <nonjoin primary>
       | <join exp>

<nonjoin primary>
   ::=   ( <nonjoin exp> )
       | TABLE <table name>
       | <table selector>
       | <select exp>

<table selector>
   ::=   VALUES <row exp commalist>

<row exp>
   ::=   <scalar exp>
       | <row selector>
       | <row subquery>

<row selector>
   ::=   ( <scalar exp commalist> )

<row subquery>
   ::=   <subquery>

<subquery>
   ::=   ( <top level table exp> )

<select exp>
   ::=   SELECT [ DISTINCT ] [ * | <select item commalist> ]
           FROM <table ref commalist>
            [ WHERE <boolean exp> ]
              [ GROUP BY <column name commalist> ]
                [ HAVING <boolean exp> ]
```

```
<select item>
   ::=   <scalar exp> [ AS <column name> ]
       | <range variable name>.*

<table ref>
   ::=   <table name> [ AS <range variable name> ]
       | [ LATERAL ] <table subquery> AS <range variable name>
       | <join exp>

<table subquery>
   ::=   <subquery>

<join exp>
   ::=   <table ref> CROSS JOIN <table ref>
       | <table ref> NATURAL JOIN <table ref>
       | <table ref> JOIN <table ref>
                     USING ( <column name commalist> )
       | ( <join exp> )
```

Упражнения

Упражнение 12.1. Какие из приведенных ниже SQL-выражений являются допустимыми термами *<top level table exp>*, а какие не являются, согласно определенной выше грамматике? (*A* и *B* – имена таблиц.)

```
A NATURAL JOIN B

A INTERSECT B

TABLE A NATURAL JOIN TABLE B

TABLE A INTERSECT TABLE B

SELECT * FROM A NATURAL JOIN SELECT * FROM B

SELECT * FROM A INTERSECT SELECT * FROM B

( SELECT * FROM A ) NATURAL JOIN ( SELECT * FROM B )

( SELECT * FROM A ) INTERSECT ( SELECT * FROM B )

( TABLE A ) NATURAL JOIN ( TABLE B )

( TABLE A ) INTERSECT ( TABLE B )

( TABLE A ) AS AA NATURAL JOIN ( TABLE B ) AS BB

( TABLE A ) AS AA INTERSECT ( TABLE B ) AS BB

( ( TABLE A ) AS AA ) NATURAL JOIN ( ( TABLE B ) AS BB )

( ( TABLE A ) AS AA ) INTERSECT ( ( TABLE B ) AS BB )
```

Быть может, следует напомнить, что с точки зрения реляционной теории, пересечение – это частный случай естественного соединения. Какие выводы вы можете сделать из этого упражнения?

Упражнение 12.2. Исследуйте какую-нибудь доступную вам SQL-систему. Поддерживаются ли в ней (а) выражения UNIQUE, (б) явные

таблицы, (в) латеральные подзапросы, (г) «потенциально недетерминированные» выражения?

Упражнение 12.3. На протяжении всей книги я понимал под термином *SQL* официальную версию стандарта этого языка (хотя сознательно не рассматривал стандарт во всей полноте). Однако любой представленный на рынке продукт в той или иной мере отклоняется от стандарта, как опуская некоторые определенные в нем средства, так и добавляя свои собственные расширения. Исследуйте любой доступный вам продукт на основе SQL. Найдите столько отклонений от стандарта, сколько сможете. (Предыдущее упражнение могло дать кое-какие идеи в этом направлении.)

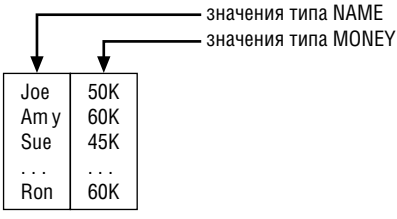
A

Реляционная модель

Я глубоко убежден, что если задуматься над этим вопросом на надлежащем уровне абстракции, то неизбежно придешь к выводу, что базы данных должны быть реляционными. Позвольте мне сразу же попытаться обосновать это весьма сильное утверждение! Я рассуждаю следующим образом.

- В главе 5 я объяснил, что база данных, несмотря на название, по существу является не совокупностью данных, а совокупностью фактов, или, иначе говоря, *истинных высказываний*, например: «зарплата Джо составляет 50K долларов в год».
- Высказывания типа «зарплата Джо составляет 50K долларов в год» легко кодируются в виде *упорядоченных пар* – в данном случае (Джо, 50K), где «Джо» – значение, скажем, типа NAME, а «50K» – значение типа MONEY (например).
- Но мы хотим записывать не все вообще высказывания, а только те, которые являются порождениями некоторых *предикатов*, принимающими значение «истина». Так, в случае высказывания «зарплата Джо составляет 50K долларов в год» предикатом является «зарплата x составляет y долларов в год», где x – значение типа NAME, а y – значение типа MONEY.
- Иными словами, мы хотим записывать *экстенцию* предиката «зарплата x составляет y долларов в год» и можем это сделать в виде множества упорядоченных пар.
- Но множество упорядоченных пар – это и есть бинарное отношение в математическом смысле слова. Вот его определение: бинарным отношением над двумя множествами A и B называется подмножество декартова произведения A и B; иначе говоря, множество упорядоченных пар (a,b) таких, что первый элемент a взят из A, а второй элемент b взят из B.

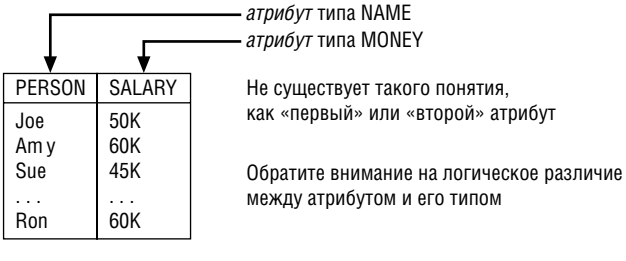
- Бинарное отношение в указанном выше смысле можно изобразить в виде *таблицы*, например:



(Кстати говоря, в этом конкретном примере мы имеем не просто отношение, а функцию, поскольку у каждого человека есть только одна зарплата. Функция – частный случай бинарного отношения.) Таким образом, мы можем считать, что на этом рисунке изображено подмножество декартова произведения множества всех имен («тип NAME») и множества всех денежных величин («тип MONEY»), именно в таком порядке.

Уже это рассуждение наводит на мысль, что у нас есть пока довольно скромные (но весьма основательные) зачатки чего-то. Однако в 1969 году Кодд осознал, что:

- Нам следует рассматривать *n-местные*, а не только двуместные предикаты и высказывания (например, «Джо имеет зарплату 50K, работает в отделе D4 и был принят на работу в 1993 году»). Следовательно, нужно работать с *n-арными*, а не только бинарными, отношениями и *n-кортежами* (для краткости просто *кортежами*), а не только с упорядоченными парами.
- Упорядочение слева направо, возможно, и приемлемо для пар, но при $n > 2$ очень быстро становится неудобным; поэтому мы заменяем понятие упорядоченности понятием атрибутов (идентифицируемых по имени) и соответственно переопределяем понятие отношения. Наш пример теперь выглядит следующим образом:



Начиная с этого места, можете считать, что термин *отношение* обозначает отношение в описанном выше смысле, если явно не оговорено противное.

- Но на представлении данных история не заканчивается – нам нужны *операторы* для порождения новых отношений из имеющихся («базовых»), чтобы можно было предъявлять запросы и т. п. (например, «Получить всех людей с зарплатой больше 60K»). Но так как отношение одновременно является и логической конструкцией (экстенция предиката), и математической (частный случай множества), то мы может применять к нему как логические, так и математические операторы. Именно поэтому Кодд и сумел определить как *реляционное исчисление* (основанное на формальной логике), так и *реляционную алгебру* (основанную на теории множеств). Так и родилась реляционная модель.

Реляционная и другие модели

Теперь вы, наверное, понимаете, почему я считаю (повторяя мысль, высказанную в главе 5), что реляционная модель покоится на твердых основаниях, «правильна» и выдержит испытание временем. Я твердо уверен, что и через сто лет системы баз данных будут основаны на реляционной модели Кодда. Почему? Потому что основания этой модели, а именно теория множеств и логика предикатов, сами прочны, как камень. В частности, элементы логики предикатов восходят еще к Аристотелю, который жил больше 2000 лет назад (384–322 годы до нашей эры).

Ну а что сказать о других моделях – к примеру, «объектно-ориентированной модели», или «иерархической модели», или «сетевой модели» CODASYL, или «полуструктурированной модели»? На мой взгляд, они просто находятся в другой весовой категории. Да, я серьезно задаю вопрос, а заслуживают ли они вообще названия модели.[1] Иерархическая и сетевая модели в особенности вообще никогда не существовали! – как абстрактные модели, я хочу сказать, предшествующие конкретным реализациям. Они были изобретены уже *постфактум*, то есть сначала были созданы иерархические и сетевые системы, а уже потом определены соответствующие модели – по индукции, а это лишь вежливое слово для обозначения догадки. Что касается объектно-ориентированной и полуструктурированной модели, то не исключено, что и к ним приложима та же критика; подозреваю, что так и есть, хотя полной уверенности у меня нет. Одна из проблем заключается в том, что отсутствует единое мнение о составе таких моделей. Например, безусловно, нельзя утверждать, что существует единая, четко определенная и всеми признанная объектно-ориентированная модель, и то же самое относится к полуструктурированной модели. (Впрочем, кто-то может сказать, что единой реляционной модели тоже нет, но к этому аргументу я вернусь чуть позже.)

---

[1] Именно поэтому я и заключил их названия в кавычки. Начиная с этого места, я буду опускать кавычки, поскольку понимаю, какое они вызывают раздражение, но все равно считайте, что в некотором виртуальном смысле они присутствуют.

Есть и еще одна важная причина, почему я не верю в то, что другие модели заслуживают названия моделей. Во-первых, полагаю, никто не станет отрицать, что реляционная модель – это действительно модель и, следовательно, по определению, не связана с деталями реализации. Напротив, все остальные модели в большинстве случаев не могут провести четкое различие между аспектами, относящимися собственно к модели, и вопросами реализации; в лучшем случае они высказываются об этом различии очень путано (все они, так сказать, «гораздо ближе к железу»). Как следствие, они труднее для понимания и использования и дают разработчикам гораздо меньше свободы (меньше, чем реляционная модель) в придумывании неординарных, творческих подходов к вопросам реализации.

А как быть с заявлениями о существовании каких-то других реляционных моделей? Например, Джо Селко (Joe Celko) автор книги «Data and Databases: Concepts in Practice» (Morgan Kaufmann, 1999) пишет:

> Говорить о существовании *единой* реляционной модели баз данных не больше оснований [sic], чем о существовании только одной геометрии.

А в обоснование этого утверждения он далее описывает шесть «различных реляционных моделей».

Я отозвался на эти заявления сразу, как только их увидел. Вот отредактированный вариант того, что я тогда написал:

> Да, действительно существует несколько различных геометрий (евклидова, эллиптическая, гиперболическая и т. д.). Но можно ли эту аналогию назвать корректной? То есть различаются ли «различные реляционные модели» так же, как различаются различные геометрии? Мне кажется, что ответ на этот вопрос *отрицателен*. Эллиптическую и гиперболическую геометрии часто явно называют *неэвклидовыми*; поэтому, чтобы аналогия была полной, по крайней мере, пять из «шести различных реляционных моделей» должны были бы быть *нереляционными*, следовательно, по определению, не «реляционными моделями». (Фактически, я бы согласился с тем, что некоторые из «шести различных реляционных моделей» действительно не реляционные. Но тогда трудно утверждать – по крайней мере, не впадая в противоречие, – что это различные *реляционные* модели.)

И далее (опять-таки текст немного отредактирован):

> Но я должен признать, что Кодд действительно пересматривал собственные определения реляционной модели в 1970–1980-е годы. Это дало основание критикам обвинить Кодда в частности и сторонников реляционной теории в целом в том, что они меняют правила по ходу игры. Например, Майк Стоунбрейкер (Mike Stonebraker) писал (во введении к книге «Readings in Database Systems», второе издание, Morgan Kaufmann, 1994), что «может представить себе четыре разные версии» модели:

- Версия 1: определена в статье, опубликованной в журнале CACM в 1970 году
- Версия 2: определена в статье 1981 года, представленной на присуждение премии Тьюринга
- Версия 3: определена в 12 правилах и оценочной системе Кодда
- Версия 4: определена в книге Кодда

Тут я хотел бы прерваться и объяснить, на что ссылается Стоунбрейкер. Все это работы Кодда. В 1970 году в журнале CACM – CACM 13, No. 6 (June 1970) – была опубликована статья «Реляционная модель данных для больших разделяемых банков данных» (*A Relational Model of Data for Large Shared Data Banks*). На присуждение премии Тьюринга была представлена статья «Relational Database: A Practical Foundation for Productivity», CACM 25, No. 2 (February 1982). 12 правил и соответствующая им оценочная система описаны в статьях Кодда для журнала Computerworld: «*Is Your DBMS Really Relational?*» и «*Does Your DBMS Run By The Rules?*» (14 и 21 октября 1985). И, наконец, под книгой Кодда разумеется книга «*The Relational Model For Database Management Version 2*» (Addison-Wesley, 1990). А теперь я вернусь к своему ответу.

> Вероятно, потому, что нас немного задела такая критика, мы с Хью Дарвеном постарались привести в нашей совместной книге «Databases, Types, and the Relational Model: The Third Manifesto»[1] свою собственную аккуратную формулировку того, что такое реляционная модель (или чем она должна быть!). Действительно, мы хотели, чтобы «Манифест» отчасти рассматривался как такая авторитетная формулировка. За подробностями отсылаю к самой книге; здесь же хотелось бы только сказать, что мы видим свой вклад в этой области, прежде всего, в расстановке всех точек над i, которые не были расставлены должным образом в работах самого Кодда. Мы нисколько не отклонились от видения Кодда в каких-либо существенных аспектах; весь «Манифест» составлен в духе идей Кодда и следует по проторенной им дороге.

Ко всему сказанному выше я хотел бы добавить еще одно соображение, которое, как мне кажется, убедительно опровергает доводы Селко. Я согласен, что существует несколько разных геометрий. Но почему они различны? Потому что в основу положены разные аксиомы. Напротив, аксиоматика реляционной модели никогда не изменялась. Да, на протяжении нескольких лет мы внесли в саму модель ряд модификаций – например, добавили реляционные сравнения, – но аксиомы (по существу, те же, что в классической логике предикатов) оставались неизменными со времен первых работ Кодда. Добавлю, что те изменения, которые вносились, имели, на мой взгляд, эволюционный, а не революционный характер. Поэтому я с уверенностью заявляю, что существует только одна реляционная модель, хотя она и развивалась со временем и,

[1] См. приложение D к настоящей книге.

вероятно, будет развиваться и дальше. В главе 1 я отмечал, что реляционную модель можно считать небольшой ветвью математики, и, следовательно, она расширяется по мере доказательства новых теорем и появления новых результатов.

Так каковы же эти эволюционные изменения? Вот некоторые из них:

- Как уже отмечалось, мы добавили реляционные сравнения.
- Мы уточнили логические различия между отношениями и переменными-отношениями.
- Мы уточнили понятие первой нормальной формы и попутно ввели понятие атрибутов с отношениями в качестве значений (RVA-атрибутов).
- Мы стали лучше понимать природу реляционной алгебры, в том числе сравнительную значимость различных операторов и важность отношений степени 0, а также идентифицировали несколько новых операторов (например, расширения и полусоединения).
- Мы добавили понятие отношений-образов.
- Мы стали лучше понимать, что такое обновление, в частности, обновление представлений.
- Мы стали лучше понимать фундаментальную значимость ограничений целостности вообще и получили немало хороших теоретических результатов, касающихся некоторых важных частных случаев.
- Мы прояснили природу взаимосвязей между моделью и логикой предикатов.
- Наконец, мы стали лучше понимать взаимосвязи между реляционной моделью и теорией типов (точнее, мы прояснили природу доменов).

Определение реляционной модели

Теперь я хотел бы дать точное определение того, что входит в состав реляционной модели. Беда в том, что мое определение действительно точное; настолько точное, что его трудно было бы понять, приведи я его в главе 1. (Как однажды афористично заметил Бертран Рассел: «Книга должна быть либо ясной, либо строгой, совместить эти два требования невозможно»). Вообще-то, некое определение я все же в первой главе дал – определение того, что назвал «оригинальной моделью», – но, честно говоря, не думаю, что его хотя бы отдаленно можно назвать хорошим, в том числе по следующим причинам:

- Для начала оно слишком длинное и бессвязное. (Что, впрочем, было неплохо, учитывая назначение этой вводной главы, но теперь я хочу иметь лаконичное и точное определение.)

- Мне совершенно не по душе идея, что модель следует представлять себе как «структуру плюс целостность плюс средства манипулирования»; я даже думаю, что в некоторых отношениях рассуждать о ней в таких терминах – только запутывать себя.
- В «оригинальной модели» было несколько вещей, которые мне не нравятся, например: операция деления, null-значения, правило целостности сущностей, мысль о том, что какой-то ключ нужно обязательно делать первичным, и представление о том, что домены и типы в чем-то различаются. Кстати, о null-значениях: замечу, что Кодд определил реляционную модель в 1969 году, а null-значения добавил только в 1979. Иными словами, в течение десяти лет модель отлично себя чувствовала – на мой взгляд, даже лучше – безо всяких null. Более того, ранние реализации тоже обходились без них.
- В оригинальной модели было упущено несколько существенных, на мой взгляд, вещей. Например, не было упоминаний, по крайней мере явных, о предикатах, ограничениях (помимо ограничений ключа и внешнего ключа), переменных-отношениях, реляционных сравнениях, выводе типа отношения и связанных с этим механизмом средствах, отношениях-образах, некоторых алгебраических операторах (особенно переименования, расширения, обобщения, полусоединения и полуразности) и важных отношениях TABLE_DUM и TABLE_DEE. (С другой стороны, будет справедливо добавить, что все эти средства и не запрещались оригинальной моделью, можно даже сказать, что некоторые из них даже были включены, в зачаточном виде. Например, безусловно, с самого начала предполагалось, что реализация должна поддерживать не только ограничения ключа и внешнего ключа, но и другие. Реляционные сравнения тоже, по меньшей мере, неявно входили в состав требований.)

А теперь позвольте мне, не мудрствуя лукаво, дать собственное определение.

> **Определение:** Реляционная модель состоит из пяти компонентов:
>
> - Расширяемый набор скалярных типов, включающий, в частности, тип BOOLEAN;
> - Генератор типов отношений и подразумеваемая интерпретация отношений сгенерированных им типов;
> - Средства для определения переменных-отношений, принадлежащих сгенерированным типам отношений;
> - Оператор реляционного присваивания для присваивания отношений-значений переменным-отношениям;
> - Расширяемый набор обобщенных реляционных операторов для порождения новых отношений-значений из существующих.

В следующих подразделах эти компоненты рассматриваются более подробно.

Скалярные типы

В общем случае скалярные типы могут быть системными или определенными пользователем; следовательно, должны быть предоставлены средства для определения новых типов пользователями (это требование отчасти вытекает из того, что множество скалярных типов расширяемо). Но тогда должны существовать доступные пользователям средства для определения скалярных операторов, поскольку типы без операторов бесполезны. Множество системных скалярных типов должно включать тип BOOLEAN – самый фундаментальный из всех, – но реальная система наверняка будет поддерживать и другие (INTEGER, CHAR и так далее).[1]

Из поддержки типа BOOLEAN вытекает поддержка обычных логических операторов (NOT, AND, OR и так далее), а также других операторов, системных или определенных пользователями, которые возвращают булевы значения. В частности, для каждого типа, скалярного или нескалярного, должен быть определен оператор сравнения на равенство «=», поскольку без него мы даже не могли бы сказать, каковы значения, составляющие данный тип. Более того, модель предписывает семантику такого оператора. Точнее, если *v1* и *v2* – значения одного и того же типа, то *v1* = *v2* возвращает TRUE, если *v1* и *v2* – одно и то же значение, и FALSE в противном случае.

Типы отношений

Генератор типов отношений позволяет пользователям определять типы отдельных отношений по своему желанию: в частности, как тип некоторой переменной-отношения или некоторого RVA-атрибута. Предполагаемая интерпретация данного отношения данного типа в данном контексте – множество высказываний; каждое такое высказывание (а) составляет порождение некоторого предиката, который соответствует заголовку отношения, (б) представлено кортежем в теле отношения и (в) предполагается истинным. Если контекстом служит некоторая переменная-отношение, то есть если мы говорим об отношении, которое является текущим значением какой-то переменной-отношения, то предикатом будет предикат этой переменной-отношения. Переменные-отношения согласуются с *допущением замкнутости мира* (см. ниже в этом приложении).

---

[1] Должен напомнить, что (а) различие между скалярными и нескалярными типами по необходимости несколько размыто и (б) поэтому реляционная модель не полагается на формальное различие между ними. Более подробное обсуждение см. в главе 2.

Пусть *RT* – тип отношения. Тогда с *RT* ассоциированы оператор-селектор отношения со следующими свойствами: (а) каждый вызов этого оператора возвращает отношение типа *RT* и (б) любое отношение типа *RT* возвращается некоторым вызовом этого оператора. Кроме того, так как для любого типа определен оператор сравнения на равенство «=», то он определен и для типа *RT*. Также определен оператор реляционного включения («⊆»); если отношения *r1* и *r2* одного и того же типа, то *r1* включается в *r2* тогда и только тогда, когда тело *r1* является подмножеством *r2*.

Переменные-отношения

Как отмечалось в предыдущем подразделе, наиболее важным применением генератора типов отношений является задание типа переменной-отношения в момент ее определения. Только такие переменные и разрешены в реляционной базе данных; в частности, скалярные переменные и переменные-кортежи запрещены. (Напротив, в программах, обращающихся к такой базе данных, они не запрещены, а, скорее всего, даже необходимы.)

Утверждение о том, что база данных не содержит ничего, кроме переменных-отношений, – одна из возможных формулировок того, что Кодд называл *принципом информации*, хотя я не думаю, что сам он такой формулировкой пользовался. Обычно он формулировал этот принцип следующим образом:

> *Все информационное содержимое базы данных в любой момент времени представлено одним и только одним способом, а именно в виде явных значений в позициях атрибутов кортежей отношений.*

Я не раз слышал, как Кодд называл этот принцип фундаментальным принципом реляционной модели. Стало быть, любое его нарушение следует считать криминалом. Таблицы базы данных, подразумевающие упорядочение строк сверху вниз или упорядочение столбцов слева направо, содержащие строки-дубликаты, или указатели, или null-значения, или имеющие безымянные столбцы либо столбцы с повторяющимися именами, – все это нарушения принципа информации. Но почему этот принцип так важен? Ответ вытекает из сделанных в главе 5 замечаний о том, что отношения (наряду с типами) необходимы и достаточны для представления любых данных на логическом уровне. Иными словами, реляционная модель дает все, что необходимо в этом плане, и не дает ничего лишнего.

Я хотел бы еще немного задержаться на этой мысли. Вообще говоря, можно считать аксиомой, что если у нас есть *n* разных способов представления данных, то необходимо *n* разных наборов операторов. Например, если бы помимо отношений были еще и массивы, то потребовался бы полный набор операторов с массивами и полный набор операторов с отношениями. Если *n* больше единицы, то возникает больше

операторов, которые нужно реализовывать, документировать, преподавать, изучать, запоминать и использовать (да еще и выбирать нужный). Однако дополнительные операторы лишь увеличивают сложность, но не добавляют выразительных возможностей! Нет ничего полезного, что можно было бы сделать лишь при $n > 1$ и нельзя при $n = 1$ (а в реляционной модели n равно 1).

Более того, реляционная модель не только дает нам всего одну конструкцию, само отношение, для представления данных, но и эта конструкция – цитируя Кодда (см. следующий раздел) – отличается *спартанской простотой*: в ней нет упорядочения столбцов, нет упорядочения атрибутов, нет кортежей-дубликатов, нет указателей и (по крайней мере, с моей точки зрения) нет null-значений. Любое отклонение от этих свойств равносильно введению еще одного способа представления данных, а, значит, и добавочных операторов. И SQL – живое доказательство этого замечания; так, в SQL аж восемь разных операторов объединения[1], тогда как в реляционной модели – всего один.

Как видите, *принцип информации* действительно важен, но следует сказать, что такое название ему только вредит. Были предложены и другие названия, в основном Хью Дарвеном, мной или нами совместно, в том числе *принцип однородного представления* или *принцип однородности представления*. (Признаю, что последнее звучит коряво, но оно все-таки более точно.)

Остался еще один момент, который я хотел бы упомянуть в разделе «Переменные-отношения». Дарвен и я продемонстрировали в *Третьем манифесте*, что база данных – не просто «контейнер для переменных-отношений», несмотря на то, что именно так мы ее воспринимали на протяжении всей книги. Логически база данных сама является переменной (пусть и довольно большой), которую можно было бы назвать *db-переменной*. Таким образом, «переменные-отношения» на самом деле переменными не являются; скорее, их следовало бы назвать, как в *Третьем манифесте*, *псевдопеременными*. Их назначение – создать у пользователя иллюзию, будто он может обновлять базу данных – точнее, db-переменную – по частям, а не целиком (точно так же, как мы говорим об обновлении отдельного кортежа переменной-отношения, а не об обновлении переменной-отношения целиком).

---

[1] Более того, по чести их должно было бы быть двенадцать, а не восемь – имеющийся в SQL оператор «объединения мультимножеств», применяемый к «мультимножествам строк», которые допустимы в качестве значений столбцов таблицы, поддерживает лишь два из шести режимов, поддерживаемых оператором объединения обычных таблиц. И при этом SQL все равно не поддерживает настоящий оператор объединения мультимножеств; то, что в SQL называется «объединением мультимножеств», соответствует встречающемуся в литературе понятию «объединение+». Дополнительную информацию см. в статье «The Theory of Bags: An Investigative Tutorial» (упомянутой в приложении D).

Реляционное присваивание

Как и оператор сравнения на равенство «=», оператор присваивания «:=» должен быть определен для любого типа, поскольку без него мы не смогли бы присваивать значения переменным этого типа. И, повторю еще раз, типы отношений – не исключение. Сокращения INSERT, DELETE и UPDATE разрешены и полезны, но, строго говоря, являются лишь сокращениями. Более того, поддержка реляционного присваивания должна (а) включать и поддержку множественного реляционного присваивания и (б) соответствовать *принципу присваивания* и **золотому правилу**.

Реляционные операторы

«Обобщенные реляционные операторы» как раз и составляют реляционную алгебру, поэтому встроены (хотя нет принципиальных причин, по которым пользователю нельзя определять дополнительные операторы, если он того хочет). Какие именно операторы включать, не указано, но требуется, чтобы совместно они обладали не меньшей выразительной мощностью, чем реляционное исчисление; другими словами, набор операторов должен быть *реляционно полным* (см. обсуждение ниже).

Похоже, существует широко распространенное заблуждение относительно целей алгебры. Точнее, многие полагают, что она предназначена лишь для написания запросов. Но это не так, задача алгебры – написание реляционных выражений. Эти выражения служат многим разным целям, в том числе и написанию запросов, но одним лишь запросами дело не ограничивается. Вот еще несколько важных целей:

- Определение представлений и снимков
- Определение множества кортежей, которые нужно вставить в некоторую переменную-отношение, удалить или обновить (или, более общо, определение множества кортежей, которое нужно присвоить какой-то переменной-отношению)
- Определение ограничений (хотя в этом случае реляционное выражение является лишь подвыражением некоторого булевого выражения; часто, но не всегда, вызова IS_EMPTY)
- Основа для исследований в других областях, например, оптимизации и проектирования баз данных

И так далее (список неполный).

Алгебра также служит эталоном, с которым сравнивается выразительная мощность языков баз данных. Говорят, что язык *реляционно полный*, если он не менее выразителен, чем алгебра (или исчисление – что одно и то же), то есть с помощью его выражений можно определить любое отношение, которое может быть определено с помощью выражений реляционной алгебры. Реляционная полнота – основная мера вырази-

тельных возможностей языка; если язык реляционно полный, то (среди прочего и допуская некоторую вольность речи) на нем можно выразить запросы произвольной сложности, не прибегая к циклам или рекурсии. Другими словами, именно реляционная полнота позволяет пользователям – по крайней мере, в принципе, хотя, быть может, и не на практике – обращаться к базе данных напрямую, минуя бутылочное горлышко ИТ-отдела.

Цели реляционной модели

Мне кажется, что хотя бы для справки в этом приложении уместно перечислить те цели, которые сам Кодд сформулировал во введении к реляционной модели. Следующий список основан на том, что он привел в работе «Recent Investigations into Relational Data Base Systems» (доклад, который его пригласили прочесть на конгрессе IFIP в 1974 году), но я его слегка подредактировал:

1. Обеспечить высокую степень независимости от данных.
2. Обеспечить единый взгляд на данные, отличающийся спартанской простотой, чтобы широкие круги пользователей в организации – от ничего не понимающих в компьютерах до самых квалифицированных – могли иметь дело с общей моделью (не запрещая накладывать поверх нее пользовательские представления для специальных целей).
3. Упростить потенциально очень трудную работу администратора базы данных.
4. Заложить теоретические основы, пусть скромные, управления базами данных (область, которой, увы, недостает твердых принципов, которыми можно было бы руководствоваться).
5. Объединить дисциплины извлечения фактов и управления файлами, готовясь в будущем добавить службы вывода знаний для использования в коммерческих приложениях.
6. Поднять программирование баз данных на новый уровень – где множества (а точнее, отношения) рассматриваются как операнды, а не обрабатываются поэлементно.

Оставляю вам судить, до какой степени, на ваш взгляд, реляционная модель отвечает поставленным целям. Мне кажется, что отвечает – и неплохо.

Некоторые принципы баз данных

В главе 11 я сказал, что меня интересуют принципы, а не продукты, и с несколькими такими принципами мы уже встречались на страницах этой книги. Теперь я соберу их в одном месте, чтобы было проще

ссылаться. *Примечание*: Перечень не претендует на полноту. В частности, я опустил принципы нормализации и другие принципы проектирования баз данных, потому что в тексте книги они не рассматривались (см., однако, приложение В).

Принцип информации (известный также под названием *принципа однородного представления,* или *принципа однородности представления*)

База данных не содержит ничего, кроме отношений. Эквивалентно, все информационное содержимое базы данных в любой момент времени представлено одним и только одним способом, а именно в виде явных значений в позициях атрибутов кортежей отношений.

Допущение замкнутости мира

Если кортеж t в какой-то момент времени оказывается в переменной-отношении R, то предполагается, что в этот момент истинно высказывание p, соответствующее t. Обратно, если кортеж t мог бы оказаться в переменной-отношении R в какой-то момент времени, но не оказался, то предполагается, что в этот момент высказывание p, соответствующее t, ложно.

Принцип взаимозаменяемости

Не должно существовать произвольных, ненужных различий между базовыми и виртуальными переменными-отношениями.

Принцип присваивания

После присваивания значения v переменной V результатом сравнения $V = v$ должно быть TRUE.

Золотое правило

Никакая операция обновления ни при каких обстоятельствах не должна приводить к тому, что хотя бы одно ограничение базы данных примет значение FALSE.

Принцип тождества неразличимых

Пусть a и b – любые два предмета (или, если вам так больше нравится, «сущности»). Тогда если не существует никакого способа различить a и b, то это не два предмета, а всего один.[1] *Примечание*: Раньше я не упоминал этот принцип, но молчаливо обращался к нему во многих случаях. Его можно сформулировать и по-другому: *у каждого предмета есть уникальная характеристика, определяющая его индивидуальность*. В реляционной модели такие характеристики представляются так же, как все остальное: с помощью значений атрибутов (см. *принцип информации* выше), и из этого простого факта вытекают многочисленные полезные следствия.

[1] Вот вам и еще одна причина – философская, если хотите, – отвергнуть понятие дубликата.

Что осталось сделать?

Все сказанное выше не означает, что мы можем почивать на лаврах, так как вся работа сделана. На самом деле, я вижу по меньшей мере четыре области, в какой-то мере взаимосвязанных, где уже ведутся или необходимы исследования: реализация, основания, высокоуровневые абстракции и высокоуровневые интерфейсы.

Реализация

В каком-то смысле идею этой книги можно свести к очень простому призыву:

Давайте реализуем реляционную модель!

Полагаю, из текста книги стало ясно, что называть SQL реляционным языком значит произносить в его адрес ничем не заслуженный комплимент. Отсюда следует, что все продукты на основе SQL можно считать реляционными только в первом приближении. В действительности реляционная модель так и не была надлежащим образом реализована в коммерческом виде, и у пользователей не было возможности насладиться всеми благами, которые несет с собой по-настоящему реляционная система. Собственно, это одна из главных причин, по которым я взялся писать эту книгу, а также одна из причин, по которым мы с Хью Дарвеном так долго работали над *Третьим манифестом*. *Третий манифест* – для краткости просто *Манифест* – это формальное предложение по созданию солидного основания будущих СУБД. Без слов понятно, что его назначение – определить реляционную модель настолько аккуратно и точно, насколько удалось авторам, и сформулировать некоторые следствия такого определения. (Там же подробно рассматривается влияние теории типов на модель и, в частности, предлагается полная модель наследования типов, логически вытекающая из теории типов.)

Нам бы очень хотелось увидеть, что идеи *Манифеста* корректно реализованы в коммерческой форме (говоря «мы», я имею в виду Дарвена и себя).[1] Мы полагаем, что такая реализация могла бы стать прочным фундаментом для возведения многих других построек, например: «объектно-реляционных» СУБД, пространственно-временных СУБД, систем, необходимых для Всемирной Паутины, и «машин правил» (известных также под названием «серверы бизнес-логики»), которые некоторым представляются следующим поколением СУБД общего назначения. Мы также полагаем, что в этом случае у нас появилась бы необходимая инфраструктура для поддержки других вещей, которые ниже

[1] В этой связи добавлю, что нам хотелось увидеть реализацию, которая в некоторых отношениях была бы более технически изощренной, чем современные реализации SQL. Конкретно, хотелось бы увидеть реализацию, основанную на так называемой *Трансреляционной™ Модели* (см. раздел «Некоторые замечания о физическом проектировании» в приложении В).

названы желательными. Лично я пошел бы и дальше; рискну предположить, что попытка реализовать все это на базе любой другой инфраструктуры окажется более трудной, чем если делать правильно. Процитирую известного математика Григория Чудновского: «Если сделать по-дурному, потом придется переделывать» (из статьи в газете *New York Times* от 24 декабря 1997 года).

Основания

В области теоретических основ есть еще много интересной работы (иными словами, ни в коем случае нельзя сказать, что все фундаментальные задачи решены). Вот три примера.

- Пусть *rx* – реляционное выражение. По определению, отношение *r*, обозначаемое *rx*, удовлетворяет ограничению *rc*, производному от ограничений, которым удовлетворяют отношения, в терминах которых выражено *rx*. В какой мере ограничение *rc* можно вычислить?
- Можно ли сделать процесс проектирования баз данных в большей степени наукой, чем он сейчас является? В частности, можно ли дать точную характеристику понятию избыточности?
- Можно ли придумать хороший способ – то есть надежный, логически обоснованный и эргономически приемлемый – решения проблемы «отсутствующей информации»?

Высокоуровневые абстракции

Один из способов добиться прогресса в компьютерных языках и приложениях состоит в том, чтобы *повысить уровень абстракции*. Например, в главе 5 я отметил, что всем известные спецификации KEY и FOREIGN KEY – в действительности лишь сокращенная запись ограничений, которые можно выразить более пространно с помощью общих средств обеспечения целостности, имеющихся в любом реляционно полном языке, например **Tutorial D**. Но эти сокращения полезны: не говоря уже том, что они уменьшают объем ввода, они позволяют поднять уровень абстракции и рассуждать в терминах некоей группы естественно взаимосвязанных понятий. В каком-то смысле они позволяют увидеть лес за деревьями.

В качестве другого примера рассмотрим реляционную алгебру. В главах 6 и 7 я показал, что многие операторы реляционной алгебры – в том числе и те, которыми мы постоянно пользуемся, даже не подозревая об этом, например полусоединение, – на самом деле являются сокращенной записью некоторых комбинаций других операторов.[1] Кстати, есть

[1] Фактически Дарвен и я в нашем *Манифесте* показываем, что любой алгебраический оператор, рассмотренный в настоящей книге, можно выразить в терминах двух примитивов: *remove* (по существу, проекция) и либо *nand*, либо *nor*.

и другие полезные операторы, которые я здесь вообще не рассматривал из-за нехватки места, и для них эти замечания в каком-то смысле «даже более справедливы». И в этом случае имеет место повышение уровня абстракции (похоже на то, как макросы повышают уровень абстракции в традиционном языке программирования).

Повышение уровня абстракции в реляционном мире можно считать возведением новых этажей на фундаменте реляционной модели; сама модель не изменяется, но становится лучше приспособленной к конкретным задачам. Особенно плодотворным такой подход, похоже, окажется в применении к темпоральным базам данных. В нашей совместной книге «Temporal Data and the Relational Model» (см. приложение D) Хью Дарвен, Никос Лоренцос (Nikos Lorentzos) и я, основываясь на оригинальной работе Лоренцоса, вводим *интервальные типы* как основу для поддержки темпоральных данных в реляционной инфраструктуре. Взгляните, например, на «темпоральное отношение» на рис. A.1, где показано, что некоторые поставщики поставляют некоторые детали на протяжении определенных промежутков времени (можно интерпретировать *d04* как «день 4» *d06* как «день 6» и т. д.; аналогично [*d04:d06*] можно интерпретировать как «интервал между днем 4 и днем 6 включительно»). Значениями атрибута DURING в этом отношении являются интервалы.

SNO	PNO	DURING
S1	P1	[d04:d06]
S1	P1	[d09:d10]
S1	P3	[d05:d10]
S2	P1	[d02:d04]
S2	P1	[d08:d10]
S2	P2	[d03:d03]
S2	P2	[d09:d10]

Рис. A.1. *Отношение с атрибутом интервального типа*

Поддержка интервальных атрибутов, а, следовательно, и темпоральных баз данных, подразумевает, среди прочего, поддержку обобщенных версий обычных алгебраических операторов. По причинам, которые здесь не так важны, мы называем эти обобщенные операторы «U_ операторами», то есть появляются операторы *U_ограничения*, *U_соединения*, *U_объединения* и т. д. Но – и это самое главное – все U_операторы при внимательном рассмотрении оказываются ни чем иным, как сокращенной записью некоторых комбинаций обычных алгебраических операторов. И здесь мы наблюдаем повышение уровня абстракции.

Еще два слова на эту тему. Во-первых, наш реляционный подход к темпоральным данным подразумевает не только «U_» версии алгебраических операторов, но также (а) «U_» ключи и «U_» внешние ключи, (б) «U_» операторы сравнения и (в) «U_» версии операторов INSERT, DELETE и UPDATE; и все эти конструкции тоже оказываются просто

сокращениями. Во-вторых, выясняется также, что описанная в *Манифесте* модель наследования типов играет в поддержке темпоральных данных важнейшую роль, что еще раз доказывает взаимосвязанность всех этих вопросов.

Высокоуровневые интерфейсы

Есть и другой способ надстраивать реляционную модель – создавая различные приложения, основанные на реляционном интерфейсе и предоставляющие специализированные службы. Одним из примеров может служить поддержка принятия решений, другим – добыча данных, еще одним – интерфейс на основе естественных языков. Для пользователей таких приложений реляционная модель скрыта, по крайней мере, в какой-то степени. (Но даже если это так и даже если большинство пользователей взаимодействуют с базой данных только посредством интерфейса, я полагаю, что проектирование самой базы данных должно основываться на твердых реляционных принципах.)

Кстати, допустим, что ваша работа – как раз реализация такого рода интерфейсных приложений. Что бы вы предпочли: реляционную СУБД или что-то другое (скажем, объектно-ориентированную СУБД)? И если вы предпочитаете первое – а я, разумеется, считаю, что так и должно быть, – то какую СУБД вы предпочли бы: ту, что поддерживает реляционную модель, или ту, что поддерживает SQL?

Если это еще неясно, изложу свою точку зрения. Мы далеко ушли от того времени, когда SQL рекламировался как язык, с которым пользователи могут работать самостоятельно, и я знаю немало людей, которые именно по этой причине отвергают мои многочисленные критические замечания в адрес SQL, считая их придирками. Конечные-то пользователи все равно с ним работать не будут, верно? Только программисты. Да и в любом случае большая часть исполняемого SQL-кода все равно пишется не человеком, а генерируется тем или иным приложением. Но мне кажется, что в качестве конечного языка SQL плох по тем же причинам, что и в качестве исходного. И потому я полагаю, что моя критика все-таки уместна.

Так что насчет SQL?

SQL не способен предоставить те прочные основания, которые нам необходимы для роста и развития. Такие основания должна заложить именно реляционная модель. Поэтому в *Третьем манифесте* мы с Дарвеном отвергаем SQL как таковой и предлагаем вместо него как можно скорее реализовать по-настоящему реляционный язык типа **Tutorial D**. Разумеется, мы не настолько наивны, чтобы думать, будто SQL когда-то исчезнет. Но мы надеемся, что **Tutorial D** или еще какой-то истинно реляционный язык окажется настолько превосходящим его в техническом отношении, что станет предпочтительным языком баз данных (путем естественного отбора), а языку SQL останется роль «языка на край-

ний случай». Фактически тут прослеживается параллель с миром языков программирования, из которого COBOL не исчез (и никогда не исчезнет), но приложения на нем пишут, когда совсем уж некуда деваться, поскольку существуют более удачные альтернативы. Нам SQL видится как COBOL для баз данных, и мы хотели бы, чтобы ему на смену пришел лучший язык.

Повторю, мы прекрасно осознаем, что базы данных и приложения на основе SQL будут сопровождать нас еще долго – думать иначе было бы утопией – и что мы должны уделить некоторое внимание тому, что делать с сегодняшним наследием SQL. Поэтому *Манифест* содержит несколько предложений по этому поводу. В частности, предлагается реализовать SQL поверх какого-нибудь истинно реляционного языка, так чтобы существующие SQL-приложения продолжали работать. Детальное обсуждение этих предложений здесь вряд ли уместно, достаточно сказать, что, на наш взгляд, возможно эмулировать такие нереляционные свойства SQL, как дубликаты и null-значения, не поддерживая их непосредственно в базовом реляционном языке.

В

Теория проектирования баз данных

Из принципа физической независимости от данных, обсуждавшегося в главе 1, напрямую следует, что логическое и физическое проектирование баз данных – разные дисциплины. Логическое проектирование связано с тем, как базу данных видит пользователь, а физическое – с тем, как логическая структура отображается на физическое хранение. Однако термин *теория проектирования баз данных* почти всегда применяется к логическому, а не к физическому проектированию (в предположении, что физическая структура обязательно зависит от деталей реализации конкретной СУБД, тогда как логическая структура является или должна являться независимой от СУБД). Поэтому в настоящем приложении я буду употреблять просто термин *проектирование* (или *структура*), понимая под ним именно логическое проектирование, если явно не оговорено противное.

Одно замечание я хотел бы сделать прямо сейчас. Напомню, что полное ограничение переменной-отношения R можно рассматривать как системную аппроксимацию предиката этой переменной-отношения; напомню также, что предикатом R называется предполагаемая интерпретация, или смысл, R. Отсюда следует, что ограничения и предикаты тесно связаны с задачей логического проектирования; в действительности вся суть логического проектирования и состоит в максимально тщательном определении предикатов и последующем отображении их на переменные-отношения и ограничения. Конечно, предикаты по необходимости не вполне формальны (в главе 8 отмечалось, что иногда их называют *бизнес-правилами*), тогда как переменные-отношения и ограничения обязательно носят формальный характер.

Кстати, описанное выше положение вещей объясняет, почему я не являюсь горячим сторонником моделей «сущность-связь» (E/R – entity/relationship) и других подобных графических методик. Проблема в том,

что диаграммы «сущность-связь» и аналогичные им картинки в состоянии представить не все ограничения, а только небольшую их часть, пусть и важную. Поэтому, хотя их использование и допустимо для того, чтобы объяснить проект на высоком уровне абстракции, думать, что на такой диаграмме структура представлена во всей полноте – заблуждение, иногда весьма опасное. *Напротив:* структура базы данных состоит из переменных-отношений, которые на диаграммах отражены, и ограничений, которые не отражены.

И еще одна мысль, которую я хотел бы высказать с самого начала. Напомню (см. главу 9), что представления должны выглядеть и вести себя в точности, как базовые переменные-отношения (я имею в виду не те представления, которые определены просто как сокращенная запись, а те, что призваны как-то изолировать пользователя от «реальной» базы данных). В общем случае пользователь взаимодействует с базой данных, которая содержит не только базовые переменные-отношения («реальной» базой данных), но также и представления (такую базу можно было бы назвать *пользовательской*). Но такая пользовательская база данных должна выглядеть и вести себя, как реальная, поэтому все обсуждаемые в настоящем приложении принципы проектирования применимы к пользовательским базам данных в той же мере, что и к реальным.

Я чувствую необходимость еще в одном вступительном замечании. Некоторые рецензенты рукописи этого приложения, похоже, подумали, что я собираюсь научить читателя элементам проектирования баз данных. Но такой цели я не ставил. Вы профессионально занимаетесь базами данных и с основами проектирования, надо полагать, уже знакомы. Поэтому я не собираюсь объяснять, как процесс проектирования происходит на практике; моя задача – закрепить некоторые известные вам аспекты проектирования, взглянув на них под, возможно, неожиданным для вас углом зрения, а также рассмотреть те аспекты, которые вам, быть может, не знакомы. Я не хочу тратить много времени на повторение всем известного. Например, я сознательно не буду углубляться в детали второй и третьей нормальной форм, поскольку они уже стали частью традиционных знаний по проектированию и не нуждаются в рассмотрении в такой книге, как эта (да и в любом случае они важны не столько сами по себе, сколько в качестве перехода к нормальной форме Бойса/Кодда, о которой я расскажу).

Место теории проектирования

Теория проектирования как таковая не является частью реляционной модели; это отдельная полноценная теория, построенная поверх этой модели. (Можно воспринимать ее как часть реляционной теории вообще, но, повторюсь, не как часть реляционной модели.) Однако она опирает-

ся на некоторые фундаментальные понятия – например, операторы проекции и соединения, – которые являются частью реляционной модели.

И еще одно: теория проектирования, о которой я буду рассказывать, не дает конкретных наставлений, как проектировать. Она лишь говорит, что плохого может случиться, если вы не станете проектировать базу данных очевидным образом. Рассмотрим пример базы данных о поставщиках и деталях. Очевидной является структура, которая предполагалась всюду в этой книге; я хочу сказать, что «очевидна» необходимость трех переменных-отношений, что атрибут STATUS принадлежит переменной-отношению S, атрибут COLOR – переменной-отношению P, атрибут QTY – переменной-отношению SP и так далее. Но почему все эти вещи так уж очевидны? Что ж, попробуем спроектировать базу данных по-другому, например, переместим атрибут STATUS из S в SP (интуитивно понятно, что это неподходящее место для него, поскольку статус – свойство поставщика, а не поставки). На рис. В.1 показаны значения атрибутов для модифицированной таким образом переменной-отношения «поставки» (я назвал ее STP во избежание недоразумений).

STP

SNO	STATUS	PNO	QTY
S1	20	P1	300
S1	20	P2	200
S1	20	P3	400
S1	20	P4	200
S1	20	P5	100
S1	20	P6	100
S2	10	P1	300
S2	10	P2	400
S3	30	P2	200
S4	20	P2	200
S4	20	P4	300
S4	20	P5	400

Рис. В.1. *Переменная-отношение STP - тестовые значения*

Одного взгляда на этот рисунок достаточно, чтобы понять, в чем недостаток такой структуры: она избыточна в том смысле, что каждый кортеж для поставщика S1 говорит нам, что S1 имеет статус 20, каждый кортеж для поставщика S2 – что S2 имеет статус 10 и так далее. А теория проектирования говорит, что проектирование базы данных способом, отличным от очевидного, всегда приводит к такой избыточности, и кроме того, сообщает, каковы будут последствия этой избыточности. Другими словами, вся теория проектирования вращается вокруг уменьшения избыточности, в чем мы вскоре и убедимся. Поэтому теорию проектирования иногда называют – пожалуй, слегка недоброжелательно – *хорошим источником плохих примеров*. Ее даже критиковали, на том основании, что все ее содержание – не более чем здравый смысл. К этой критике я еще вернусь в следующем разделе.

Чтобы придать обсуждению позитивную нотку, замечу, что теория проектирования может быть полезна для проверки того, что проекты, созданные путем применения какой-то другой методологии, не нарушают формальных принципов проектирования. Но... мы снова сталкиваемся с печальным фактом – хотя формальные принципы проектирования и составляют научную часть дисциплины проектирования, есть немало аспектов, которые в них вообще не рассматриваются. Проектирование базы данных остается по природе своей субъективным делом; формальные принципы, которые я собираюсь описать в настоящем приложении, – это то немногое, что можно отнести к науке в той отрасли, которую в основном следует относить, скорее, к искусству.

Итак, я хочу рассмотреть научную сторону проектирования. Точнее, я собираюсь исследовать две обширные темы: *нормализацию* и *ортогональность*. Полагаю, что хотя бы первая из них вам хорошо знакома. В частности, я считаю известными следующие факты:

- Существует несколько *нормальных форм* (первая, вторая, третья и т. д.).
- Проще говоря, если переменная-отношение R находится в ($n+1$)-ой нормальной форме, то она обязательно находится и в n-ой нормальной форме.
- Переменная-отношение может находиться в n-ой, но не находиться в ($n+1$)-ой нормальной форме.
- Чем выше нормальная форма, тем лучше с точки зрения проектирования.
- Все эти идеи опираются на некоторые *зависимости* (в данном контексте это просто другое название ограничений целостности).

Я хотел бы чуть подробнее остановиться на последних двух пунктах. Я сказал, что ограничения, вообще говоря, крайне важны для процесса проектирования. Однако оказывается, что конкретные ограничения, о которых идет речь здесь – так называемые зависимости, – обладают некоторыми формальными свойствами, отсутствующими (насколько нам известно) у других ограничений. Очень глубоко вдаваться в этот вопрос я не хочу, скажу лишь, что для таких зависимостей можно определить некоторые правила вывода, и именно существование правил вывода и позволяет разработать теорию проектирования, которую я собираюсь описать.

Еще раз повторю, что предполагаю наличие у вас определенных знаний в этой области. Но, как уже отмечалось, я хочу сосредоточить внимание на тех сторонах предмета, которые вам, возможно, незнакомы; я попытаюсь высветить более важные части и умалить остальные, а, более общо, я хочу взглянуть на предмет в целом под несколько непривычным для вас углом зрения.

Функциональные зависимости и нормальная форма Бойса/Кодда

Хорошо известно, что понятия *второй нормальной формы* (2НФ), *третьей нормальной формы* (3НФ) и *нормальной формы Бойса/Кодда* (НФБК) опираются на понятие *функциональной зависимости*.[1] Приведу точное определение этого понятия:

> **Определение:** Пусть A и B – подмножества заголовка переменной-отношения R. Тогда говорят, что переменная-отношение R удовлетворяет *функциональной зависимости* (ФЗ) $A \rightarrow B$, если для любого отношения, являющегося допустимым значением R, в любых двух кортежах, где одинаковы значения A, одинаковы также значения B.

ФЗ $A \rightarrow B$ произносится «B функционально зависит от A», или «A функционально определяет B», или просто «A стрелка B».

В качестве примера предположим, что существует ограничение целостности, состоящее в том, что если два поставщика находятся в одном городе, то они должны иметь одинаковый статус (см. рис. В.2, на котором я изменил статус поставщика S2 с 10 на 30, чтобы не нарушать это гипотетическое новое ограничение). Тогда ФЗ

```
{ CITY } → { STATUS }
```

удовлетворяется этой модифицированной версией – назовем ее RS – переменной-отношения S. Кстати, обратите внимание на фигурные скобки, я использую их, чтобы подчеркнуть тот факт, что обе части ФЗ – множества атрибутов, даже в том случае, когда (как в этом примере) они состоят всего из одного атрибута.

RS

SNO	SNAME	STATUS	CITY
S1	Smith	20	London
S2	Jones	30	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

← обратите внимание на изменения

Рис. В.2. *Модифицированная переменная-отношение RS – тестовые значения*

Как видно из примера, тот факт, что некоторая переменная-отношение R удовлетворяет ФЗ, составляет ограничение базы данных в смысле главы 8; точнее, ограничение (с одной переменной-отношением) этой переменной-отношения R. Например, ФЗ в рассмотренном выше примере эквивалентно следующему ограничению на языке **Tutorial D**:

---

[1] Термины *зависимость* и *функциональная зависимость* в литературе и в этой книге считаются синонимами.

```
CONSTRAINT RSC COUNT ( RS { CITY } ) =
               COUNT ( RS { CITY , STATUS } ) ;
```

Кстати, полезно запомнить: если переменная-отношение R удовлетворяет ФЗ $A \rightarrow B$, то она удовлетворяет и ФЗ $A' \rightarrow B'$ для любого надмножества $A' \supset A$ и любого подмножества $B' \subset B$. Иными словами, всегда можно добавить атрибуты в левую часть или удалить атрибуты из правой части, и то, что получится, по-прежнему будет ФЗ, которой удовлетворяет рассматриваемая переменная-отношение.

Сейчас я хочу напомнить вам один термин и ввести новый. Первый термин – *суперключ*. Напомню (см. главу 5), что суперключ – это, по существу, надмножество ключа (не обязательно собственное);[1] эквивалентно, подмножество SK заголовка переменной-отношения R является суперключом R тогда и только тогда, когда обладает свойством уникальности, но не обязательно свойством неприводимости. Таким образом, любой ключ является суперключом, но большинство суперключей не являются ключами. Например, {SNO,CITY} – суперключ, но не ключ переменной-отношения S. Замечу, в частности, что сам заголовок переменной-отношения R является суперключом R. Также напомню, что если SK – суперключ R и A – произвольное подмножество заголовка R, то R обязательно удовлетворяет ФЗ $SK \rightarrow A$, потому что если в двух кортежах R одинаковы значения SK, то, по определению, это один и тот же кортеж, а, значит, значения A тоже одинаковы.

Новый термин, который я хочу ввести, – это *тривиальная ФЗ*. ФЗ называется *тривиальной*, если ее невозможно нарушить. Например, все показанные ниже ФЗ тривиально удовлетворяются любой переменной-отношением, у которой есть атрибуты STATUS and CITY:

```
{ CITY , STATUS } → { CITY }
{ CITY , STATUS } → { STATUS }
{ CITY }          → { CITY }
```

Например, в первом случае, если в двух кортежах одинаковы значения CITY и STATUS, то понятно, что значения CITY одинаковы. На самом деле, ФЗ тривиальна тогда и только тогда, когда левая часть есть надмножество правой части (не обязательно собственное). Обычно мы не думаем о тривиальных ФЗ при проектировании базы данных просто потому, что они тривиальны. Но когда требуется формально и точно рассуждать о таких вещах, надо принимать во внимание все ФЗ – тривиальные они или нет.

Дав точное определение функциональной зависимости, я теперь могу сказать, что нормальная форма Бойса/Кодда (НФБК) нормальна относительно ФЗ, и точно определить, что это значит:

---

[1] Напомню также, что термин *ключ* я зарезервировал для потенциального ключа.

> **Определение:** Говорят, что переменная-отношение R находится в НФБК, если для любой нетривиальной ФЗ $A \rightarrow B$, которой удовлетворяет R, A является суперключом R.

Другими словами, для переменной-отношения, находящейся в НФБК, возможны только два вида ФЗ: тривиальные (от них, очевидно, избавиться невозможно) и «стрелки, исходящие из суперключей» (от них тоже нельзя избавиться). Или, как иногда говорят: *каждый факт представляет собой факт о ключе, всем ключе и ни о чем, кроме ключа*, – хотя я должен сразу же добавить, что эта неформальная характеристика, как бы она ни была приятна на слух, в действительности не точна, потому, в частности, что предполагает наличие только одного ключа.

Я чувствую необходимость развернуть мысль, изложенную в предыдущем абзаце. Боюсь, что слова об «избавлении» от некоторой ФЗ тоже несколько небрежны... Например, модифицированная переменная-отношение «поставщики» – RS на рис. В.2 – удовлетворяет ФЗ {SNO} → {STATUS}; но если разложить ее (что я и порекомендую сделать чуть ниже) на переменные-отношения SNC и CS (где SNC имеет атрибуты SNO, SNAME и CITY, а CS – атрибуты CITY и STATUS), то эта ФЗ в каком-то смысле «исчезает» и, следовательно, мы от нее «избавились». Но что мы имеем в виду, когда говорим, что ФЗ исчезла? Ответ таков: она превратилась в ограничение с несколькими переменными-отношениями (то есть в ограничение, в котором участвуют две или более переменные-отношения). Так что ограничение, конечно, никуда не делось – просто оно перестало быть ФЗ. Аналогичные замечания применимы и ко всем другим употреблениям слова «избавиться» в этом приложении.

Наконец, вы наверняка знаете, что теория нормализации гласит: если переменная-отношение R не находится в НФБК, то следует разложить ее на меньшие, которые уже будут находиться в такой форме (где под «меньшими» понимается «имеющие меньше атрибутов»). Например:

- Переменная-отношение STP (см. рис. В.1) удовлетворяет ФЗ {SNO} → {STATUS}, которая не является ни тривиальной, ни «стрелкой, исходящей из суперключа», – {SNO} не есть суперключ для STP, – следовательно, эта переменная-отношение не находится в НФБК (и, как вы видели выше, страдает избыточностью). Поэтому разложим ее на переменные-отношения, которые назовем SP и SS, где SP имеет атрибуты SNO, PNO и QTY (как обычно), а SS – атрибут SNO и STATUS. *Упражнение*: Выпишите значения переменных-отношений SP и SS, соответствующие значению STP, показанному на рис. В.1; убедитесь, что SP и SS находятся в НФБК, и, значит, декомпозиция устранила избыточность.
- Аналогично переменная-отношение RS (см. рис. В.2) удовлетворяет ФЗ {CITY} → {STATUS} и потому должна быть разложена на переменные-отношения SNC (с атрибутами SNO, SNAME и CITY) и CS

(с атрибутами CITY и STATUS). *Упражнение*: Выпишите значения переменных-отношений STC и CS, соответствующие значению RS, показанному на рис. B.2; убедитесь, что SNC и CS находятся в НФБК, и, значит, декомпозиция устранила избыточность.

Декомпозиция без потери информации

Если некоторая переменная-отношение не находится в НФБК, она может и должна быть разложена в меньшие переменные-отношения. Разумеется, важно, чтобы при такой декомпозиции не терялась информация – мы должны иметь возможность вернуться туда, откуда начали. Рассмотрим еще раз переменную-отношение RS (рис. B.2) с ФЗ {CITY} → {STATUS}. Предположим, что мы решили разложить ее не на переменные-отношения SNC и CS, а на переменные-отношения SNS и CS, как показано на рис. B.3. (CS в обоих случаях одинакова, а SNS имеет атрибуты SNO, SNAME и STATUS вместо SNO, SNAME и CITY.) Тогда, надеюсь, понятно, что (а) SNS и CS находятся в НФБК, но (б) декомпозиция приводит к потере информации – например, мы теперь не можем сказать, находится ли поставщик S2 в Париже или в Афинах.

SNS

SNO	SNAME	STATUS
S1	Smith	20
S2	Jones	30
S3	Blake	30
S4	Clark	20
S5	Adams	30

CS

CITY	STATUS
London	20
Paris	30
Athens	30

Рис. B.3. *Переменные-отношения SNS и CS – тестовые значения*

Почему одни декомпозиции приводят к потере информации, а другие не приводят? Во-первых, отметим, что процесс декомпозиции – это, по существу, *взятие проекции*; все «меньшие» переменные-отношения в приведенных до сих пор примерах были проекциями исходной переменной-отношения. Иными словами, оператор декомпозиции в точности совпадает с оператором проекции из реляционной алгебры (и именно в таком смысле я и буду использовать термин *декомпозиция*, если явно не оговорено противное). *Примечание*: И снова я небрежен. Как и все алгебраические операторы, проекция применяется к отношениям, а не к переменным-отношениям. Но мы часто говорим, что переменная-отношение CS является проекцией переменной-отношения RS, когда в действительности имеем в виду, что *отношение, которое является значением переменной-отношения CS в произвольный момент времени, есть проекция отношения, которое является значением переменной-отношения RS в тот же момент времени* (надеюсь, я выразился не слишком запутанно).

Идем далее. Когда мы говорим, что некоторая декомпозиция не приводит к потере информации, то в действительности имеем в виду следующее: *если соединить обе получившиеся проекции, то получится исход-*

ная переменная-отношение. Но, обратившись к рис. В.3, мы увидим, что переменная-отношение RS не равна результату соединения своих проекций SNS и CS, и именно поэтому декомпозиция приводит к потере информации. Напротив, если взглянуть на рис. В.2, то станет ясно, что RS равна результату соединения проекций SNC и CS, так что такая декомпозиция не приводит к потере информации.

Повторю еще раз, что оператор декомпозиции – это проекция, а оператор рекомпозиции – соединение. А формальный вопрос, составляющий самую суть теории нормализации, звучит так:

> Пусть R – переменная-отношение, и пусть $R1$, $R2$, ..., Rn – проекции R. Какие условия должны быть выполнены, чтобы R была равна соединению этих проекций?

Важный, хотя и неполный, ответ на этот вопрос дал Ян Хит (Ian Heath) в 1971 году, доказав следующую теорему:

> Пусть A, B, C – подмножества заголовка переменной-отношения R такие, что теоретико-множественное объединение A, B и C равно этому заголовку. Пусть AB обозначает теоретико-множественное объединение A и B и аналогично для AC. Если R удовлетворяет ФЗ $A \rightarrow B$, то R равно соединению своих проекций на AB и AC.

В качестве примера еще раз рассмотрим переменную-отношение RS (рис. В.2). Эта переменная-отношение удовлетворяет ФЗ {CITY} → {STATUS}. Следовательно, если взять в качестве A {CITY}, в качестве B {STATUS}, а в качестве C {SNO,SNAME}, то по теореме Хита RS может быть без потери информации разложена на проекции на {CITY,STATUS} и {CITY,SNO,SNAME}, что мы и так уже знаем.

Примечание

Если вам интересно, почему я назвал ответ Хита на исходный вопрос «неполным», то я поясню на рассмотренном выше примере. По существу, теорема говорит нам, что декомпозиция, показанная на рис. В.2, не приводит к потере информации, но она не говорит, что декомпозиция, показанная на рис. В.3, приводит к потере. То есть она дает достаточное условие, которое, однако, не является необходимым для того, чтобы декомпозиция не приводила к потере информации. (Более сильную форму теоремы Хита, которая дает необходимое и достаточное условие, доказал в 1977 году Рон Фейджин (Ron Fagin), но детали выходят за рамки этого обсуждения. См. упражнение В.18 в конце приложения.)

Попутно отмечу, что в той же статье, где Хит доказал эту теорему, он дал определение того, что назвал «третьей» нормальной формой, хотя фактически это было определение НФБК. Так как это определение было дано на три года раньше, чем его предложили Бойс и Кодд, то мне кажется, что НФБК по праву должна называться нормальной формой Хита. Но не называется.

И последнее замечание: из обсуждений в этом подразделе следует, что показанное раньше ограничение для переменной-отношения RS

```
CONSTRAINT RSC COUNT ( RS { CITY } ) =
               COUNT ( RS { CITY , STATUS } ) ;
```

можно было бы по-другому записать так:

```
CONSTRAINT RSC RS = JOIN { RS { SNO , SNAME , CITY } ,
                           RS { CITY , STATUS } } ;
```

(«В любой момент времени переменная-отношение RS равна соединению своих проекций на {SNO,SNAME,CITY} и {CITY,STATUS}»; здесь я воспользовался префиксной, или *n*-местной версией JOIN.)

Но разве все это не простой здравый смысл?

Выше я уже отмечал, что теорию нормализации критикуют на том основании, что все это простой здравый смысл. Рассмотрим, к примеру, снова переменную-отношение STP (рис. B.1). Очевидно, что она спроектирована плохо; избыточности как на ладони, последствия тоже ясны, и любой компетентный проектировщик «естественно» разложит ее на проекции SP и SS, которые мы обсуждали выше, пусть даже он никогда не слыхал о НФБК. Но что здесь означает слово «естественно»? Какими принципами руководствовался проектировщик, выбирая это «естественное» решение?

Ответ таков: именно принципами нормализации. То есть эти принципы уже отпечатались в мозгу компетентного проектировщика, даже если он никогда не изучал их формально и не может правильно назвать. Да, принципы продиктованы здравым смыслом – но *формализованным* здравым смыслом. (Здравый-то он здравый, но не всегда легко сказать, в чем смысл!) Теория нормализации как раз и формулирует, в чем заключаются некоторые стороны здравого смысла. На мой взгляд, реальное достижение теории нормализации состоит в следующем: она формализует некоторые принципы, продиктованные здравым смыслом, открывая путь к механизации этих принципов (то есть включения их в автоматизированные инструменты проектирования). Критики нормализации часто упускают эту мысль из виду; они заявляют, совершенно справедливо, что ее идеи – не более чем обычный здравый смысл, но, как правило, не осознают тот факт, что точное и формальное описание того, что означает здравый смысл, – само по себе существенное достижение.

1НФ, 2НФ, 3НФ

Нормальные формы низших по сравнению с НФБК порядков представляют, в основном, исторический интерес; я даже не буду приводить их определения. Напомню только, что все переменные-отношения нахо-

дятся, по меньшей мере, в 1НФ, даже те, у которых есть RVA-атрибуты.[1] Однако с точки зрения проектирования переменные-отношения с RVA-атрибутами обычно, хотя и не всегда, противопоказаны. Это не означает, что RVA-атрибутами никогда не следует пользоваться (на самом деле нет ничего плохого в том, что результаты запроса включают такие атрибуты); обычно не стоит лишь «забивать RVA-атрибуты в базу на этапе проектирования». Я бы не хотел детально заниматься этим вопросом здесь, скажу лишь, что переменные-отношения с RVA-атрибутами очень напоминают иерархические структуры, присутствующие в других, нереляционных системах типа IMS, и вместе с ними возникают все старые проблемы, присущие иерархиям. Вот краткий перечень таких проблем:

- Фундаментальная проблема связана с тем, что иерархии асимметричны. Поэтому, хотя на первый взгляд кажется, что они «упрощают» решение некоторых задач, другие задачи определенно только усложняются.
- Поэтому запросы тоже становятся асимметричны, а также усложняются по сравнению с симметричными аналогами.
- То же самое относится к ограничениям целостности.
- То же самое относится к обновлениям, только в еще более отягченной форме.
- Не существует рекомендаций по выбору «наилучшей» иерархии.
- Даже «естественные» иерархии, например организационные диаграммы, обычно лучше представлять неиерархически.

В примерах 7.10, 7.12 и 7.13 в главе 7 приведены примеры, иллюстрирующие некоторые из этих моментов. Повторю, однако, что иногда RVA-атрибуты могут оказаться полезными даже в базовых переменных-отношениях. См. упражнение В.14 в конце приложения.

Зависимости соединения и пятая нормальная форма

Пятая нормальная форма (5НФ) в некотором смысле, который я объясню ниже в этом разделе, – это «последняя нормальная форма». На самом деле, если НФБК – нормальная форма относительно функциональных зависимостей, то пятая нормальная форма нормальна относительно так называемых зависимостей соединения:

---

[1] Строго говоря, в 1НФ всегда находятся отношения, а не переменные-отношения (см. главы 1 и 2), то есть первая нормальная форма – это свойство отношений, а не переменных-отношений. Но никакого вреда не будет, если мы обобщим это понятие и на переменные-отношения тоже.

> **Определение:** Пусть *A*, *B*, ..., *C* – подмножества заголовка переменной-отношения *R*. Тогда *R* удовлетворяет зависимости соединения (ЗС)
>
> ```
> * { A , B , ... , C }
> ```
>
> (произносится «звездочка *A*, *B*, ..., *C*»), если любое отношение, являющееся допустимым значением *R*, равно соединению своих проекций на *A*, *B*, ..., *C*.

Из этого определения вытекают следующие выводы:

- Очевидно, что *R* можно без потери информации разложить на свои проекции на *A*, *B*, ..., *C* тогда и только тогда, когда оно удовлетворяет ЗС * {*A*, *B*, ..., *C*}.
- Также очевидно, что всякая ФЗ является также и ЗС, поскольку (по теореме Хита) если *R* удовлетворяет некоторой ФЗ, то она может быть без потери информации разложена на некоторые проекции (другими словами, удовлетворяет некоторой ЗС).

Для иллюстрации последнего замечания рассмотрим еще раз переменную-отношение RS (см. рис. В.2). Эта переменная-отношение удовлетворяет ФЗ {CITY} → {STATUS} и потому может быть без потерь разложена на проекции SNC (на SNO, SNAME и CITY) и CS (на CITY и STATUS). Отсюда следует, что переменная-отношение RS удовлетворяет ЗС *{SNC,CS} – если вы позволите мне использовать имена SNC и CS для обозначения не только самих проекций, но и соответствующих подмножеств заголовка этой переменной-отношения.

В предыдущем разделе мы видели, что всегда существуют «стрелки, исходящие из суперключей», то есть существуют некоторые функциональные зависимости, которые вытекают из наличия суперключей и от которых невозможно избавиться. Более общо, наличие суперключей подразумевает некоторые зависимости *соединения*, от которых также невозможно избавиться. Точнее, если переменная-отношение *R* удовлетворяет ЗС *{*A*,*B*,...,*C*}, то эта ЗС *обусловлена суперключами* тогда и только тогда, когда каждое подмножество *A, B, ..., C* является суперключом для *R*.[1] Например, рассмотрим нашу привычную переменную-отношение «поставщики» S. Она может быть без потерь разложена на свои проекции на SNO и SNAME, на SNO и STATUS и на SNO и CITY; иными словами, она удовлетворяет ЗС

```
* { SN , SS , SC }
```

---

[1] Это определение того, что «ЗС обусловлена суперключами», корректно лишь, если существует ровно один ключ. Его можно обобщить на случай двух и более ключей, но детали слишком запутаны, поэтому я их опущу. Точно и полное определение можно найти в работе «The Relational Database Dictionary, Extended Edition» (см. приложение D).

где SN – это {SNO,SNAME}, SS – {SNO,STATUS}, а SC – {SNO,CITY}. Поскольку каждая из этих проекций, очевидно, является суперключом для S, то ЗС действительно обусловлена суперключами. (Хотим ли мы в действительности выполнять такую декомпозицию – дело другое. Сможем, если захотим, и этого достаточно.)

В предыдущем разделе мы также видели, что некоторые ФЗ тривиальны. Вы уже, наверное, понимаете, что есть и тривиальные ЗС. Точнее, ЗС *{*A*,*B*,...,*C*} называется тривиальной, если хотя бы одно из подмножеств *A*, *B*, ..., *C* совпадает со всем заголовком соответствующей переменной-отношения *R*. Например, вот одна из многих тривиальных ЗС, которым удовлетворяет переменная-отношение S:

```
* { S , SN , SS , SC }
```

Примечание

Здесь я использую имя S для обозначения множества всех атрибутов – заголовка – переменной-отношения S (соответствует тождественной проекции S, то есть проекции этой переменной-отношения на все свои атрибуты). Кстати, я надеюсь, вы понимаете, что любую переменную-отношение можно без потерь разложить на заданное множество проекций, если одна из проекций в этом множестве является тождественной, хотя говорить в этом случае о «декомпозиции» было бы натяжкой, так как одна из проекций в такой «декомпозиции» совпадает с исходной переменной-отношением, то есть ни о каком разложении говорить, в общем-то, не приходится.

Дав точное определение понятию ЗС, я могу теперь определить, что такое 5НФ.

> **Определение:** Говорят, что переменная-отношение *R* находится в *5НФ*, если любая нетривиальная ЗС, которой удовлетворяет *R*, обусловлена суперключами *R*.

Иными словами, единственные ЗС, которым удовлетворяет переменная-отношение, находящаяся в 5НФ, – это те, от которых нельзя избавиться. Если переменная-отношение удовлетворяет еще каким-то ЗС, то она не находится в 5НФ (и потому страдает избыточностью) и, возможно, должна быть подвергнута декомпозиции.

Значимость 5НФ

Уверен, вы заметили, что при обсуждении ЗС и 5НФ я не привел примера переменной-отношения, которая находилась бы в НФБК, но не в 5НФ (и потому могла бы быть без потери информации на что-то с пользой разложена). А причина такова: хотя существуют ЗС, которые не являются простыми ФЗ, (а) такие ЗС редко встречаются на практике и (б) они оказываются довольно сложными, просто по определению. Из-за этой

сложности я решил не приводить пример сразу (но в следующем подразделе приведу), а, поскольку они редко встречаются, то с практической точки зрения они не слишком важны. На этом я хотел бы остановиться чуть подробнее.

Прежде всего, если вы проектируете базы данных, то, конечно, должны знать о ЗС и 5НФ, это часть вашего инструментария, и (при прочих равных условиях) следует стремиться к тому, чтобы все переменные-отношения в базе данных находились в 5НФ. Но большинство переменных-отношений (не все), встречающихся на практике, таковы, что если находятся хотя бы в НФБК, то находятся и в 5НФ. Редко приходится сталкиваться с переменной-отношением, которая бы находилась в НФБК, но не в 5НФ. Есть даже соответствующая теорема:

> Пусть R – переменная-отношение, находящаяся в НФБК, и пусть R не имеет составных ключей (то есть ключей, составленных из двух или более атрибутов). Тогда R находится в 5НФ.

Это очень полезная теорема. Она говорит, что если вам удалось получить НФБК (что достаточно просто) и если ваша переменная-отношение, находящаяся в НФБК, не имеет составных ключей (так бывает часто, хотя и не всегда), то о сложностях, связанных с ЗС и 5НФ, можно не беспокоиться, – заведомо известно, что эта переменная-отношение уже находится в 5НФ.

Попутно в интересах точности замечу, что эта теорема на самом деле относится к 3НФ, а не к НФБК, то есть она утверждает, что любая переменная-отношение, находящаяся в 3НФ и не имеющая составных ключей, обязательно находится в 5НФ. Но всякая переменная-отношение, находящаяся в НФБК, находится также и в 3НФ, а с прагматической точки зрения НФБК гораздо важнее 3НФ.

Есть и еще одна простая теорема на ту же тему:

> Пусть R – переменная-отношение, находящаяся в НФБК, и пусть R имеет хотя бы один неключевой атрибут (то есть атрибут, не являющийся частью ни одного ключа R). Тогда R находится в 5НФ.

Эта теорема, как и предыдущая, описывает ситуацию, которая на практике встречается очень часто.

Поэтому понятие 5НФ, по-видимому, не так уж важно с практической точки зрения. Однако с теоретической оно очень важно, потому что (как я сказал в начале раздела) это «последняя нормальная форма», и – что то же самое – эта форма нормальна относительно зависимостей соединения общего вида. Поскольку, если переменная-отношение R находится в 5НФ, то нетривиальные ЗС исчерпываются теми, что обусловлены суперключами. Поэтому единственными декомпозициями без потери информации являются те, в которых проекции производятся только на атрибуты некоторого суперключа; иными словами, всякая такая про-

екция включает какой-то ключ *R*. Как следствие,[1] соответствующие соединения «рекомпозиции» все взаимно однозначны, поэтому такая декомпозиция не устраняет и не может устранить никакую избыточность.

Сформулирую это по-другому. Сказать, что переменная-отношение *R* находится в 5НФ, – это все равно, что сказать, что дальнейшее разложение *R* на проекции без потери информации, хотя и возможно, но не устраняет никакую избыточность. *Но заметьте и не забывайте, что нахождение R в 5НФ еще не означает, что R свободна от избыточности*. Есть много видов избыточности, которые проецирование устранить не может, и это иллюстрирует мысль, высказанную мной ранее в разделе «Место теории проектирования», о том, что многие вопросы современная теория проектирования даже не затрагивает. Взгляните, к примеру, на рис. В.4, где показано тестовое значение переменной-отношения SPJ, которая находится в 5НФ и тем не менее страдает избыточностью. Так, тот факт, что поставщик S2 поставляет деталь P3, представлен несколько раз, как и факт, что деталь P3 поставляется для проекта J4 (JNO обозначает номер проекта). И то же самое можно сказать о факте, что проект J1 поставляется поставщиком S2. (Предикат звучит так *Поставщик SNO поставляет деталь PNO для проекта JNO в количестве QTY, и единственный ключ равен {SNO,PNO,JNO}*.) Единственная нетривиальная ЗС, которой удовлетворяет эта переменная-отношение, – *функциональная* зависимость[2]

```
{ SNO , PNO , JNO } → { QTY }
```

то есть «стрелка, исходящая из суперключа». Иначе говоря, QTY зависит от всех трех атрибутов SNO, PNO и JNO, и потому не может встречаться ни в какой переменной-отношении иначе, как со всеми тремя; таким образом, не существует декомпозиции без потерь, которая могла бы устранить такую избыточность.

Я хотел бы сделать еще несколько небольших замечаний. Во-первых, хотя раньше я об этом и не упоминал, вы, наверное, знаете, что 5НФ всегда достижима, то есть всегда можно разложить переменную-отношение, не находящуюся в 5НФ, на проекции, находящиеся в 5НФ (без потери информации).

---

[1] На самом деле то, что я здесь называю «следствием», может быть неверно, если *R* обладает более чем одним ключом. Рассуждение можно было бы обобщить и на этот случай, но детали опять же слишком запутаны, поэтому я их опускаю. Дополнительную информацию можно найти в работе «The Relational Database Dictionary, Extended Edition» (см. приложение D).

[2] Эквивалентно, если назвать SPJQ множество всех атрибутов SPJ, то эта переменная-отношение удовлетворяет ЗС *{SPJQ}, которая одновременно тривиальна и обусловлена суперключами *r*. (Напомню (см. главу 6), что соединение единственного отношения *r* – это само *r*.)

SPJ

SNO	PNO	JNO	QTY
S1	P1	J1	200
S1	P3	J4	700
S2	P3	J1	400
S2	P3	J2	200
S2	P3	J3	200
S2	P3	J4	500
S2	P3	J5	600
S2	P3	J6	400
S2	P3	J7	800
S2	P5	J1	100

Рис. В.4. *Переменная-отношение SPJ, находящаяся в 5НФ, – тестовое значение*

Во-вторых, каждая переменная-отношение, находящаяся в 5НФ, находится также в НФБК, поэтому то, что *R* находится в НФБК, безусловно, не исключает возможность, что *R* также находится и в 5НФ. Однако неформально фразу о том, что *R* находится в НФБК, часто интерпретируют так, будто *R не находится в нормальной форме более высокого порядка*. Я не придерживался такой практики в настоящем приложении (и дальше не буду ей следовать).

В-третьих, поскольку 5НФ – «последняя нормальная форма», ее часто называют нормальной формой *проекции-соединения* (ПС/НФ), чтобы подчеркнуть тот факт, что эта форма является нормальной до тех пор, пока мы ограничиваемся проекцией в качестве оператора декомпозиции и соединением в качестве оператора рекомпозиции. Но, конечно, можно рассматривать и другие операторы, а потому и другие нормальные формы. В частности, возможно и желательно определить (а) обобщенные версии операторов проекции и соединения и, значит, (б) обобщенные версии зависимости соединения и, значит, (в) новую «шестую» нормальную форму 6НФ. Эти идеи приобрели особое значение в связи с поддержкой темпоральных данных и подробно рассматриваются в книге «Temporal Data and the Relational Model», написанной Хью Дарвеном, Никосом Лоренцосом и мной (см. приложение D). Однако здесь я хочу лишь дать определение 6НФ, которое пригодно для «обычных» (то есть не темпоральных) переменных-отношений.

> **Определение:** Говорят, что переменная-отношение *R* находится в 6НФ, если она не удовлетворяет вообще никаким нетривиальным ЗС.

Другими словами, переменную-отношение, находящуюся в 6НФ, никак нельзя разложить без потери информации, разве что тривиально.[1] В частности, «обычная» переменная-отношение находится в 6НФ тог-

[1] Декомпозиция называется тривиальной, если она основана на зависимостях, которые сами по себе тривиальны.

да и только тогда, когда она находится в 5НФ, имеет степень *n* и не имеет ключей степени, меньшей чем *n – 1* (отметим, что всякая переменная-отношение, находящаяся в 6НФ, обязательно находится и в 5НФ). Наша привычная переменная-отношение «поставки» SP находится в 6НФ, как и переменная-отношение SPJ (см. рис. В.4). А переменные-отношения S (поставщики) и P (детали) находятся в 5НФ, но не в 6НФ. *Примечание*: Переменную-отношение в 6НФ иногда называют *неприводимой*, потому что ее нельзя без потерь разложить с помощью проецирования иначе, чем тривиально.

В заключение этого подраздела отметим, что ниоткуда не следует, что переменная-отношение хорошо спроектирована просто потому, что находится в 5НФ или 6НФ. Например, если переменная-отношение RS (рис. В.2) удовлетворяет ФЗ {CITY} → {STATUS}, то ее проекция на {SNO,STATUS}, конечно, находится в 6НФ, но вряд ли кто станет спорить, что такой выбор базовой переменной-отношения неудачен. См. обсуждение «сохранения зависимостей» в разделе «Тост за здоровье нормализации» ниже.

Еще о 5НФ

Рассмотрим рис. В.5, на котором показано значение упрощенного варианта переменной-отношения SPJ из предыдущего подраздела. Предположим, что этот упрощенный вариант удовлетворяет зависимости соединения *{SP,PJ,SJ}, где SP, PJ и SJ обозначают соответственно {SNO,PNO}, {PNO,JNO} и {SNO,JNO}. Что тогда означает эта ЗС с интуитивной точки зрения?

SPJ

SNO	PNO	JNO
S1	P1	J2
S1	P2	J1
S2	P1	J1
S1	P1	J1

***Рис. В.5.** Упрощенная переменная-отношение SPJ – тестовое значение*

- Прежде всего, ЗС по определению означает, что переменная-отношение равна соединению своих проекций SP, PJ и SJ и, стало быть, может быть без потерь разложена на эти проекции. (Я использую имена SP, PJ и SJ для обозначения самих проекций, а не соответствующих подмножеств заголовка переменной-отношения SPJ; надеюсь, что такая вольность с моей стороны вас не смутит.)
- Отсюда следует, что удовлетворяется такое ограничение

  ```
  IF (s,p) ∈ SP AND (p,j) ∈ PJ AND (s,j) ∈ SJ THEN (s,p,j) ∈ SPJ
  ```

  потому что если (*s*,*p*), (*p*,*j*) и (*s*,*j*) встречаются соответственно в SP, PJ и SJ, то (*s*,*p*,*j*), конечно, встречается в соединении SP, PJ и SJ, и это

соединение предположительно равно SPJ (именно в этом и состоит смысл ЗС). Так, для тех данных, что изображены на рис. В.5, кортежи (S1,P1), (P1,J1) и (S1,J1) встречаются соответственно в SP, PJ и SJ, а кортеж (S1,P1,J1) встречается в SPJ. *Примечание*: Надеюсь, что использованная мной сокращенная нотация кортежей понятна без объяснений, и напоминаю, что символ ∈ произносится как «элемент» или «встречается в».

- Далее, кортеж (*s,p*) встречается в SP тогда и только тогда, когда кортеж (*s,p,z*) встречается в SPJ для некоторого *z*. Аналогично, кортеж (*p,j*) встречается в PJ тогда и только тогда, когда кортеж (*x,p,j*) встречается в SPJ для некоторого *x*, а кортеж (*s,j*) встречается в SJ тогда и только тогда, когда кортеж (*s,y,j*) встречается в SPJ для некоторого *y*. Поэтому приведенное выше ограничение логически эквивалентно следующему:

```
IF для некоторых x, y, z (s,p,z) ∈ SPJ AND
                         (x,p,j) ∈ SPJ AND
                         (s,y,j) ∈ SPJ
THEN                     (s,p,j) ∈ SPJ
```

  Обращаясь к рис. В.5, мы видим, например, что каждый из кортежей (S1,P1,J2), (S2,P1,J1) и (S1,P2,J1) встречается в SPJ, поэтому там встречается и кортеж (S1,P1,J1).

Итак, исходная ЗС эквивалентна показанному выше ограничению. Но что означает это ограничение в терминах реального мира? Рассмотрим конкретный пример. Предположим, что переменная-отношение SPJ содержит кортежи, говорящие о том, что все три приведенных ниже высказывания истинны:

1. Смит поставляет разводные гаечные ключи для какого-то проекта.
2. Кто-то поставляет разводные гаечные ключи для Манхэттенского проекта.
3. Что-то поставляется Смитом для Манхэттенского проекта.

Тогда наличие ЗС говорит, что эта переменная-отношение должна содержать кортеж, подтверждающий истинность еще и следующего высказывания:

4. Смит поставляет разводные гаечные ключи для Манхэттенского проекта.

Но обычно из высказываний 1, 2 и 3, взятых вместе, высказывание 4 не вытекает. Если мы знаем только то, что высказывания 1, 2 и 3 истинны, то знаем, что Смит поставляет разводные ключи для какого-то проекта (пусть *z*), что какой-то поставщик (пусть *x*) поставляет разводные ключи для Манхэттенского проекта и что Смит поставляет какую-то деталь (пусть *y*) для Манхэттенского проекта, – но мы не можем сделать отсюда вывод, что *x* – это Смит, или что *y* – это разводной ключ, или что *z* –

это Манхэттенский проект. Подобные ложные выводы – пример того, что иногда называют *ловушками связи* (*connection trap*). Однако в данном случае наличие ЗС говорит, что *ловушки нет*, то есть мы можем без опаски выводить высказывание 4 из высказываний 1, 2 и 3.

Обратите внимание на циклическую природу ограничения («если *s* связано с *p*, а *p* связано с *j*, а *j* связано снова с *s*, то *s* и *p* и *j* должны быть связаны напрямую в том смысле, что должны встречаться в одном и том же кортеже»). Если имеется такое циклическое ограничение, то, возможно, мы столкнулись с переменной-отношением, которая находится в НФБК, но не в 5НФ.[1] Впрочем, в моей практике такие ограничения встречались крайне редко, поэтому я и сказал в предыдущем подразделе, что не думаю, что с практической точки зрения они так уж существенны.

Я хочу завершить этот раздел кратким замечанием о четвертой нормальной форме (4НФ). В подразделе «Значимость 5НФ» я сказал, что всякий проектировщик баз данных должен знать о ЗС и 5НФ. На самом деле, вы должны знать и о многозначных зависимостях (МЗЗ) и о четвертой нормальной форме. Однако эти понятия я упоминаю лишь для полноты картины; как и 2НФ и 3НФ, они представляют, в основном, исторический интерес. Для справки:

- МЗЗ – это такая ЗС, которая включает не более двух проекций (на практике обычно ровно две);
- переменная-отношение находится в 4НФ, если любая нетривиальная МЗЗ, которой она удовлетворяет, обусловлена некоторым суперключом.

Детали того, что означает тривиальность МЗЗ и обусловленность некоторым суперключом, выходят за рамки настоящего обсуждения (см. упражнение В.19 в конце приложения); отмечу лишь, что из этих определений следует, что повторяющегося разложения без потери информации ровно на две проекции достаточно для достижения 4НФ. Но вы, конечно, заметили, что ЗС в предыдущем подразделе требовала трех проекций. На самом деле, мы можем сказать, что для достижения 5НФ декомпозиция на n проекций (где $n > 2$) необходима только в том случае, когда рассматриваемая переменная-отношение удовлетворяет циклическому ограничению с n ветвями, или, эквивалентно, когда она удовлетворяет ЗС, включающей n проекций, и не удовлетворяет никакой ЗС с меньшим числом проекций.

---

[1] Если ограничение принимает более простую форму «если *s* связано с *p* и *j* связано с *s*, то *s* и *p* и *j* должны быть связаны напрямую», то, возможно, мы столкнулись с переменной-отношением, которая находится в НФБК, но не в 4НФ (и тем более не в 5НФ). См. упражнение В.16 в конце приложения.

Тост за здоровье нормализации

Нормализацию ни в коем случае нельзя считать панацеей, в чем легко убедиться, рассмотрев ее цели и то, насколько хорошо она им отвечает. Итак, вот цели:

- Прийти к структуре, которая бы «хорошо» представляла реальный мир, то есть была бы интуитивно понятна и являлась хорошей основой для будущего расширения
- Уменьшить избыточность
- Тем самым избежать некоторых аномалий при обновлении
- Упростить формулировку и проверку некоторых ограничений целостности

Рассмотрим их по порядку.

Хорошее представление реального мира. С этим нормализация справляется. У меня нет критических замечаний.

Уменьшение избыточности. Нормализация – неплохой первый шаг на пути решения этой проблемы, но только первый. С одной стороны, она сводится к взятию проекций, а, как мы видели, не всякую избыточность удается устранить с помощью проецирования; есть и такие виды избыточности, пока еще не обсуждавшиеся, которые нормализация даже не рассматривает. С другой стороны, цель уменьшения избыточности может вступать в конфликт с другой целью, также пока не обсуждавшейся, а именно с целью сохранения зависимостей. Объясню. Рассмотрим следующую переменную-отношение (атрибут ZIP обозначает почтовый индекс):

```
ADDR { STREET , CITY , STATE , ZIP }
```

Предположим, что эта переменная-отношение удовлетворяет таким ФЗ:

```
{ STREET , CITY , STATE } → { ZIP }
{ ZIP }                  → { CITY , STATE }
```

Первая ФЗ означает, что {STREET,CITY,STATE} – ключ, вторая – что эта переменная-отношение не находится в НФБК. Однако, если применить теорему Хита и разложить ее без потери информации на НФБК-проекции следующим образом (в качестве *A* выступает {ZIP}, в качестве *B* – {CITY,STATE}, а в качестве *C* – {STREET}):

```
ZCS { ZIP , CITY , STATE }
    KEY { ZIP }

ZS { ZIP , STREET }
   KEY { ZIP , STREET }
```

то ФЗ {STREET,CITY,STATE} → {ZIP}, которой исходная переменная-отношение, конечно, удовлетворяла, «исчезает». (Она удовлетворяется со-

единением ZCS и ZS, но ни одной из этих проекций в отдельности.) Следовательно, переменные-отношения ZCS и ZS нельзя обновлять независимо. Предположим, например, что эти проекции в настоящий момент имеют значения, показанные на рис. В.6; тогда любая попытка вставить кортеж <10111,Broadway> в ZS нарушит «отсутствующую» ФЗ. Однако этот факт нельзя обнаружить, не принимая во внимание проекцию ZCS, помимо ZS. Именно по таким причинам цель сохранения зависимостей гласит: *не расщепляйте зависимость на проекции*. Однако приведенный пример показывает, что, как ни печально, эта цель и цель разложения на НФБК-проекции иногда могут конфликтовать.

ZCS

ZIP	CITY	STATE
10003	New York	NY
10111	New York	NY

ZS

ZIP	STREET
10003	Broadway

Рис. В.6. *Проекции ZCS и ZS – тестовые значения*

Отступление

Однако в главе 9 показано, что если мы все-таки выполним такую декомпозицию, то, по крайней мере, сможем определить соединение проекций ZCS и ZS в виде представления, а затем определить {ZIP,CITY,STATE} в качестве ключа этого представления; иными словами, можно хотя бы без особых трудов сформулировать необходимое ограничение.

Избежать аномалий при обновлении. Этот пункт по существу совпадает с предыдущим («уменьшение избыточности»). Хорошо известно, что не до конца нормализованная структура может быть подвержена некоторым аномалиям при обновлении именно из-за остаточной избыточности. Например, в переменной-отношении STP (рис. В.1) может случиться так, что поставщик S1 в одном кортеже будет иметь статус 20, а в другом 25. Конечно, такая «аномалия обновления» может возникать только при недостаточно тщательной проверке ограничений целостности. Быть может, правильнее было бы охарактеризовать проблему аномалий при обновлении следующим образом: ограничения, необходимые для предотвращения подобных аномалий, проще формулировать и, быть может, проверять, если структура базы данных полностью нормализована (см. следующий абзац). Или так: в случае полностью нормализованной базы данных увеличивается количество допустимых обновлений одного кортежа (поскольку отсутствие нормализации означает избыточность, а при наличии избыточности иногда требуется обновлять сразу несколько сущностей).

Упростить формулировку и проверку некоторых ограничений целостности. Из общих соображений очевидно, что одни ограничения влекут за собой другие. Вот тривиальный пример: если количество должно

быть меньше или равно 5000, то очевидно, что оно должно быть меньше или равно 6000. Если ограничение *A* влечет за собой ограничение *B*, то, формулируя и проверяя *A*, мы «автоматически» формулируем и проверяем *B* (на самом деле, *B* вообще необязательно формулировать, разве что для документирования). А нормализация до 5НФ дает очень простой способ сформулировать и проверить некоторые важные ограничения; фактически нам нужно лишь определить ключи и постулировать их уникальность, – что мы все равно делаем, – и тогда автоматически будут сформулированы и проверены все ЗС (а также все МЗЗ и ФЗ), так как все они обусловлены этим ключами. Поэтому нормализация неплохо справляется и с этой задачей.

С другой стороны, вот несколько причин, помимо названных выше, по которым нормализация не панацея:

- Во-первых, ЗС – не единственный вид ограничений, а для других нормализация бесполезна.
- Во-вторых, для данного множества переменных-отношений часто возможно несколько вариантов декомпозиции без потерь на 5НФ-проекции, и не существует никаких формальных рекомендаций, какой выбрать.
- В-третьих, существует много вопросов, которые нормализация попросту игнорирует. Например, исходя из чего мы решаем, что должна быть только одна переменная-отношение «поставщики», а не по одной для поставщиков в Лондоне, в Париже и так далее? Очевидно, нормализация в классическом понимании тут не причем.

Но я вовсе не хочу, чтобы сделанные мною в этом разделе замечания воспринимались как какие-то нападки. Я твердо уверен, что любая не до конца нормализованная база данных противопоказана. И хочу завершить этот раздел доводом – логическим доводом, который вам, возможно, раньше не встречался, – в поддержку того, что *прибегать к денормализации следует лишь в самом крайнем случае*. Иначе говоря, отступать от полностью нормализованной структуры следует лишь тогда, когда все остальные попытки повысить производительность не увенчались успехом. (Кстати, замечу, что я согласен с расхожим мнением о том, что нормализация влияет на производительность. Для современных продуктов на основе SQL так оно и есть; однако это другая тема, к которой я еще вернусь в разделе «Некоторые замечания о физическом проектировании»). Итак, вот этот довод.

Все мы знаем, что денормализация негативно отражается на обновлении (логически негативно; некоторые обновления становится сложнее формулировать, а целостность базы данных может оказаться под угрозой). Но не так хорошо известен тот факт, что денормализация плоха и с точки зрения выборки, то есть становится сложнее формулировать некоторые запросы (или, что то же самое, становится проще сформулировать их неправильно, а это означает, что при выполнении таких

запросов вы получаете «правильные» ответы на неверно заданные вопросы). Приведу пример. Рассмотрим еще раз переменную-отношение RS (см. рис. В.2) с ФЗ {CITY} → {STATUS}. Ее можно рассматривать как результат денормализации переменных-отношений SNC (с атрибутами SNO, SNAME и CITY) и CS (с атрибутами CITY и STATUS). Теперь возьмем запрос «Получить средний статус города». Для данных на рис. В.2 значения статус для Афин, Лондона и Парижа равны соответственно 30, 20 и 30, поэтому с точностью до трех десятичных знаков после запятой средний статус равен 26,667.

Вот несколько попыток сформулировать этот запрос на SQL:

1.
```
SELECT AVG ( STATUS ) AS RESULT
FROM    RS
```

   Результат (неправильный): 26. Проблема в том, что статусы Лондона и Парижа учитываются дважды. Быть может, нужно включить DISTINCT в вызове AVG? Попробуем:

2.
```
SELECT AVG ( DISTINCT STATUS ) AS RESULT
FROM   RS
```

   Результат (неправильный): 25. Нет, нам нужно рассматривать разные *города*, а не разные значения статуса. Попробуем воспользоваться группировкой:

3.
```
SELECT CITY, AVG ( STATUS ) AS RESULT
FROM    RS
GROUP BY CITY
```

   Результат (неправильный): (Athens,30), (London,20), (Paris,30). При такой формулировке мы получаем средний статус *по городу*, а не общее среднее. Быть может, надо усреднить средние?

4.
```
SELECT CITY, AVG ( AVG ( STATUS ) ) AS RESULT
FROM   RS
GROUP BY CITY
```

   Результат: синтаксическая ошибка. Стандарт SQL совершенно правильно запрещает такие вложенные вызовы «функций множества».[1] Еще одна попытка:

---

[1] Я говорю «совершенно правильно» только потому, что речь идет об SQL; более ортодоксальный язык типа **Tutorial D**, безусловно, разрешил бы такие вложенные вызовы. Объясню. Рассмотрим SQL-выражение SELECT SUM(SP.QTY) AS SQ FROM SP WHERE SP.QTY > 100 (я сознательно взял другой пример). Аргументом оператора SUM здесь на самом деле является SP.QTY FROM SP WHERE SP.QTY > 100, и более ортодоксальный язык потребовал бы заключить все выражение в скобки. Но не SQL. Как следствие, выражение вида AVG(SUM(QTY)) обязано быть некорректным, потому что SQL не может понять, какие компоненты объемлющего выражения являются частью аргумента AVG, а какие – частью аргумента SUM.

5.
```
SELECT AVG ( TEMP.STATUS ) AS RESULT
FROM ( SELECT DISTINCT RS.CITY, RS.STATUS
       FROM   RS ) AS TEMP
```

Результат (наконец-то правильный): 26,667. Но обратите внимание, насколько сложнее это выражение по сравнению с его аналогом для полностью нормализованной структуры, представленной на рис. B.3:

```
SELECT AVG ( STATUS ) AS RESULT
FROM   CS
```

И это все, что я хотел сказать о нормализации (пока, во всяком случае). А теперь обратимся к теме, которая почти наверняка знакома вам куда меньше. Я имею в виду ортогональность, составляющую еще один скромный вклад науки в дело проектирования баз данных.

Ортогональность

На рис. B.7 показаны тестовые значения возможной, но откровенно неудачной структуры поставщиков: переменная-отношение SA представляет поставщиков в Париже, а переменная SB – поставщиков, которые либо не находятся в Париже, либо имеют статус 30. Как видите, такая схема приводит к избыточности – конкретно, кортеж для поставщика S3 встречается в обеих переменных-отношениях, – и, как обычно, наличие избыточности может приводить к аномалиям при обновлении. (Любая избыточность *всегда* является причиной таких аномалий.)

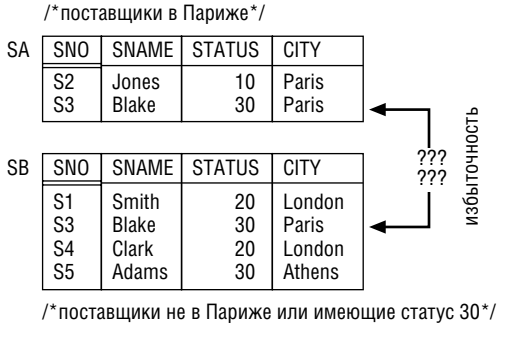


Рис. B.7. *Переменные-отношения SA и SB – тестовые значения*

Кстати, отметим, что поставщик S3 обязательно должен присутствовать в обеих переменных-отношениях. Допустим, что он встречается в SB, но не встречается в SA. Тогда из *допущения замкнутости мира*, примененного к SA, следовало бы, что поставщик S3 не находится в Париже. Но SB говорит, что S3 *находится* в Париже. Следовательно, мы имеем противоречие, и база данных противоречива (а из главы 8 мы знаем, к каким бедам это может привести).

Проблема, присущая структуре на рис. В.7, очевидна. Она как раз и заключается в том, что один и тот же кортеж должен присутствовать в двух разных переменных-отношениях, или, иначе говоря, в том, что семантика или предикаты этих переменных-отношений «перекрываются». Действительно, мы можем и, наверное, должны были бы написать соответствующее ограничение:

```
CONSTRAINT SA_AND_SB_OVERLAP
  ( SA WHERE STATUS = 30 ) =
  ( SB WHERE STATUS = 30 AND CITY = 'Paris' ) ;
```

Итак, очевидное правило гласит:

> *Принцип ортогонального проектирования (первая версия)*: не должно быть двух различных переменных-отношений таких, что если некоторый кортеж *t* присутствует в одной из них, то ограничения этих переменных-отношений требуют, чтобы тот же кортеж присутствовал и в другой.

Термин *ортогональный* призван подчеркнуть тот факт, что этот принцип по существу говорит, что переменные-отношения должны быть независимы друг от друга, а это не так, если их семантики перекрываются в указанном выше смысле.

Должно быть понятно, что две переменные-отношения не смогут нарушить этот принцип, если их типы различны, поэтому может сложиться впечатление, будто принцип ортогональности немногого стоит, – ведь очень редко бывает, что база данных содержит две или более переменные-отношения одного и того же типа. Однако взгляните на рис. В.8, где показана еще одна возможная, хотя и столь же неудачная, структура. В этом случае один и тот же кортеж никак не может встретиться в обеих переменных-отношениях, зато вполне может статься, что у кортежей из SX и из SY будут одинаковые проекции на {SNO,SNAME}[1], и это снова приводит к избыточности и аномалиям при обновлении. Поэтому нам следует обобщить сформулированный ранее принцип проектирования:

SX

SNO	SNAME	STATUS
S1	Smith	20
S2	Jones	10
S3	Blake	30
S4	Clark	20
S5	Adams	30

SY

SNO	SNAME	CITY
S1	Smith	London
S2	Jones	Paris
S3	Blake	Paris
S4	Clark	London
S5	Adams	Athens

Рис. В.8. *Переменные-отношения SX и SY – тестовые значения*

[1] Конечно, проекция – это реляционный оператор, но вполне имеет смысл определить вариант этого оператора, который будет применим к кортежам, а не к отношениям, и говорить о проекции кортежа. Аналогичное замечание справедливо и для других реляционных операторов.

> *Принцип ортогонального проектирования (вторая версия)*: пусть *A* и *B* – различные переменные-отношения. Тогда не должно существовать декомпозиций *A* и *B* без потери информации на *A1, A2, ..., Am* и *B1, B2, ..., Bn* соответственно таких, что для некоторой проекции *Ai* из множества *A1, A2, ..., Am* и некоторой проекции *Bj* из множества *B1, B2, ..., Bn* имеются ограничения переменных-отношений, которые говорят, что если некоторый кортеж *t* присутствует в *Ai*, то этот же кортеж *t* должен присутствовать и в *Bj*.

Первая версия принципа является частным случаем второй, потому что искомая «декомпозиция без потери информации», имеющаяся для любой переменной-отношения *R*, – это просто тождественная проекция *R*. Но и эта версия еще не является безупречной. Для справки я приведу формулировку, которую считаю безупречной, но не буду комментировать ее подробно.

> *Принцип ортогонального проектирования (окончательная версия)*: пусть *A* и *B* – различные переменные-отношения. Заменим *A* и *B* декомпозициями без потери информации на проекции *A1, A2, ..., Am* и *B1, B2, ..., Bn* соответственно такие, что любая *Ai* (*i = 1, ..., m*) и любая *Bj* (*j = 1, ..., n*) находятся в 6НФ. Пусть *i* и *j* таковы, что существует последовательность из нуля или более переименований атрибутов со следующим свойством: (а) применение ее к *Ai* дает *Ak* и (б) *Ak* и *Bj* имеют одинаковый тип. Тогда не должно существовать ограничения, утверждающего, что в любой момент времени (*Ak* WHERE *ax*) = (*Bj* WHERE *bx*), где *ax* и *bx* – условия, ни одно из которых не обращается в FALSE тождественно.

Из сказанного выше следует несколько выводов.

- Как и принципы нормализации, *принцип ортогонального проектирования*, по существу, продиктован здравым смыслом, но (опять же, как и в случае нормализации) – *формализованным* здравым смыслом.
- Цель ортогонального проектирования состоит в том, чтобы уменьшить избыточность и тем самым избежать аномалий при обновлении (как и в случае нормализации). На самом деле, ортогональность дополняет нормализацию в том смысле, что – говоря неформально – нормализация уменьшает избыточность *внутри одной* переменной-отношения, а ортогональность – *между разными* переменными-отношениями.
- В действительности ортогональность дополняет нормализацию и еще одним способом. Снова рассмотрим декомпозицию переменной-отношения S на ее проекции SX и SY (рис. В.8). Замечу, что эта декомпозиция удовлетворяет всем обычным принципам нормализации: обе проекции находятся в 5НФ, потери информации нет, зависимости сохранены и для реконструкции исходной переменной-отношения S

необходимы обе проекции.[1] Именно нарушение ортогональности, а не принципов нормализации – причина неудачности такой структуры.

- Предположим, мы по какой-то причине решили разложить переменную-отношение «поставщики» на множество ограничений (restriction). Тогда принцип ортогональности говорит, что эти ограничения должны попарно не пересекаться в том смысле, что никакой кортеж не должен присутствовать более, чем в одном ограничении. (Кроме того, объединение ограничений, которое фактически будет дизъюнктным объединением, должно давать исходную переменную-отношение.) Такую декомпозицию называют *ортогональной декомпозицией*.

Некоторые замечания о физическом проектировании

Реляционная модель ничего не говорит о физическом проектировании. Тем не менее есть ряд полезных вещей, которые можно сказать на эту тему в реляционном контексте. Как минимум, они вытекают из духа модели, хотя явно не сформулированы (и несмотря на то, что детали физической реализации по необходимости зависят от СУБД и в разных системах различны).

Первое: *физическое проектирование должно следовать за логическим*. То есть «правильный» подход состоит в том, чтобы сначала создать не имеющий изъянов логический проект, а уже потом отобразить его на те физические структуры, которые поддерживает целевая СУБД. По-другому эту мысль можно выразить, сказав, что физический проект должен вытекать из логического, а не наоборот. В идеале система должна самостоятельно вывести оптимальный физический проект без вмешательства человека. (Эта задача не так уж амбициозна, как может показаться. Чуть я ниже я скажу еще несколько слов на эту тему.)

Второе: в главе 1 я говорил, что одна из причин исключения любых физических аспектов из реляционной модели состояла в том, чтобы дать разработчикам возможность реализовать ее, как они сочтут нужным, – и тут-то недостаточное понимание модели широкими массами разработчиков нам здорово аукнулось. Без сомнения, большинство продуктов на основе SQL не сумели раскрыть весь заложенный в модели по-

[1] Ранее я говорил, что декомпозиция не приводит к потере информации, если соединение проекций дает исходную переменную-отношение. Это правда, но не вся правда. На практике мы хотим, чтобы выполнялось дополнительное требование: для соединения необходимы все проекции. Например, вряд ли мы сочли бы разложение переменной-отношения S на проекции на {SNO,STATUS}, {SNO,SNAME} и {SNO,STATUS,CITY} разумной структурой, несмотря на то, что S, безусловно, равно соединению этих трех проекций. А все потому, что первая проекция не принимает участия в процедуре реконструкции.

тенциал; в этих продуктах видимое пользователю и физически хранящееся – по существу, одно и то же. Другими словами, то, что физически хранится, – *прямое отображение* того, что пользователь логически видит (рис. В.9). (Я понимаю, что высказанные замечания чрезмерно упрощены, но для настоящего обсуждения этого достаточно.)

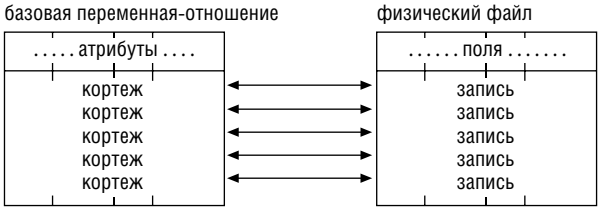


Рис. В.9. *Реализация с прямым отображением (достойна порицания)*

В реализации с прямым отображением многое не так, слишком многое, чтобы детально обсуждать недостатки здесь. Но самое главное – *почти полное отсутствие независимости от данных*: если нам понадобится изменить физическую структуру (как правило, из соображений производительности), то придется менять и логическую структуру тоже. В частности, речь идет о так часто раздающемся предложении «денормализовать ради повышения производительности». В принципе, логическая структура не имеет ничего общего с производительностью, но если она один к одному отображается на физическую... В общем, сами понимаете. Очевидно, можно найти более удачное решение. Защитники реляционной теории много лет убеждали, что реляционную модель необязательно реализовывать таким способом. И действительно, если все пойдет хорошо, то скоро появится совершенно новая технология, которая решит все проблемы, свойственные схеме с прямым отображением. Эта технология называется *The TransRelational™ Model*. Поскольку это технология реализации, она выходит за рамки настоящей книги, но ее предварительное описание можно найти в моей книге «An Introduction to Database Systems» (см. приложение D). Здесь я хочу лишь рассказать о нескольких желательных последствиях реализации, в которой физический и логический уровни строго разделены.

Во-первых, нам никогда не придется «денормализовывать ради повышения производительности» (на логическом уровне); все переменные-отношения могут находиться в 5НФ, а то и в 6НФ, и это не повредит производительности. Логическая структура вообще никак не будет отражаться на производительности.

Во-вторых, 6НФ создает основу для по-настоящему реляционного подхода к проблеме отсутствующей информации (без использования null-значений и трехзначной логики). Используя null-значения, вы по существу заставляете базу данных явно сообщать о том, что есть нечто, чего вы не знаете. Но если вы чего-то не знаете, то гораздо лучше об

этом вообще не говорить. Цитируя Виттгенштейна: «*Wovon man nicht reden kann, darüber muss man schweigen*» («О чем не можешь ничего сказать, о том должен молчать»). Например, предположим для простоты, что есть всего два поставщика S1 и S2, и статус S1 нам известен, а статус S2 – нет. В этой ситуации 6НФ могла бы выглядеть, как показано на рис. В.10.

SN

SNO	SNAME
S1	Smith
S2	Jones

SS

SNO	STATUS
S1	20

SC

SNO	CITY
S1	London
S2	Paris

Рис. В.10. *Статус поставщика S2 неизвестен*

Разумеется, о таком подходе к отсутствующей информации можно было бы сказать гораздо больше, но это приложение – неподходящее место для подобного разговора. (В статье «The Closed World Assumption», упомянутой в приложении D, имеется дополнительная информация.) Здесь я только хочу подчеркнуть, что при такой структуре отсутствует «кортеж», в котором статус поставщика S2 содержал бы null, – у нас *вообще* нет кортежа, показывающего статус S2.

И последнее: в той системе, которую я бегло обрисовал, будет возможность автоматически вывести оптимальную физическую структуру из логической без участия или почти без участия проектировщика-человека. Из-за нехватки места и ряда других причин я не могу привести свидетельства в пользу этой позиции, но твердо стою на ней.

Заключительные замечания

Основной упор в данном приложении был сделан на теории проектирования баз данных, под которой я понимаю, прежде всего, нормализацию и ортогональность (научные составляющие дисциплины проектирования). Идея в том, что логическая структура, в отличие от физической, является – или должна являться – независимой от конкретной СУБД, и, как мы видели, существуют солидные теоретические принципы, которые можно с пользой применить к этой проблеме.

Еще одна мысль, которую я не высказывал явно, заключается в том, что логическая структура, вообще говоря, должна в идеале не зависеть и от приложения, а не только от СУБД.[1] Задача заключается в том, чтобы составить проект, который описывает, что означают данные, а не как они будут использоваться, и в этой связи я подчеркивал важность ограниче-

[1] Одна из причин, по которым так желательна независимость от приложения, состоит в том, что мы не можем знать всех применений данных. Отсюда следует, что нам нужен проект, который не рассыпался бы на части при появлении новых требований к обработке.

ний и предикатов («бизнес-правил»). База данных должна быть верным представлением семантики ситуации, а семантику представляют именно ограничения. Таким образом, абстрактно процесс логического проектирования выглядит следующим образом:

1. Максимально тщательно выписать предикаты переменных-отношений.
2. Отобразить результаты шага 1 на переменные-отношения и ограничения. (В роли некоторых ограничений будут выступать ФЗ, МЗЗ и ЗС).

Значительная часть теории проектирования связана с *уменьшением избыточности*. Нормализация уменьшает избыточность внутри переменных-отношений, ортогональность – между несколькими переменными-отношениями. При обсуждении нормализации я уделил основное внимание НФБК и 5НФ, которые представляют собой *нормальные формы* относительно ФЗ и ЗС соответственно. (Однако вкратце я упомянул и другие нормальные формы, в том числе 6НФ.) Я подчеркнул, что нормализация позволяет проще формулировать (и, быть может, проверять) некоторые ограничения; эквивалентно можно сказать, что ее наличие допускает большее количество обновлений единственного кортежа, чем было бы возможно в противном случае. Я объяснил, что нормализация – это, по существу, формализованный здравый смысл. Я также привел логический и, возможно, не слишком широко известный довод в пользу отказа от денормализации, рассматривая проблему с точки зрения не столько обновления, сколько выборки. Я бы хотел также добавить, что хотя чаще всего к денормализации призывают во имя повышения производительности, на самом деле, как вы, наверное, знаете, это может привести к снижению производительности (и в плане обновления, и в плане выборки). Фактически «денормализация ради производительности» обычно означает повышение производительности одного приложения за счет остальных.

Я также описал *принцип ортогонального проектирования* (еще один пример формализации здравого смысла) и поделился некоторыми соображениями относительно физического проектирования. Во-первых, физическая структура должна вытекать из логической, а не наоборот. Во-вторых, нам остро необходимо уйти от повсеместно распространившегося способа реализации путем прямого отображения. В-третьих, было бы хорошо добиться полной автоматизации физического проектирования, и у меня есть некоторые надежды в этом отношении.

И последнее: я хочу подчеркнуть, что принципы нормализации и ортогональность в некотором смысле необязательны. Их нельзя назвать непререкаемыми правилами, которые следует заучить и никогда не нарушать. Как мы знаем, бывают основательные причины не доводить нормализацию «до конца» (я имею в виду основательные логические причины, а не «денормализацию ради производительности»). То же самое

справедливо в отношении ортогональности. Правда, надо понимать, что как неполная нормализация, так и нарушение ортогональности могут стать причиной избыточности и некоторых аномалий при обновлении. Даже при наличии теории проектирования, описанной в этом приложении, проектирование баз данных обычно подразумевает различные компромиссы.

Упражнения

Упражнение В.1. Дайте по возможности точные определения *функциональной зависимости* и *зависимости соединения*.

Упражнение В.2. Перечислите все ФЗ, тривиальные и нетривиальные, которым удовлетворяет переменная-отношение «поставки» SP.

Упражнение В.3. Понятие ФЗ опирается на понятие равенства кортежей. Истинно или ложно это утверждение?

Упражнение В.4. Декомпозиция без потери информации означает, что переменную-отношение можно разложить на проекции таким образом, что соединение этих проекций восстанавливает исходную переменную-отношение. Фактически, если проекции *r1* и *r2* отношения *r* таковы, что каждый атрибут *r* встречается хотя бы в одной из проекций *r1* и *r2*, то соединение *r1* и *r2* всегда порождает каждый кортеж *r*. Докажите это утверждение. (Из сказанного выше следует, что проблема декомпозиции с потерей информации связана с тем, что соединение порождает «лишние» кортежи. Так как в общем случае мы не знаем, какие кортежи соединения лишние, а какие настоящие, то имеет место потеря информации.)

Упражнение В.5. Докажите теорему Хита. Докажите, что обратное этой теореме утверждение неверно.

Упражнение В.6. Что такое суперключ? Что означает фраза «ФЗ обусловлена суперключом»? Что означает фраза «ЗС обусловлена суперключом»?

Упражнение В.7. Предполагается, что ключи уникальны и неприводимы. Нет сомнений, что система способна проверить уникальность. А как насчет неприводимости?

Упражнение В.8. Что такое (а) тривиальная ФЗ, (б) тривиальная ЗС? Является ли первая частным случаем второй?

Упражнение В.9. Пусть R – переменная-отношение степени n. Каково максимальное количество ФЗ, которым может удовлетворять R (тривиальных и нетривиальных)?

Упражнение В.10. Принимая во внимание, что A и B в ФЗ $A \rightarrow B$ – множества атрибутов, что произойдет, если хотя бы одно из этих множеств пусто?

Упражнение В.11. Рассмотрим предикат: в день *d* на протяжении периода *p* студент посещает занятие *l*, которое проводит преподаватель *t* в аудитории *c* (где *d* – рабочий день, от понедельника до пятницы, а *p* – протяженность периода, от 1 до 8 часов в течение этого дня). Каждое занятие занимает один период и имеет название, уникальное относительно всех занятий, проводимых в течение недели. Спроектируйте множество переменных-отношений для этой базы данных. Находятся ли ваши отношения в 5НФ? А в 6НФ? Каковы ключи?

Упражнение В.12. В большинстве примеров декомпозиции без потерь, приведенных в тексте этого приложения, показано разложение переменной-отношения ровно на две проекции. Бывает ли необходимо разлагать на три или более проекций?

Упражнение В.13. Многие проектировщики баз данных рекомендуют использовать искусственные или суррогатные ключи вместо так называемых естественных ключей. Например, мы могли бы добавить атрибут (назовем его SPNO) в переменную-отношение «поставки» (конечно, позаботившись о свойстве уникальности), а затем объявить {SPNO} суррогатным ключом этой переменной-отношения. (Отметим, однако, что {SNO,PNO} от этого не перестанет быть ключом, просто теперь этот ключ уже не единственный.) Таким образом, суррогатные ключи – это ключи в обычном реляционном смысле, но (а) они всегда состоят из единственного атрибута и (б) их значения служат просто суррогатами обозначаемых сущностей (то есть лишь представляют тот факт, что сущность действительно существует, никакой дополнительные семантики такие ключи не несут). В идеале суррогатные ключи должна генерировать система, но вопрос о том, кто их генерирует – система или пользователь, – к самой идее суррогатного ключа отношения не имеет. Два вопроса: суррогатные ключи и идентификаторы кортежей – это одно и то же? Как вы думаете, сама идея здравая?

Упражнение В.14. (*С благодарностью Хью Дарвену.*) Я решил организовать вечеринку, поэтому составил список тех, кого хотел бы пригласить, и предварительно поговорил с ними. Общая реакция была положительной, но некоторые сказали, что придут, только если придут некоторые другие приглашенные. Например, Боб и Кэл сказали, что придут, если придет Эми; Хэл сказал, что придет, если одновременно придут Дон и Ева либо придет Фэй; Гай сказал, что придет в любом случае; Джо сказал, что придет, если одновременно придут Боб и Эми и так далее. Спроектируйте базу данных, из которой было бы ясно, чье согласие от чьего зависит.

Упражнение В.15. Спроектируйте базу данных для следующей задачи. Сущностями являются служащие и программисты. Каждый программист – служащий, но не все служащие – программисты. У слу-

жащего есть номер, имя и зарплата. У программиста есть навык – владение (одним) языком программирования. Что изменилось бы, если бы у программиста могло быть произвольное количество навыков?

Упражнение В.16. Пусть A, B и C – подмножества заголовка переменной-отношения R такие, что теоретико-множественное объединение A, B и C равно всему заголовку. Обозначим AB теоретико-множественное объединение A и B и аналогично для AC. Тогда говорят, что R удовлетворяет многозначным зависимостям (МЗЗ)

$$A \twoheadrightarrow B$$
$$A \twoheadrightarrow C$$

(где $A \twoheadrightarrow B$ произносится «A двойная стрелка B», или «A многозначно определяет B», или «B многозначно зависит от A»), если R удовлетворяет ЗС *{AB,AC}. Покажите, что переменная-отношение R, удовлетворяющая МЗЗ $A \twoheadrightarrow B$ и $A \twoheadrightarrow C$, обладает следующим свойством: если она включает пару кортежей (*a,b1,c1*) и (*a,b2,c2*), то включает также пару кортежей (*a,b1,c2*) и (*a,b2,c1*).

Упражнение В.17. Покажите, что если R удовлетворяет ФЗ $A \rightarrow B$, то она удовлетворяет также МЗЗ $A \twoheadrightarrow B$.

Упражнение В.18. (*Теорема Фейджина*) Пусть R определена, как в упражнении В.16. Покажите, что R можно без потери информации разложить на проекции на AB и AC тогда и только тогда, когда она удовлетворяет МЗЗ $A \twoheadrightarrow B$ и $A \twoheadrightarrow C$.

Упражнение В.19. Покажите, что если K – ключ R, то $K \twoheadrightarrow A$ удовлетворяется для всех атрибутов A переменной-отношения R. *Примечание*: Здесь удобное место для того, чтобы ввести еще несколько определений. Напомним, что R находится в 4НФ, если любая нетривиальная МЗЗ, которой R удовлетворяет, обусловлена каким-то суперключом. МЗЗ $A \twoheadrightarrow B$ называется *тривиальной*, если AB равно заголовку R или A – надмножество B; эта МЗЗ называется *обусловленной суперключом*, если A – суперключ.

Упражнение В.20. Приведите пример переменной-отношения, которая находится в НФБК, но не находится в 4НФ.

Упражнение В.21. Спроектируйте базу данных для следующей задачи. Сущностями являются торговые представители, регионы продаж и изделия. Каждый представитель отвечает за один или несколько регионов; в каждом регионе есть один или несколько представителей. Каждый представитель отвечает за продажу одного или нескольких изделий; каждое изделие продает один или несколько представителей. Каждое изделие продается в каждом регионе, однако никакие два представителя не продают одно и то же изделие в одном и том же регионе. Каждый представитель продает один и тот же набор изделий в каждом регионе, за который отвечает.

Упражнение В.22. Напишите на языке **Tutorial D** предложение CONSTRAINT, выражающее ЗС, которой удовлетворяет переменная-отношение SPJ на рис. В.5.

Упражнение В.23. (Модифицированный вариант упражнения В.21.) Спроектируйте базу данных для следующей задачи. Сущностями являются торговые представители, регионы продаж и изделия. Каждый представитель отвечает за один или несколько регионов; в каждом регионе есть один или несколько представителей. Каждый представитель отвечает за продажу одного или нескольких изделий; каждое изделие продает один или несколько представителей. Каждое изделие продается в одном или нескольких регионах, и в каждом регионе продается одно или несколько изделий. Наконец, если представитель *r* отвечает за регион *a*, и изделие *p* продается в регионе *a*, и представитель *r* продает изделие *p*, то *r* продает *p* в *a*.

Упражнение В.24. Какие из следующих утверждений верны?

a. Любая переменная-отношение, ключом которой является множество всех атрибутов, находится в НФБК.

b. Любая переменная-отношение, ключом которой является множество всех атрибутов, находится в 5НФ.

c. Любая бинарная переменная-отношение находится в НФБК.

Упражнение В.25. В настоящее время в индустрии ведутся оживленные дискуссии по поводу возможности *баз данных XML*. Но XML-документы по природе своей иерархические. Как вы думаете, относятся ли приведенные в этом приложении критические замечания в адрес иерархий к базам данных XML? Обоснуйте свой ответ.

Упражнение В.26. Нарисуйте диаграммы «сущность-связь» для баз данных из упражнений В.11, В.14, В.21 и В.23. Какие выводы вы можете сделать из этого упражнения?

C

Ответы к упражнениям

Глава 1

Упражнение 1.1. Вот несколько примеров предложений из начала главы, в которых термин *отношение* следует заменить на *переменная-отношение*:

- «В каждом отношении есть по меньшей мере один потенциальный ключ»;
- «Внешним ключом называется множество атрибутов одного отношения, значения которых должны совпадать со значениями некоторого потенциального ключа в каком-то другом (или в том же самом) отношении»;
- «Оператор реляционного присваивания ... позволяет присвоить значение реляционного выражения ... какому-то отношению»;
- «Представлением (или виртуальным отношением) называется именованное отношение, значением которого в каждый момент времени t является результат вычисления некоторого реляционного выражения в этот момент».

И так далее.

Упражнение 1.2. Э. Ф. Кодд (1923–2003) первым придумал реляционную модель, среди многих других работ. В декабре 2003 года я опубликовал небольшую статью, в которой отдал дань уважения ему и его достижениям, ее текст можно найти на сайте ACM SIGMOD по адресу *http://www.acm.org/sigmod* и в других местах. Расширенное описание заслуг Кодда включено в мою книгу «Date on Database: Writings 2000-2006» (Apress, 2006).

Упражнение 1.3. Домен можно представлять себе как концептуальный пул значений, из которого выбираются значения фактических атрибутов фактических отношений. Другими словами, домен – это тип, а термины *домен* и *тип* можно считать синонимами, хотя лично я предпочитаю слово *тип*, как имеющее более длинную родословную (по крайней мере, в компьютерном мире). Однако термин *домен* употреблялся в большинстве старых работ по базам данных. *Примечание*: не путайте понятие домена в реляционной теории с таким же термином в SQL, где его в лучшем случае можно рассматривать как очень слабую разновидность типа. См. главу 2 (в частности, ответ к упражнению 2.1).

Упражнение 1.4. База данных удовлетворяет условию ссылочной целостности, если для каждого кортежа, содержащего *ссылку* (другими словами, значение внешнего ключа), существует *объект ссылки* (то есть кортеж в той переменной-отношении, на которую ведет ссылка, с таким же значением соответствующего потенциального ключа). Неформально говоря, если B ссылается на A, то A должно существовать. Более подробное обсуждение см. в главах 5 и 8.

Упражнение 1.5. Пусть R – переменная-отношение, и пусть r – отношение, являющееся значением R в некоторый момент времени. Тогда по определению заголовок, атрибуты и степень R совпадают с заголовком, атрибутами и степенью r. Аналогично тело, кортежи и кардинальность R по определению совпадают с телом, кортежами и кардинальностью r. Отметим, однако, что тело, кортежи и кардинальность R изменяются со временем, тогда как заголовок, атрибуты и степень не изменяются.

Кстати, из этих определений следует, что если мы используем SQL-предложение для добавления или удаления столбца из некоторой базовой таблицы T, то, по сути дела, заменяем эту таблицу T другой, логически отличной от нее таблицей T' (в таких контекстах термин *таблица* употребляется как SQL-аналог реляционного термина *переменная-отношение*). Иначе говоря, со строгой логической точки зрения T' – не «та же таблица, что и раньше». Однако в неформальных контекстах на эту тонкость удобно не обращать внимания.

Упражнение 1.6. См. раздел «Модель и реализация» в тексте главы.

Упражнение 1.7. Физическая независимость от данных – это независимость пользователей и приложений от способа физического хранения данных и доступа к ним. Это логическое следствие строгого разделения между моделью и ее реализацией. В той мере, в какой такое разделение соблюдается и, стало быть, достигается физическая независимость от данных, мы свободны вносить изменения в методы физического хранения и доступа к данным (возможно, из соображений производительности), не внося соответствующих изменений в запросы и приложения. Такая независимость желательна, по-

скольку позволяет уменьшить капиталовложения в обучение и разработку приложений.

Модель – это абстрактная система, с которой взаимодействуют пользователи, а реализация – конкретное воплощение этой абстрактной системы в конкретной физической вычислительной системе. Пользователи должны понимать модель, так как именно она определяет интерфейс, с которым они работают, но не обязаны понимать реализацию, потому что она скрыта (по крайней мере, так должно быть). Полезна следующая аналогия: чтобы водить автомобиль, вам не нужно понимать, что происходит под капотом, достаточно знать, как рулить, как переключать передачи и т. д. Следовательно, правила руления, переключения передач и т. д. – это модель, а происходящее под капотом – реализация. (Да, понимая, что творится под капотом, вы, возможно, стали бы водить лучше, но это знание необязательно. Аналогично может статься, что вы смогли бы лучше использовать модель, зная что-то о том, как она реализована, но в идеале это не должно быть необходимо.) *Примечание*: иногда употребляется термин *архитектура*, очень близкий по смыслу к тому, что выше названо *моделью*.

Упражнение 1.8. Строки в таблицах упорядочены сверху вниз, а кортежи в отношениях – нет; столбцы в таблицах упорядочены слева направо, а атрибуты отношений – нет; в таблицах могут быть строки-дубликаты, а в отношениях никогда не бывает кортежей-дубликатов. Кроме того, отношения содержат значения, а изображения таблиц – нет (они не содержат даже «вхождений» таких значений), вместо этого они содержат символы, обозначающие такие значения, – например, символ (или числительное) 5, который обозначает значение пять. Другие различия названы в ответе к упражнению 3.5 в главе 3.

Упражнение 1.9. *Без ответа*.

Упражнение 1.10. В этой книге я под термином *реляционная модель* понимаю абстрактную систему, предложенную самим Коддом (хотя с тех пор эта абстрактная система улучшалась, уточнялась и дополнялась). Я не называю этим словом реляционный проект какой-то конкретной базы данных. Реляционных моделей во втором смысле существует множество, а в первом – только одна. (Как отмечалось в тексте главы, эта тема значительно подробнее освещена в приложении А.)

Упражнение 1.11. Вот несколько примеров.

- Реляционная модель не имеет ничего общего с «хранимыми отношениями»; в частности, она абсолютно ничего не говорит о том, какие отношения хранятся, а какие – нет. На самом деле, она даже не говорит, что отношения вообще должны храниться, – может существовать и более удачный способ работы с ними (и такой способ есть, хотя подробности выходят за рамки этой книги).

- Даже если согласиться с тем, что термин «хранимое отношение» имеет какой-то смысл – и означает видимое пользователю отношение, которое представлено в памяти некоторым прямым и эффективным способом, не уточняя, что такое *прямой* и *эффективный*, – какие именно отношения «хранятся», не должно иметь никакого значения на реляционном (пользовательском) уровне системы. В частности, реляционная модель *не говорит*, что «таблицы» (или «базовые таблицы», или «базовые отношения») должны храниться, а представления – нет.
- В приведенной цитате не упомянуто важнейшее логическое различие между отношениями и переменными-отношениями.
- В приведенной цитате, похоже, предполагается, что *таблица* и *базовая таблица* – взаимозаменяемые термины и понятия (на мой взгляд, очень серьезная ошибка).
- В приведенной цитате, похоже, проводится различие между таблицами и отношениями (и/или переменными-отношениями). Если под «таблицей» понимается таблица SQL, то я, конечно, согласен с тем, что между ними имеются некоторые важные отличия, но, кажется, не они интересуют автора отрывка.
- «Важно различать хранимые отношения ... и виртуальные отношения». На самом деле с точки зрения пользователя (и реляционной модели, если на то пошло) крайне важно *вообще не* различать их!

Упражнение 1.12. Вот несколько ошибок:

- Сама реляционная модель вообще не «определяет таблицы» в том смысле, который подразумевается в цитированном отрывке. Она даже не «определяет» отношения (точнее, переменные-отношения). Такие определения предоставляются пользователем. И в любом случае, что такое «простая» таблица? А бывают еще и сложные?
- Что означает фраза «для каждого отношения и для связей многие-ко-многим»? Что означает «определить таблицы» для таких предметов?
- Насколько мне известно, следующие понятия не являются частью модели: сущности, связи между сущностями, связывание таблиц, «перекрестные ключи». (Да, действительно в оригинальной модели Кодда было правило, названное «целостность сущностей», но название так и осталось всего лишь названием, и в любом случае я это правило отвергаю.) Справедливо и то, что всем этим терминам можно дать некоторые интерпретации, но утверждения, проистекающие из таких интерпретаций, обычно оказываются неверными. Например, отношения не всегда представляют «сущности» (какая «сущность» представлена отношением, которое является проекцией поставщиков на атрибуты STATUS и CITY?).

- Первичные и вторичные индексы, равно как и быстрый доступ к данным, – понятия уровня реализации, они не имеют ничего общего с моделью. В частности, первичные или потенциальные ключи не следует приравнивать к «первичным индексам».
- «На основе заданных параметров»? Нельзя ли как-нибудь поточнее? Очень раздражает, и особенно в реляционном контексте (где точность мысли и выражения всегда было одной из основных целей), встретить такой ужасающе небрежный стиль. Э-э, ну-у, как бы это сформулировать, отношение – это что-то вроде таблицы, некий вид таблицы... в общем, вы понимаете, что я хочу сказать.
- Наконец, *где же операторы*? Весьма распространенная ошибка – полагать, что реляционная модель имеет дело только со структурой, напрочь забывая об операторах. Но без операторов никуда! Как заметил сам Кодд, «структура без операторов – все равно, что анатомия без физиологии».

Упражнение 1.13. Вот несколько возможных предложений CREATE TABLE. Относительно типов данных столбцов см. главу 2. *Примечание*: Эти предложения CREATE TABLE, вместе со своими эквивалентами на языке **Tutorial D**, повторены в главе 5, где обсуждаются более подробно.

```
CREATE TABLE S
  ( SNO    VARCHAR(5)   NOT NULL ,
    SNAME  VARCHAR(25)  NOT NULL ,
    STATUS INTEGER      NOT NULL ,
    CITY   VARCHAR(20)  NOT NULL ,
    UNIQUE ( SNO ) ) ;

CREATE TABLE P
  ( PNO    VARCHAR(6)   NOT NULL ,
    PNAME  VARCHAR(25)  NOT NULL ,
    COLOR  CHAR(10)     NOT NULL ,
    WEIGHT NUMERIC(5,1) NOT NULL ,
    CITY   VARCHAR(20)  NOT NULL ,
    UNIQUE ( PNO ) ) ;

CREATE TABLE SP
  ( SNO    VARCHAR(5)   NOT NULL ,
    PNO    VARCHAR(6)   NOT NULL ,
    QTY    INTEGER      NOT NULL ,
    UNIQUE ( SNO , PNO ) ,
    FOREIGN KEY ( SNO ) REFERENCES S ( SNO ) ,
    FOREIGN KEY ( PNO ) REFERENCES P ( PNO ) ) ;
```

Обратите внимание, что в SQL определения столбцов, а также спецификации ключей и внешних ключей находятся внутри одних и тех же скобок (сравните с тем, как это делается в **Tutorial D**, см. главу 2). Заметьте также, что по умолчанию столбцы SQL допускают

null-значения; если мы хотим их запретить (лично я хочу), то необходимо задавать явное ограничение. Задать такое ограничение можно разными способами, включение NOT NULL в состав определения столбца, пожалуй, самый простой.

Упражнение 1.14. На языке **Tutorial D** (на SQL показать не могу, потому что SQL не поддерживает реляционное присваивание):

```
SP := SP UNION
        RELATION { TUPLE { SNO 'S5' , PNO 'P6' , QTY 250 } } ;
```

Текст между ключевым словом UNION и завершающей точкой с запятой – это вызов селектора отношения (см. главу 3), он обозначает отношение, содержащее только вставляемый кортеж. *Примечание*: На практике лучше бы вместо UNION использовать D_UNION. См. главу 5.

Упражнение 1.15. Для полноты я дам ответ, но подробные объяснения отложу до главы 7:

```
S := WITH ( S WHERE CITY = 'Paris' ) AS R1 ,
          ( EXTEND R1 ADD ( 25 AS NEW_STATUS ) ) AS R2 ,
          R2 { ALL BUT STATUS } AS R3 ,
          R3 RENAME ( NEW_STATUS AS STATUS ) AS R4 :
     ( S MINUS R1 ) UNION R4 ;
```

Упражнение 1.16. Сначала рассмотрим присваивание общего вида:

```
R := rx ;
```

Здесь R – имя переменной-отношения, а rx – реляционное выражение, обозначающее отношение, которое следует присвоить R. Аналог на SQL мог бы выглядеть так:

```
DELETE FROM T ;
INSERT INTO T tx ;
```

Здесь T – SQL-таблица, соответствующая переменной-отношению R, а tx – SQL-таблица, соответствующая реляционному выражению rx. Отметим необходимость предварительного DELETE; также отметим, что (допуская вольность речи) между DELETE и последующим INSERT может произойти все что угодно, тогда как в реляционной формулировке *не существует* никакого «между DELETE и INSERT» (присваивание – семантически атомарная операция).

Итак, я ответил на вопрос: «Все ли реляционные присваивания можно выразить в терминах предложений INSERT / DELETE / UPDATE?» Однако обращаю внимание, что реляционная присваивание – это одна операция, тогда как приведенная выше комбинация DELETE/INSERT – последовательность двух разных операций. Отсюда, в частности, следует, что между ними может произойти ошибка, которая не случилась бы, если бы выполнялось настоящее присваивание.

Я бы хотел прояснить еще один вопрос. В тексте главы я сказал, что SQL не поддерживает реляционное присваивание напрямую, и это правда. Однако один из рецензентов возразил, что, например, следующее SQL-выражение «можно было бы рассматривать как реляционное присваивание» (я несколько упростил приведенный рецензентом пример):

```
SELECT LS.*
FROM ( SELECT SNO, SNAME, STATUS
       FROM   S
       WHERE  CITY = 'London' ) AS LS
```

По существу, рецензент считает, что это выражение есть присваивание некоторого табличного значения табличной переменной с именем LS. Но это не так. В частности, невозможно сделать следующий шаг – предъявить запрос к LS или как-то обновить ее. LS не является независимой полноправной таблицей, это лишь временная таблица, которая концептуально материализована в ходе процедуры вычисления заданного выражения. Такое выражение не является реляционным присваиванием. (Да и в любом случае присваивание любого вида – это предложение, а не выражение.)

И еще один момент. Стандарт SQL поддерживает вариант предложения CREATE TABLE – «CREATE TABLE AS», – который позволяет создать базовую таблицу, инициализировав ее результатом выполнения некоторого запроса, то есть не только создать, но и присвоить начальное значение. Однако после инициализации такая таблица ведет себя, как любая другая базовая таблица, то есть и CREATE TABLE AS тоже нельзя считать поддержкой реляционного присваивания.

Упражнение 1.17. Следующие далее рассуждения основаны на более развернутых, которые можно найти в моей книге «An Introduction to Database Systems» (см. приложение D).

Кортежи-дубликаты. По существу, это понятие не имеет смысла. Предположим для простоты, что отношение «поставщики» имеет всего два атрибута: SNO и CITY, и предположим, что оно содержит кортеж, говорящий о том, что «поставщик S1 находится в Лондоне» – истинный факт. Тогда если бы оно содержало дубликат такого кортежа (если бы такое было возможно), мы просто получили бы повторное подтверждение истинности этого факта. Но (как замечено в главе 4), если некое утверждение является истинным, то оно не станет более истинным, если повторить его дважды! Более подробное обсуждение см. в главе 4 или в статье «Double Trouble, Double Trouble», упомянутой в приложении D.

Упорядочение кортежей. Отсутствие упорядочения кортежей означает, что не существует такой вещи, как «первый кортеж», «пятый кортеж» или «97-й кортеж» отношения, как не существует и поня-

тия «следующий кортеж». Иными словами, не существует понятия позиционной адресации или «очередности». Если бы такие понятия существовали, то понадобились бы и дополнительные операторы, например: «получить *n*-й кортеж», «вставить кортеж *в это место*», «переместить кортеж *отсюда туда*» и так далее. На самом деле (цитируя приложение А), можно считать аксиомой, что если у нас есть *n* разных способов представления информации, то необходимо *n* разных наборов операторов. Если $n > 1$, то возникает больше операторов, которые нужно реализовывать, документировать, преподавать, изучать, запоминать и использовать. Однако дополнительные операторы лишь увеличивают сложность, но не добавляют выразительных возможностей! Нет ничего полезного, что можно было бы сделать лишь при $n > 1$ и нельзя при $n = 1$.

Кстати, еще один аргумент против упорядочения (любого) состоит в том, что позиционная адресация неустойчива – по мере вставок и удалений адреса изменяются. А кроме того, семантика конкретного элемента данных (например, кортежа) может зависеть от его позиции.

Упорядочение атрибутов. Отсутствие упорядочения атрибутов означает, что не существует такой вещи, как «первый атрибут» или «второй атрибут» (и так далее), как не существует и понятия «следующий атрибут» (нет понятия «очередности»), – на атрибуты можно ссылаться только по имени, но не по позиции. В результате сокращается простор для ошибок и запутанного программирования. Например, невозможно подорвать систему, каким-то образом «перепрыгнув» от одного атрибута к другому. Эта ситуация резко контрастирует с тем, что мы видим во многих языках программирования, где часто есть возможность, намеренно или случайно, воспользоваться физическим соседством логически различных элементов для совершения разного рода подрывных действий. *Примечание*: Многие другие негативные последствия упорядочения атрибутов (или упорядочения столбцов в контексте SQL) обсуждаются в последующих главах. См. также статью «A Sweet Disorder», упомянутую в приложении D.

Точности ради я должен добавить, что по причинам, которые нас здесь не интересуют, в математике (в противоположность реляционной модели) атрибуты отношений считаются упорядоченными слева направо. Аналогичное замечание касается и кортежей.

Глава 2

Упражнение 2.1. Тип – это конечное именованное множество значений – всех значений определенного вида, например: всех возможных целых чисел, всех возможных строк символов, всех возможных номеров поставщиков, всех возможных XML-документов, всех возможных отношений с некоторым заголовком и т. д. и т. п. Между типом

и доменом нет никакой разницы. *Примечание*: однако в SQL проводится различие между доменами и типами. В частности, поддерживаются предложения CREATE TYPE и CREATE DOMAIN. В первом приближении CREATE TYPE можно считать SQL-аналогом предложения TYPE языка **Tutorial D**, которое я буду обсуждать в главе 8 (хотя между тем и другим существует очень много различий, и не все из них по природе своей тривиальны). Предложение CREATE DOMAIN можно с большим снисхождением считать попыткой предоставить маленькую часть полной функциональности CREATE TYPE (оно появилось в языке в 1992 году, а CREATE TYPE только в 1999); теперь, когда существует CREATE TYPE, не осталось причин использовать и даже поддерживать CREATE DOMAIN.

Упражнение 2.2. С каждым типом ассоциирован по меньшей мере один селектор; селектором называется оператор, который позволяет выбрать или задать произвольное значение рассматриваемого типа. Пусть T – некоторый тип, а S – селектор для T; тогда каждое значение типа T должно возвращаться некоторым вызовом S, и каждый вызов S должен возвращать некоторое значение типа T. Дополнительные сведения см. в главе 8. *Примечание*: В языке **Tutorial D** селекторы предоставляются «автоматически» (так как реляционная модель, по крайней мере неявно, требует их наличия), но в SQL (в общем случае) это не так. На самом деле, хотя понятие селектора по необходимости существует, в SQL нет для него термина, и уж точно само слово *селектор* термином SQL не является. Подробности выходят за рамки этой книги.

Литерал – это «самоопределенный символ»; он обозначает значение, которое можно определить на этапе компиляции. Точнее, литерал – это символ, обозначающий значение, которое фиксировано и определено этим символом (а тип значения поэтому тоже фиксирован и определен типом символа). Вот примеры литералов в языке **Tutorial D**:

```
4                          /* литерал типа INTEGER  */
'XYZ'                      /* литерал типа CHAR     */
FALSE                      /* литерал типа BOOLEAN  */
5.0                        /* литерал типа FIXED    */
POINT(5.0,2.5)             /* литерал типа POINT    */
```

Для любого значения любого типа, включая типы кортежа и отношения, должен быть какой-то способ обозначения с помощью литерала. Литерал – это частный случай вызова селектора; точнее, это вызов селектора, все аргументы которого сами заданы литералами (откуда, в частности, следует, что вызов селектора без аргументов, например вызов селектора типа INTEGER, 95, является литералом по определению). Отметим, что существует логическое различие между литералом как таковым и константой; константа – это значение, а литерал – символ, обозначающий значение.

Упражнение 2.3. Оператор THE_ предоставляет доступ к некоторому компоненту некоторого «допустимого представления» какого-то значения заданного типа. Подробнее см. главу 8. *Примечание*: В первом приближении можно считать, что операторы THE_ предоставляются в языках **Tutorial D** и SQL «автоматически». Однако, хотя понятие оператора THE_ по необходимости существует, в SQL нет для него термина, и уж точно само слова *оператор THE_* термином SQL не является. Подробности выходят за рамки этой книги.

Упражнение 2.4. В принципе, верно; но на практике не всегда (впрочем, неверно оно может быть только из-за путаницы между моделью и реализацией).

Упражнение 2.5. *Параметр* – это формальный операнд, участвующий в определении некоторого оператора. *Аргумент* – это фактический операнд, подставляемый вместо параметра при вызове этого оператора. (Часто эти термины употребляются как синонимы, из-за чего возникает много недоразумений, так что будьте начеку.) *Примечание*: существует логическое различие между аргументом как таковым и выражением, которое применяется для его задания. Например, рассмотрим выражение (2+3)–1, являющееся вызовом арифметического оператора «–». Первый аргумент этого вызова – значение, равное пяти, но задан он выражением 2+3 (которое является вызовом арифметического оператора «+»; на самом деле, *любое* выражение представляет собой вызов какого-то оператора).

База данных – это репозиторий данных. (*Примечание*: Можно дать гораздо более точные определения, одно из них приведено в главе 5 этой книги.) *СУБД* – это программная система для управления базами данных; она обеспечивает восстановление, конкурентный доступ, запросы и обновление, а также другие службы.

Внешний ключ – это подмножество заголовка некоторой переменной-отношения, значения которого должны быть равны значениям некоторого ключа какой-то другой (или той же самой) переменной-отношения. *Указатель* – это значение (по существу, адрес), для которого должны быть определены операторы весьма специального вида и, прежде всего, операторы ссылки и разыменования.[1] *Примечание*: краткие определения операторов ссылки и разыменования приведены в сноске в тексте главы.

Сгенерированный тип – это тип, полученный в результате выполнения некоторого генератора типа, например ARRAY или RELATION; таким образом, конкретные типы массива и отношения являются сгенерированными типами. Несгенерированный тип – это тип, не являющийся сгенерированным.

[1] Гораздо более полное обсуждение логического различия между внешними ключами и указателями можно найти в статье «Inclusion Dependencies and Foreign Keys» (см. приложение D).

Отношение – это значение; у него есть тип, но само оно типом не является. *Тип* – это именованное множество значений (всех возможных значений определенного вида).

Скалярный тип – это тип, у которого нет видимых пользователю компонентов; *нескалярный тип* – это тип, не являющийся скалярным. Значения, переменные, операторы и т. д. называются скалярными или нескалярными в зависимости от того, принадлежат ли они скалярному или нескалярному типу. Однако имейте в виду, что эти термины при тщательном анализе не являются ни формальными, ни точными. В частности, в главе 3 мы столкнемся с двумя важными отношениями TABLE_DUM и TABLE_DEE, которые согласно данному определению являются скалярными (по крайней мере, такая точка зрения возможна)!

Тип – это понятие, относящееся к модели; у типов есть семантика, которая должна быть понятна пользователю. *Представление типа* – понятие, относящееся к реализации; предполагается, что представления скрыты от пользователя. В частности (и это отмечалось в тексте главы), если *X* – значение или переменная типа *T*, то операторы, применимые к *X*, – это операторы, определенные для *T*, а не операторы, определенные для представления *T*. Так, из того, что представление типа ENO («номера служащих») – CHAR (к примеру), не следует, что мы можем конкатенировать два номера служащих; это возможно только, если оператор конкатенации («‖») определен для типа ENO.

Системный (или *встроенный*) тип – это тип, который становится доступен сразу после установки системы («лежит в одной коробке с системой»). *Определенный пользователем*, или *пользовательский*, тип – это тип, определение и реализация которого предоставляются квалифицированным пользователем уже после установки системы. (Однако для пользователя – в отличие от проектировщика и разработчика – такой тип должен выглядеть и вести себя в точности так же, как системный.)

Системный (или *встроенный*) оператор – это оператор, который становится доступен сразу после установки системы («лежит в одной коробке с системой»). *Определенный пользователем*, или *пользовательский*, оператор – это оператор, определение и реализация которого предоставляются квалифицированным пользователем уже после установки системы. (Однако для пользователя – в отличие от проектировщика и разработчика – такой оператор должен выглядеть и вести себя в точности так же, как системный.) Определенные пользователем операторы могут принимать значения как системных, так и пользовательских типов (в любом сочетании), но системные операторы могут принимать значения только системных типов.

Упражнение 2.6. Приведением типа называется неявное преобразование типа. Оно не поощряется, так как чревато ошибками (однако от-

метим, что этот вопрос носит в основном прагматический характер; разрешены приведения или нет, не имеет отношения к реляционной модели как таковой).

Упражнение 2.7. Потому что при этом происходит смешение понятий типа и его представления.

Упражнение 2.8. Генератором типа называется оператор, который возвращает не значение, а тип (и вызывается во время компиляции, а не выполнения). В реляционной модели поддерживается два таких оператора: TUPLE и RELATION. *Примечание*: Эти конкретные генераторы генерируют нескалярные типы, но нет никаких причин, по которым только нескалярные типы и можно генерировать. Так, оператор REF в SQL являет собой пример генератора скалярного типа.

Упражнение 2.9. Говорят, что отношение находится в первой нормальной форме (1НФ), если каждый кортеж содержит единственное значение некоторого типа в позиции каждого атрибута; иными словами, каждое отношение находится в первой нормальной форме. С учетом этого факта простительным выглядит вопрос – а зачем вообще говорить об этом понятии (и, в частности, называть эту нормальную форму «первой»). Уверен, вы знаете, что причина в том, что (а) мы можем обобщить его на переменные-отношения (а не только на отношения) и (б) мы можем определить для переменных-отношений нормальные формы «более высокого порядка», которые оказываются важны при проектировании баз данных. Другими словами, 1НФ – это основа, на которой строятся нормальные формы более высокого порядка. Но само по себе это понятие не так уж важно.

Примечание

Должен добавить, что 1НФ – одно из тех понятий, определение которых со временем эволюционировало. Раньше оно означало, что любой кортеж должен содержать единственное «атомарное» значение в позиции каждого атрибута. Но впоследствии мы осознали (и я пытался показать это в тексте главы), что понятие атомарности значения данных не имеет объективного смысла. Развернутое обсуждение 1НФ можно найти в статье «What First Normal Form Really Means» (см. приложение D).

Упражнение 2.10. Типом X является тип, назовем его T, определенный в качестве типа результата последнего выполненного при вычислении X оператора – «самого внешнего оператора». Этот тип важен, потому что благодаря ему X можно использовать точно в тех (то есть в тех и только тех) местах, где допустимо появление литерала типа T.

Упражнение 2.11.

```
OPERATOR CUBE ( I INTEGER ) RETURNS INTEGER ;
   RETURN I * I * I ;
END OPERATOR ;
```

Упражнение 2.12.

```
OPERATOR FGP ( P POINT ) RETURNS POINT ;
   RETURN POINT ( F ( THE_X ( P ) ) , G ( THE_Y ( P ) ) ) ;
END OPERATOR ;
```

Упражнение 2.13. Следующий тип отношения является типом переменной-отношения «поставщики» S:

```
RELATION { SNO CHAR , SNAME CHAR , STATUS INTEGER , CITY CHAR }
```

Сама переменная-отношение S является переменной этого типа. И любое допустимое значение этой переменной – например, показанное на рис. 1.3 в главе 1, – является значением этого типа.

Упражнение 2.14. Определения на SQL даны в ответе к упражнению 1.13 в главе 1. Вот определения на **Tutorial D**:

```
VAR P BASE RELATION
  { PNO CHAR , PNAME CHAR , COLOR CHAR , WEIGHT FIXED , CITY CHAR }
  KEY { PNO } ;

VAR SP BASE RELATION
  { SNO CHAR , PNO CHAR , QTY INTEGER }
  KEY { SNO }
  FOREIGN KEY { SNO } REFERENCES S
  FOREIGN KEY { PNO } REFERENCES P ;
```

Перечислим некоторые различия между определениями на SQL и **Tutorial D**:

- Как отмечалось в ответе к упражнению 1.13 в главе 1, в SQL ключи и внешние ключи определяются вместе со столбцами таблицы[1] внутри одной и той же пары скобок. Из-за этого трудно определить, из чего состоит собственно тип. (На самом деле SQL вообще не поддерживает понятие типа отношения, как мы увидим в главе 3.)
- Порядок перечисления столбцов слева направо в SQL имеет значение. Дальнейшее обсуждение см. в главе 5.
- Таблицы в SQL не обязательно должны иметь ключи.

Тот факт, что переменная-отношение (скажем, *P*) принадлежит определенному типу, важен по следующим причинам:

- Переменной-отношению *P* можно присвоить в качестве значения только отношение соответствующего типа.
- Ссылка на переменную-отношение *P* может встречаться всюду, где допустим литерал соответствующего типа (как, например, в выражении P JOIN SP), и в этом случае обозначает отношение, которое

[1] Как и еще некоторые элементы, выходящие за рамки настоящего обсуждения.

является текущим значением переменной-отношения в некоторый момент времени. (Другими словами, *ссылка на переменную-отношение* – допустимое реляционное выражение языка **Tutorial D**; отметим, однако, что аналогичное замечание неприменимо к SQL, по крайней мере, не на 100 процентов.) Дальнейшее обсуждение см. в главе 6.

Упражнение 2.15. a. Недопустимо; LOCATION = CITY('London'). **b.** Допустимо; BOOLEAN. **c.** Предположительно допустимо; MONEY (я предполагаю, что умножение значения типа MONEY на целое число дает значение типа MONEY). **d.** Недопустимо; BUDGET+MONEY(50000). **e.** Недопустимо; ENO > ENO('E2'). **f.** Недопустимо; NAME(THE_C(ENAME)‖THE_C(DNAME)) (я предполагаю, что допустимое представление типа NAME состоит из единственного компонента C типа CHAR). **g.** Недопустимо; CITY(THE_C(LOCATION)‖'burg') (я предполагаю, что допустимое представление типа CITY состоит из единственного компонента C типа CHAR). *Примечание*: В ответах к этому упражнению я также предполагаю, что у типа *T* имеется селектор с таким же именем. Дальнейшее обсуждение см. в главе 8.

Упражнение 2.16. Такая операция логически означает замену одного типа другим, а не «изменение типа» (типы – не переменные). Рассмотрим следующие соображения. Прежде всего, операция определения типа не создает соответствующее множество значений; концептуально эти значения уже существуют и всегда будут существовать (возьмите, к примеру, тип INTEGER). На самом деле операция «определить тип» (предложение TYPE в **Tutorial D** – см. главу 8) лишь вводит имя, по которому на это множество значений можно будет ссылаться. Аналогично удаление типа не уничтожает соответствующие значения, а лишь удаляет имя, созданное в ходе операции «определить тип». Отсюда следует, что «изменение типа» в действительности означает удаление имени типа с последующим созданием такого же имени, которое теперь ссылается на другое множество значений. Конечно, ничто не мешает ради упрощения поддержать некую сокращенную нотацию «изменить тип» – и в SQL такой оператор есть, – но использование сокращения не означает, что мы в действительности «изменяем тип».

Упражнение 2.17. Пустой тип, конечно, допустим; однако определять переменную такого типа особого смысла не имеет, потому что ей нельзя присвоить никакого значения! Однако, несмотря на это, пустой тип оказывается крайне важен в связи с наследованием типов, но эта тема (к сожалению) выходит за рамки настоящей книги. Дополнительную информацию см. в написанной Хью Дарвеном и мной книге «Databases, Types, and the Relational Model: The Third Manifesto» (см. приложение D).

Упражнение 2.18. Пусть *T* – тип SQL, для которого оператор «=» не определен, и пусть *C* – столбец типа *T*. Тогда *C* не может быть частью

потенциального или внешнего ключа, не может быть частью аргумента DISTINCT, GROUP BY или ORDER BY и в терминах *C* не могут быть определены ограничения, соединения, объединения, пересечения и разности. А как насчет таких конструкций уровня реализации, как индексы? Вероятно, имеются и другие последствия.

Далее, пусть *T* – тип SQL, для которого семантика оператора «=» определена пользователем, и пусть *C* – столбец типа *T*. Тогда результат включения *C* в состав потенциального или внешнего ключа, а также применения к нему DISTINCT или GROUP BY (и т. д. и т. п.) тоже должен быть определен пользователем – в лучшем случае, а в худшем случае окажется непредсказуемым.

Упражнение 2.19. Вот тривиальный пример такого нарушения. Пусть *X* – строка символов 'AB ' (обратите внимание на пробел в конце), а Y – строка символов 'AB', и пусть для соответствующей схемы упорядочения задана опция PAD SPACE. Тогда сравнение X = Y возвращает TRUE, но при этом вызовы операторов CHAR_LENGTH(X) и CHAR_LENGTH(Y) возвращают 3 и 2 соответственно. Предлагаю вам детально рассмотреть последствия такого положения вещей, но должно быть ясно, что это приведет к проблемам при выполнении операций DISTINCT, GROUP BY и ORDER BY наряду со многими другими (а также в таких конструкциях уровня реализации, как индексы).

Упражнение 2.20. Потому что (а) они логически излишни, (б) чреваты ошибками, (в) их не может использовать конечный пользователь, (в) неудобны для работы – в частности, с указателем ассоциировано направление, отсутствующее для не-указательных типов, и (г) подрывают идею наследования типов. (Детали последнего замечания выходят за рамки этой книги.) Есть и другие причины. Дополнительную информацию см. в цитированной ранее работе «Inclusion Dependencies and Foreign Keys».

Упражнение 2.21. Один ответ связан с null-значениями; если мы «устанавливаем X в null» (это не означает присваивания X некоторого значения, потому что null – не значение, но неважно), то сравнение X = NULL заведомо не дает TRUE. Есть и много других примеров, не относящихся к null-значениям. Предлагаю вам самостоятельно поразмыслить над последствиями.

Упражнение 2.22. Нет! (Какой базе данных принадлежит, к примеру, тип INTEGER?) В некотором важном смысле весь механизм типов и управления типами ортогонален тематике баз данных и управления базами данных. Мы можем даже представить себе «администратора типов», в задачу которого входит следить за типами аналогично тому, как администратор базы данных следит за базами данных.

Упражнение 2.23. Выражение обозначает значение; его можно трактовать как правило вычисления или определения значения. Предложение не обозначает значение; оно приводит к выполнению некото-

рого действия, например, присваивания значения переменной или изменения потока управления. Например, в SQL

```
X + Y
```

является выражением, а

```
SET Z = X + Y ;
```

является предложением.

Упражнение 2.24. RVA-атрибут – это атрибут, типом которого является тип некоторого отношения, а значениями, следовательно, – отношения этого типа. Повторяющаяся группа – это «атрибут» некоторого типа *T*, значениями которого являются не значения типа *T*, – обратите внимание на противоречивую терминологию! – а мультимножества, или множества, или последовательности (и т. д.) значений типа *T*. *Примечание*: В данном случае «атрибут» сам часто принадлежит некоторому типу кортежа (или чему-то, аппроксимирующему тип кортежа). Например, файл в системе, допускающей повторяющиеся группы, может быть составлен из записей, состоящих из поля ENO (номер служащего), поля ENAME (имя служащего) и повторяющейся группы JOBHIST, каждый элемент которой состоит из поля JOB (должность), поля FROM и поля TO (где FROM и TO – даты).

Упражнение 2.25. В SQL подзапрос – это, неформально говоря, табличное выражение, заключенное в скобки. В последующих главах (особенно в главе 12) этот вопрос рассматривается подробнее.

Упражнение 2.26. См. главу 3.

Глава 3

Упражнение 3.1. См. текст главы.

Упражнение 3.2. Два значения любого вида равны тогда и только тогда, когда имеют одно и то же значение! В частности, (а) два кортежа *tx* и *ty* равны тогда и только тогда, когда имеют одинаковые атрибуты *A1, A2, ..., An* и для любого *i* (*i* = 1, 2, ..., *n*) значение *vx* атрибута *Ai* кортежа *tx* равно значению *vy* атрибута *Ai* кортежа *ty*; (б) два отношения *rx* и *ry* равны тогда и только тогда, когда имеют одинаковые заголовки и одинаковые тела.

Упражнение 3.3. Вызовы селекторов кортежей на языке **Tutorial D**:

```
TUPLE { PNO 'P1' , PNAME 'Nut' ,
                   COLOR 'Red' , WEIGHT 12.0 , CITY 'London' }

TUPLE { SNO 'S1' , PNO 'P1' , QTY 300 }
```

Аналоги на SQL (вызовы «конструкторов значений строк»):

```
ROW ( 'P1' , 'Nut' , 'Red' , 12.0 , 'London' )
```

```
ROW ( 'S1' , 'P1' , 300 )
```

Обратите внимание на отсутствие имен столбцов (или имен «полей», если использовать терминологию SQL) и зависимость от упорядочения столбцов слева направо в SQL-выражениях. Ключевое слово ROW можно опустить без потери смысла.

Упражнение 3.4. Следующий вызов селектора обозначает отношение, состоящее из двух кортежей:

```
RELATION { TUPLE { SNO 'S1' , PNO 'P1' , QTY 300 } ,
           TUPLE { SNO 'S1' , PNO 'P2' , QTY 200 } }
```

Аналог на SQL (вызов «конструктора табличного значения»):

```
VALUES ROW ( 'S1' , 'P1' , 300 ) ,
       ROW ( 'S1' , 'P2' , 200 )
```

В обоих вызовах «конструкторов значений строк» можно при желании опустить ключевое слово ROW. Кстати, отсутствие скобок вокруг вызовов этих конструкторов значений строк – не ошибка. На самом деле, SQL-выражение

```
VALUES ( ROW ( 'S1' , 'P1' , 300 ) ,
         ROW ( 'S1' , 'P2' , 200 ) )
```

(синтаксически вполне допустимое) обозначает нечто совершенно иное! См. упражнение 3.10.

Упражнение 3.5. Следующий перечень основан на приведенном в моей книге «An Introduction to Database Systems» (см. приложение D).

- С любым атрибутом в заголовке отношения связано имя типа, однако эти имена типов опускаются в таблицах (я имею в виду графическое изображение отношений в виде таблиц).
- С любым компонентом любого кортежа в теле отношения связаны имя типа и имя атрибута, но эти имена типов и атрибутов обычно опускаются на рисунках в виде таблиц.
- Значение любого атрибута любого кортежа в теле отношения является значением соответствующего типа, но эти значения (или, точнее, обозначающие их литералы) обычно изображаются на рисунках в сокращенной форме – например, S1 вместо 'S1'.
- Столбцы таблицы упорядочены слева направо, а атрибуты отношения – нет. Одним из следствий такого положения вещей является тот факт, что столбцы могут иметь повторяющиеся имена или вообще не иметь имени. Рассмотрим, к примеру такое SQL-выражение:

```
SELECT DISTINCT S.CITY, S.STATUS * 2, P.CITY
FROM   S, P
```

Какие имена имеют столбцы результата вычисления этого выражения?

- Строки таблицы упорядочены сверху вниз, а кортежи отношения – нет.
- Таблица может содержать строки-дубликаты, а отношение никогда не содержит кортежей-дубликатов.
- Обычно считается, что таблица имеет хотя бы один столбец, тогда как отношение может вообще не иметь атрибутов (см. раздел «TABLE_DUM и TABLE_DEE» в тексте главы).
- Таблицы (по крайней мере, в SQL) могут включать null-значения, а отношения – нет.
- Таблицы являются «плоскими», или двумерными, а отношения n-мерны.

Упражнение 3.6. Вот пример исключения: так как никакое отношение в базе данных не может иметь атрибутов указательного типа, то никакой кортеж отношения также не может иметь атрибутов указательного типа. Другое исключение сформулировать труднее, но сводится оно к тому, что если кортеж t имеет заголовок $\{H\}$, то никакой атрибут t нельзя выразить в терминах типа кортежа или отношения с таким же заголовком $\{H\}$ на любом уровне вложенности.

Ниже приведены примеры (а) кортежа, атрибутом которого является кортеж, и (б) кортежа, атрибутом которого является отношение:

```
TUPLE { NAME 'Superman' ,
        ADDR TUPLE { STREET '1600 Pennsylvania Ave.' ,
                     CITY   'Washington' ,
                     STATE  'DC' ,
                     ZIP    '20500' } }

TUPLE { SNO 'S2' ,
        PNO_REL RELATION { TUPLE { PNO 'P1' } ,
                           TUPLE { PNO 'P2' } } }
```

Упражнение 3.7. Примером отношения с одним RVA-атрибутом может служить отношение R4 на рис. 2.2 в главе 2; эквивалентное ему отношение без RVA-атрибутов – отношение R1 на рис. 2.1 в главе 2. Вот примеры отношений с двумя RVA-атрибутами:

CNO	TEACHER	TEXT
C1	TNO T2 T4 T5	XNO X1 X2
C2	TNO T4	XNO X2 X4 X5

Предполагаемая семантика такова: *курс CNO может быть прочитан любым преподавателем TNO в TEACHER по любому учебнику XNO в TEXT*. Вот как выглядит отношение без RVA-атрибутов, несущее ту же самую информацию:

CNO	TNO	XNO
C1	T2	X1
C1	T2	X2
C1	T4	X1
C1	T4	X2
C1	T5	X1
C1	T5	X2
C2	T4	X2
C2	T4	X4
C2	T4	X5

Что касается отношения с RVA-атрибутом такого, что не существует отношения без RVA-атрибута, представляющего точно такую же информацию, то простой пример можно получить из рис. 2.2 в главе 2, заменив, к примеру, значение PNO_REL для поставщика S2 пустым отношением:

SNO	PNO_REL
S2	PNO
S3	PNO P2
S4	PNO P2 P4 P5

Однако необязательно привлекать понятие пустого отношения, чтобы получить пример отношения с RVA-атрибутом такого, что не существует отношения без RVA-атрибута, представляющего точно такую же информацию. (*Дополнительное упражнение*: Обоснуйте это замечание! Если сдаетесь, ознакомьтесь с ответом к упражнению B.14 в конце этого приложения.)

Быть может, мне следует пояснить, что я имею в виду, когда говорю, что два отношения представляют одну и ту же информацию. По существу, отношения *r1* и *r2* представляют одну и ту же информацию, если можно отобразить *r1* в *r2* и наоборот средствами операций реляционной алгебры, не внося дополнительной информации ни в том, ни в другом направлении. Например, если взять отношения R4 на рис. 2.2 в главе 2 и R1 на рис. 2.1 в главе 2, то будем иметь:

```
R4 = R1 GROUP ( { PNO } AS PNO_REL )

R1 = R4 UNGROUP ( PNO_REL )
```

Таким образом, каждое отношение можно выразить через другое, поэтому оба представляют одну и ту же информацию.[1] Дополнительное обсуждение операторов GROUP и UNGROUP см. в главе 7.

Упражнение 3.8. TABLE_DEE и TABLE_DUM (для краткости DEE и DUM) – единственные отношения без атрибутов; DEE содержит всего один кортеж (0-кортеж), а DUM вообще не содержит кортежей. В SQL они не поддерживаются, так как таблицы в SQL обязаны иметь по крайней мере один столбец. (Другими словами, SQL-версия реляционной алгебры подобна арифметике без нуля.)

Упражнение 3.9. (*Примечание*: Рекомендую вернуться к этому ответу после прочтения главы 10.) Прежде чем формулировать понятие отношения степени 0, нам необходимо иметь понятие отношения вообще. Общее понятие отношения основано на логике предикатов. Логика предикатов основана на логике высказываний. Логика высказываний основана на значениях истинности TRUE и FALSE. Поэтому если заменить TRUE и FALSE на DEE и DUM, то мы будем ходить по кругу!

Кроме того, было бы несколько странно (мягко говоря), если бы все булевы выражения внезапно превратились в реляционные выражения, и, следовательно, от включающего языка потребовалась бы поддержка типов отношений в качестве типов данных.

Имеет ли вообще смысл определять переменную-отношение степени 0? Трудно, хотя и возможно, представить себе ситуацию, в которой такая переменная-отношение могла бы оказаться полезной, но дело не в этом. Система все равно не должна запрещать определение такой переменной-отношения. Иначе был бы нарушен принцип ортогональности, а такие нарушения рано или поздно обязательно выходят боком.

Упражнение 3.10. Первое обозначает SQL-таблицу из четырех строк (три разных и одна – дубликат). Второе обозначает SQL-таблицу из одной строки, состоящей из четырех значений «полей», каждое из которых, в свою очередь, является строкой. Обратите внимание, что все поля (в обоих случаях) не именованы.

Упражнение 3.11. Данное выражение семантически эквивалентно такому:

---

[1] Еще одна полезная неформальная характеристика формулируется так: отношения *r1* и *r2* представляют одну и ту же информацию, если для любого запроса *q1*, который можно предъявить к *r1*, существует запрос *q2* к *r2*, дающий тот же самый результат (и наоборот).

```
SELECT SNO
FROM   S
WHERE  STATUS > 20
OR   ( STATUS = 20 AND SNO > 'S4' )
OR     STATUS IS NULL
OR     SNO IS NULL
```

Упражнение 3.12. См. текст главы.

Упражнение 3.13. См. текст главы.

Упражнение 3.14. EXISTS (*tx*), где *tx* – SQL-аналог реляционного выражения *rx* (следовательно, *tx* – табличное выражение SQL).

Упражнение 3.15. См. текст главы.

Упражнение 3.16. AS используется во фразах SELECT (для назначения имен столбцам); во фразах CREATE VIEW и FROM (для назначения имен переменным кортежа – причем синтаксис назначения имен *столбцам* в этом контексте не требует использования AS); во фразах WITH и в других контекстах, не рассматриваемых в этой книге.

Были также заданы вопросы, (а) в каких случаях это ключевое слово необязательно; (б) в каких случаях фраза AS принимает вид «<что-то> AS имя»; (в) в каких случаях она принимает вид «имя AS <что-то>». *Эти вопросы оставлены без ответа.*

Глава 4

Упражнение 4.1. Чтобы должным образом возразить на этот довод, потребовалось бы больше места, чем мы располагаем, но смысл сводится к так называемому *принципу тождества неразличимых* (см. приложение А). Пусть *a* и *b* – произвольные сущности, например две копейки. Если не существует никакого способа отличить *a* от *b*, то мы имеем не две сущности, а одну! Справедливо, что в некоторых случаях мы можем использовать одну сущность вместо другой, но этого еще недостаточно, чтобы сделать их неразличимыми (фактически существует логическое различие между взаимозаменяемостью и неразличимостью, а аргументы типа «дубликаты естественным образом встречаются в реальном мире» игнорируют это различие). Детальный анализ этого вопроса можно найти в статье «Double Trouble, Double Trouble» (см. приложение D).

Упражнение 4.2. Строго говоря, перед тем как отвечать на этот вопрос, мы должны уточнить, что означают WHERE и UNION при наличии дубликатов. Подробно этот вопрос освещается в работе «The Theory of Bags: An Investigative Tutorial» (см. приложение D); здесь я лишь скажу, что если принять определения SQL, то это правило, безусловно, неприменимо. На самом деле, оно неприменимо даже к UNION ALL или UNION DISTINCT! В качестве примера предположим, что *r* – SQL-таблица всего с одним столбцом – назовем его C, – содер-

жащая две строки, каждая из которых содержит одно значение *v*. Ниже показан ряд выражений вместе с их результатами:

```
SELECT C
FROM   r
WHERE  TRUE
OR     TRUE

Результат: v * 2.

SELECT C
FROM   r
WHERE  TRUE
UNION  DISTINCT
SELECT C
FROM   r
WHERE  TRUE

Результат: v * 1.

SELECT C
FROM   r
WHERE  TRUE
UNION  ALL
SELECT C
FROM   r
WHERE  TRUE

Результат: v * 4.
```

Примечание

Но если все ALL (явные и неявные) в приведенных выше выражениях заменить на DISTINCT, то результаты будут совершенно другими. Какой вывод вы отсюда сделаете?

Упражнение 4.3. Замечания, аналогичные высказанным в ответе на предыдущее упражнение, сохраняют силу и в этом случае. Детали я снова опускаю, но смысл состоит в том, это правило почти наверняка неприменимо. Предлагаю вам придумать контрпример.

Упражнение 4.4. Насколько я могу судить, единственный способ разрешить эту неоднозначность состоит в том, чтобы определить отображение каждой из (мультимножества) таблиц-аргументов на настоящее множество и аналогично определить отображение таблицы-результата (мультимножества), то есть требуемого декартова произведения, на настоящее множество. (Для определения такого отображения необходимо сопоставить каждой строке уникальный идентификатор.) На самом деле, мне кажется, что неудачная попытка стандарта дать определение – лишнее доказательство того, что самые базовые концепции в языке SQL (в частности, идея о допустимости

строк-дубликатов в таблицах) имеют фундаментальные дефекты, которые нельзя устранить, не отказавшись от концепции целиком.

Упражнение 4.5. Не думаю, что эту проблему можно решить.

Упражнение 4.6. *Без ответа!*

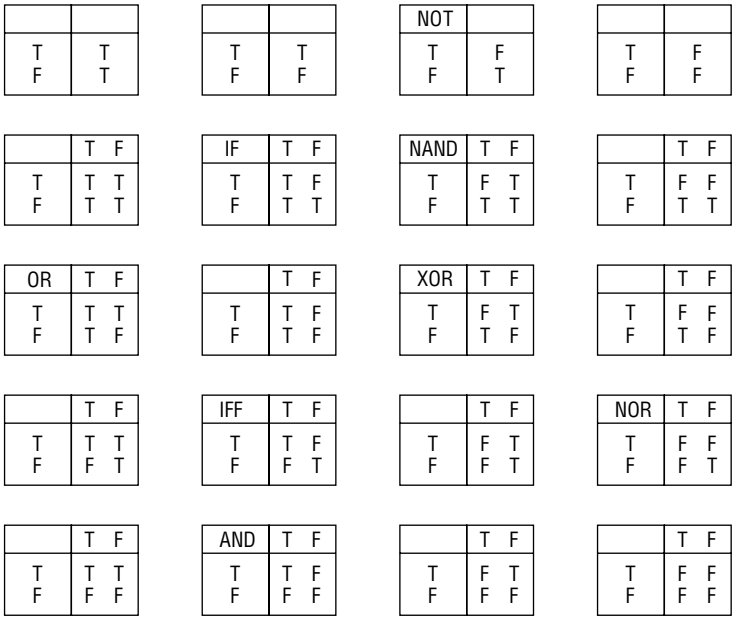
Упражнение 4.7. Вопрос ставился так: на ваш взгляд, встречаются ли null-значения в реальном мире естественным образом? Только вы можете ответить на этот вопрос – но, если вы отвечаете «да», то полагаю, что ваши доводы следует подвергнуть тщательному анализу. Например, рассмотрим утверждение «зарплата Джо составляет 50 000 долларов». Это утверждение либо истинно, либо ложно. Конечно, вы можете не знать, истинно оно или ложно, но это не имеет никакого отношения к его реальной истинности. В частности, не знать величину зарплаты Джо и сказать, что она равна null, – совершенно разные вещи! «Зарплата Джо составляет 50 000 долларов» – это утверждение о реальном мире. «Зарплата Джо составляет null» – утверждение о вашей осведомленности (точнее, об отсутствии таковой). Мы ни в коем случае не должны смешивать эти два очень разных вида утверждений в одном отношении или в одной переменной-отношении! *Примечание*: Обсуждение *предикатов переменных-отношений* в следующей главе должно дать вам новую пищу для размышлений в этом направлении.

Предположим, что вам необходимо представить свое незнание зарплаты Джо в бумажной анкете. Вы напишете в соответствующей графе null? Не думаю! Скорее, вы оставите ее незаполненной или впишете вопросительный знак или слово «неизвестно» или еще что-то в этом роде. И это что-то, будь то незаполненная графа, вопросительный знак или слово «неизвестно», есть значение, а не null (напомню, что о null твердо можно сказать только одно – это не значение). Поэтому лично я не считаю, что у null «есть естественное место в реальном мире».

Упражнение 4.8. Истинно (хотя и не в SQL!). Null – это маркер, обозначающий отсутствие информации. UNKNOWN – это значение, точно такое же, как TRUE и FALSE. Между тем и другим есть логическое различие, и смешивание их в SQL – логическая ошибка (Я бы сказал, серьезная логическая ошибка, но все логические ошибки серьезны по определению).

Упражнение 4.9. Да, поддерживает; аналог MAYBE *p* в SQL – *p* IS UNKNOWN.

Упражнение 4.10. В 2VL существует 4 одноместных связки и 16 двуместных, им соответствуют 4 возможных одноместных таблицы истинности и 16 двуместных. Вот эти таблицы истинности (я указал общепринятые имена, например NOT, AND и OR, для тех, у которых такие имена существуют):



В 3VL существует 27 (3 в третьей степени) одноместных связок и 19683 (3 в степени 3^2) двуместных. (В общем случае в n-значной логике (nVL) существует n в степени n одноместных и n в степени n^2 двуместных связок). Из этого факта можно вывести много следствий, и одно из самых очевидных состоит в том, что 3VL гораздо сложнее, чем 2VL (намного сложнее, чем думает или, по крайней мере, готово признать большинство людей, полагающих, что null – это хорошо).

Упражнение 4.11. 2VL функционально полна, если она поддерживает NOT и либо AND, либо OR (дополнительное обсуждение см. в ответе к упражнению 10.2). И как выясняется, 3VL в смысле SQL – при некоторой весьма снисходительной трактовке этого термина! – тоже функционально полна. Более детальное исследование этого вопроса см. в статье «Is SQL's Three-Valued Logic Truth Functionally Complete?» (см. приложение D).

Упражнение 4.12. В 3VL это не тавтология, потому что, если *bx* принимает значение UNKNOWN, то и все выражение принимает значение UNKNOWN. Однако в 3VL существует аналогичная тавтология, а именно: *bx* OR NOT *bx* OR MAYBE *bx*. *Примечание*: Это объясняет, почему в SQL результаты запросов «Получить всех поставщиков в Лондоне» и «Получить всех поставщиков не в Лондоне» в совокупности не обязательно покрывают множество всех поставщиков; не исключено, что придется еще выполнить запрос «Получить всех поставщиков, которые, возможно, находятся в Лондоне». Подумайте о том, что отсюда следует с точки зрения переписывания запро-

сов... не говоря уже о возможности серьезных ошибок (со стороны как пользователей, так и системы). Подчеркну еще раз: очень естественно предполагать, что тавтологии в 2VL являются также тавтологиями и в 3VL, но в общем случае это не так.

Упражнение 4.13. Это не противоречие в 3VL, потому что, если *bx* принимает значение UNKNOWN, то и все выражение принимает значение UNKNOWN. Однако в 3VL существует аналогичное (довольно хитрое!) противоречие, а именно: *bx* AND NOT *bx* AND NOT MAYBE *bx*. *Примечание*: Как и следовало ожидать, такое положение вещей ведет к последствиям, аналогичным упомянутым в ответе к предыдущему упражнению.

Упражнение 4.14. В 3VL (а) *r* JOIN *r* не обязательно равно *r* и (б) INTERSECT не является частным случаем JOIN. Причина состоит в том, что (а) для соединения два null-значения не считаются «равными», но (б) для пересечения они равны (еще одна из очень многих – бесконечно многих? – нелепостей, к которым неизбежно ведут null-значения). Однако операция TIMES все-таки является частным случаем JOIN, как и в 2VL.

Упражнение 4.15. Вот как формулируются правила. Пусть *x* – SQL-строка. Предположим для простоты, что *x* имеет всего два компонента: *x1* и *x2* (разумеется, в этом порядке слева направо!). Тогда *x* IS NULL по определению эквивалентно *x1* IS NULL AND *x2* IS NULL, а *x* IS NOT NULL по определению эквивалентно *x1* IS NOT NULL AND *x2* IS NOT NULL. Отсюда следует, что данная строка не является ни null, ни не-null... Какой вывод вы отсюда сделаете?

Кстати, один из рецензентов заметил, что он никогда не думал, что о строках можно говорить, как о null-значениях. Но строки – это значения (так же, как значениями являются кортежи и отношения), поэтому идея о том, что строка может быть неизвестной, имеет такое же право на существование, как идея о неизвестной величине зарплаты. Поэтому, если концепция представления неизвестного значения с помощью null вообще имеет смысл – а лично я считаю, что не имеет, – то она, безусловно, применима и к строкам (и к таблицам, и вообще к любым мыслимым значениям) точно так же, как к скалярам. И, как следует из этого упражнения, SQL пытается поддержать эту позицию (по крайней мере, для строк, хотя и не для таблиц), правда, безуспешно.[1]

Упражнение 4.16. Нет. Вот их таблицы истинности в 3VL:

NOT	
T	F
U	U
F	T

IS NOT TRUE	
T	F
U	T
F	T

[1] Логично было бы поддержать ее и для таблиц, но это не сделано.

Упражнение 4.17. Нет. Для определенности рассмотрим случай, когда *x* – SQL-строка. Предположим для простоты (как в ответе к упражнению 4.15), что *x* имеет всего два компонента: *x1* и *x2*. Тогда *x* IS NOT NULL по определению эквивалентно *x1* IS NOT NULL AND *x2* IS NOT NULL, а NOT (x IS NULL) по определению эквивалентно *x1* IS NOT NULL OR *x2* IS NOT NULL. Какой вывод вы отсюда сделаете?

Упражнение 4.18. Такая трансформация некорректна, и, чтобы убедиться в этом, достаточно рассмотреть, что произойдет, когда EMP.DNO содержит null (вы удивлены?). Вывод все тот же: и пользователи, и система могут допустить ошибки (и такие ошибки действительно случались).

Упражнение 4.19. Этот запрос означает «Получить поставщиков, относительно которых известно, что они не поставляют деталь P2» (обратите внимание на тонкое различие между формулировками «известно, что не поставляют» и «неизвестно, что поставляют»). Это не то же самое, что ««Получить поставщиков, которые не поставляют деталь P2». Эти две формулировки не эквивалентны (рассмотрите, например, случай, когда единственное «значение» номера поставщика, соответствующее детали P2 в таблице SP, – это null).

Упражнение 4.20. Никакие два из этих трех утверждений не эквивалентны. Утверждение (a) следует из правил трехзначной логики в SQL; утверждение (b) следует из определения оператора UNIQUE в SQL; утверждение (c) следует из определения дубликатов в SQL. В частности, если *k1* и *k2* одновременно равны null, то (a) дает UNKNOWN, (b) дает FALSE, а (c) дает TRUE. *Примечание*: для справки приведу правила, на которые ссылаюсь:

- В трехзначной логике в интерпретации SQL сравнение *k1* = *k2* дает TRUE, если *k1* и *k2* отличны от null и равны; FALSE, если *k1* и *k2* отличны от null и не равны, и UNKNOWN в остальных случаях.
- В определении SQL-оператора UNIQUE сравнение *k1* = *k2* дает TRUE тогда и только тогда, когда *k1* и *k2* отличны от null и равны, и FALSE в противном случае.
- В SQL *k1* и *k2* называются дубликатами, если (a) они отличны от null и равны или (б) оба совпадают с null.

Примечание

Выше слово «равны» относится к определению оператора «=» в SQL (несколько своеобразному) (см. главу 2). Дополнительное упражнение: как вы думаете, эти правила разумны? Обоснуйте свой ответ.

Упражнение 4.21. Результаты выполнения операторов INTERSECT ALL и EXCEPT ALL могут содержать дубликаты, но только если они уже присутствовали в исходных операндах.

Упражнение 4.22. Да! (Мы не хотим дубликатов в базе данных, но это не означает, что мы не хотим дубликатов и в других местах.)

Упражнение 4.23. Очень хороший вопрос.

Глава 5

Упражнение 5.1. В некоторых отношениях кортеж действительно напоминает запись, а атрибут – поле, но это сходство очень приблизительное. Переменную-отношение следует рассматривать не просто как файл, а как «файл с определенной дисциплиной». Эта дисциплина приводит к существенному упрощению структуры данных, видимой пользователю, а, значит, и к соответственному упрощению операторов, необходимых для работы с этими данными, и пользовательского интерфейса в целом. Так в чем же состоит эта дисциплина? В том, что записи не упорядочены сверху вниз; что поля не упорядочены слева направо; что отсутствуют null-значения; что нет повторяющихся групп; что нет указателей; что нет анонимных полей (и т. д. и т. п.). Отчасти поэтому гораздо лучше трактовать переменную-отношение следующим образом: заголовок представляет некоторый предикат, а тело в конкретный момент времени представляет экстенцию этого предиката в этот момент.

Упражнение 5.2. Говоря неформально, эта фраза означает «Обновить атрибут STATUS в кортежах для всех поставщиков в Лондоне». Но кортежи (а тем более значения атрибутов кортежей) – это значения, и их просто невозможно обновить по определению. Поэтому вот более точная формулировка:

- Пусть отношение *s* – текущее значение переменной-отношения S.
- Пусть *ls* – ограничение *s*, для которого значением CITY является Лондон.
- Пусть *ls′* – отношение, идентичное *ls* с тем отличием, что значение STATUS в каждом кортеже такое, как указано в операции UPDATE.
- Пусть *s′* – отношение, обозначенное выражением (*s* MINUS *ls*) UNION *ls′*.
- Тогда *s′* присваивается S.

Упражнение 5.3. Потому что реляционные операции принципиально определены над множествами, а операции «позиционного обновления» в SQL – над кортежами (точнее, строками). Хотя операции над множествами кардинальности 1 иногда – быть может, даже часто – допустимы, они могут и не сработать. Например, операции обновления кортежа могут некоторое время работать нормально, а потом – после изменения ограничения целостности – перестать.

Упражнение 5.4. Эти предложения не эквивалентны. Для первого исходной является таблица *t1*, обозначенная заданным *табличным* подзапросом; для второго исходной является таблица *t2*, содержащая только строку, обозначенную заданным *однострочным* подзапросом (то есть аргументом VALUES). Если таблица S включает строку для поставщика S6, то *t1* и *t2* идентичны. Но если S не включает такую строку, то *t1* пуста, а *t2* содержит строку, состоящую только из null.

Упражнение 5.5. *Принцип присваивания* утверждает, что после присваивания значения *v* переменной *V* сравнение *V* = *v* должно возвращать TRUE. SQL нарушает этот принцип в случае, если «*v* есть null», он также нарушает его для некоторых присваиваний символьных строк и, безусловно, нарушает для любого типа, для которого не определен оператор «=», в том числе для типа XML, а также (вообще говоря) для некоторых пользовательских типов. *Негативные последствия*: их слишком много, чтобы перечислить здесь.

Упражнение 5.6. Как и в тексте главы, в следующих определениях я предполагаю, что определены некоторые пользовательские типы:

```
CREATE TABLE TAX_BRACKET
  ( LOW        MONEY   NOT NULL ,
    HIGH       MONEY   NOT NULL ,
    PERCENTAGE INTEGER NOT NULL ,
    UNIQUE ( LOW ) ,
    UNIQUE ( HIGH ) ,
    UNIQUE ( PERCENTAGE ) ) ;

CREATE TABLE ROSTER
  ( DAY   DAY_OF_WEEK NOT NULL ,
    HOUR  TIME_OF_DAY NOT NULL ,
    GATE  GATE        NOT NULL ,
    PILOT NAME        NOT NULL ,
    UNIQUE ( DAY, HOUR, GATE ) ,
    UNIQUE ( DAY, HOUR, PILOT ) ) ;

CREATE TABLE MARRIAGE
  ( SPOUSE_A         NAME NOT NULL ,
    SPOUSE_B         NAME NOT NULL ,
    DATE_OF_MARRIAGE DATE NOT NULL ,
    UNIQUE ( SPOUSE_A, DATE_OF_MARRIAGE ) ,
    UNIQUE ( DATE_OF_MARRIAGE, SPOUSE_B ) ,
    UNIQUE ( SPOUSE_B, SPOUSE_A ) ) ;
```

Упражнение 5.7. Поскольку ключи представляют собой ограничения, а ограничения применяются к переменным, а не к значениям. (Но при этом, безусловно, допустимо, а иногда и полезно, рассматривать подмножество *k* заголовка отношения *r* так, будто оно является «ключом для *r*», если это подмножество уникально и неприводимо относительно кортежей *r*. Однако, строго говоря, такой взгляд

некорректен, может стать причиной путаницы и, конечно, гораздо менее полезен, чем взгляд на ключи, как на принадлежность переменных-отношений, а не отношений.)

Упражнение 5.8. Вот еще одна. Предположим, что переменная-отношение *A* обладает «неприводимым ключом», который состоит из дизъюнктного объединения *K* и *X*, где *K* и *X* – подмножества заголовка *A*, и *K* – настоящий ключ. Тогда переменная-отношение *A* удовлетворяет функциональной зависимости $K \rightarrow X$. Предположим далее, что переменная-отношение *B* имеет внешний ключ, ссылающийся на этот «неприводимый ключ» *A*. Тогда *B* тоже удовлетворяет функциональной зависимости $K \rightarrow X$; в результате *B*, вероятно, обладает некоторой избыточностью (фактически, она, скорее всего, не находится в нормальной форме Бойса-Кодда).

Упражнение 5.9. Ключи – это множества атрибутов (фактически каждый ключ представляет собой некоторое подмножество заголовка), а значениями ключей по определению являются кортежи, даже если эти кортежи состоят из одного атрибута. Так, например, ключом переменной-отношения «детали» P является {PNO}, а не просто PNO, и значением этого ключа для детали P1 является кортеж TUPLE {PNO 'P1'}, а не просто 'P1'.

Упражнение 5.10. Пусть *m* – наименьшее целое число, большее или равное *n/2*. *R* будет будет иметь максимально возможное количество ключей, если либо (а) каждое различающееся множество *m* атрибутов есть ключ, либо (б) *n* нечетно и каждое различающееся множество *m-1* атрибутов есть ключ. В любом случае отсюда следует, что максимальное поличество ключей *R* равно *n!/(m!*(n-m)!)*. *Примечание*: выражение *n!* произносится либо как «*n* факториал», либо как «факториал *n*» и по определению равно произведению *n * (n-1) * ... * 2 * 1*. Обе переменные-отношения TAX_BRACKET и MARRIAGE (см. упражнение 5.6) дают примеры переменных-отношений с максимально возможным количеством ключей, как и любая переменная-отношение степени 0. (Если *n = 0*, то эта формула принимает вид 0!/(0!*0!), а 0! равно 1.)

Упражнение 5.11. Суперключом называется подмножество заголовка, обладающее свойством уникальности. Ключом называется суперключ, обладающий свойством неприводимости. Все ключи являются суперключами, но «большинство» суперключей ключами не являются.

Понятие *подключа* может быть полезно при изучении нормализации. Вот его определение. Пусть *X* – подмножество заголовка переменной-отношения *R*; тогда *X* называется подключом *R*, если существует такой ключ *K* для *R*, что *K* является надмножеством *X*. Например, каждое из следующих подмножеств является подключом переменной-отношения SP: {SNO,PNO}, {SNO}, {PNO} и {}. Отмечу,

что пустое множество {} является подключом любой допустимой переменной-отношения R.

Упражнение 5.12. Пример данных:

EMP

ENO	MNO
E4	E2
E3	E2
E2	E1
E1	E1

Здесь я допустил, что некоторый служащий (а именно E1) является собственным начальником, – это один из способов избежать null-значений в подобной ситуации. Другой и, пожалуй, более удачный подход – выделить такие связи в отдельную переменную-отношение, исключив из нее тех служащих, у которых нет начальников.

EMP

ENO	...
E4	...
E3	...
E2	...
E1	...

EM

ENO	MNO
E4	E2
E3	E2
E2	E1

Дополнительное упражнение: Каковы предикаты переменной-отношения EM и обоих вариантов переменной-отношения EMP? (Хорошенько подумав над этим упражнением, вы обнаружите добавочные доводы в пользу второй структуры.)

Упражнение 5.13. Потому что в ней нет необходимости, учитывая, что соответствие между столбцами устанавливается исходя из номера позиции, а не имени. См. обсуждение в тексте главы.

Упражнение 5.14. Отметим, что такая ситуация по определению должна представлять взаимно однозначное соответствие. Один очевидный случай возникает, когда мы расщепляем некоторую переменную-отношение «по вертикали», как в следующем примере:

```
VAR SNT BASE RELATION
  { SNO SNO, SNAME NAME, STATUS INTEGER }
  KEY { SNO }
  FOREIGN KEY { SNO } REFERENCES SC ;

VAR SC BASE RELATION
  { SNO SNO, CITY CHAR }
  KEY { SNO }
  FOREIGN KEY { SNO } REFERENCES SNT ;
```

Одним из последствий такого решения является то, что нам, скорее всего, понадобится механизм для одновременного обновления двух или более переменных-отношений (и, возможно, для одновременно-

го определения двух или более переменных-отношений). См. обсуждение множественного присваивания в главе 8.

Упражнение 5.15. В **Tutorial D** определения выглядят следующим образом (для этого упражнения я предполагаю, что **Tutorial D** поддерживает ссылочные действия CASCADE и NO CASCADE):

```
VAR P BASE RELATION { PNO ... , ... } KEY { PNO } ;

VAR PP BASE RELATION { MAJOR_PNO ... , MINOR_PNO ... , QTY ... }
    KEY { MAJOR_PNO , MINOR_PNO }
    FOREIGN KEY { MAJOR_PNO } REFERENCES P
            RENAME ( PNO AS MAJOR_PNO ) ON DELETE CASCADE
    FOREIGN KEY { MINOR_PNO } REFERENCES P
            RENAME ( PNO AS MINOR_PNO ) ON DELETE NO CASCADE ;
```

При таких определениях удаление детали каскадно распространяется на те кортежи, в которых номер этой детали встречается в качестве значения атрибута MAJOR_PNO (то есть данная деталь включает другие детали в качестве компонентов), но не на кортежи, в которых номер этой детали встречается в качестве значения атрибута MINOR_PNO (то есть данная деталь выступает только в роли компонентов других деталей).

Определения на SQL:

```
CREATE TABLE P ( PNO ... , ... , UNIQUE ( PNO ) ) ;

CREATE TABLE PP ( MAJOR_PNO ... , MINOR_P# ... , QTY ... ,
                  UNIQUE ( MAJOR_PNO , MINOR_PNO ) ,
                  FOREIGN KEY ( MAJOR_PNO ) REFERENCES P ( PNO )
                          ON DELETE CASCADE ,
                  FOREIGN KEY ( MINOR_PNO ) REFERENCES P
                          ON DELETE RESTRICT ) ;
```

Упражнение 5.16. Очевидно, что исчерпывающий ответ на этот вопрос дать невозможно. Упомяну лишь ссылочные действия, поддерживаемые в стандарте: NO ACTION (по умолчанию), CASCADE, RESTRICT, SET DEFAULT и SET NULL. Дополнительное упражнение: в чем разница между NO ACTION и RESTRICT?

Упражнение 5.17. Предикат – это функция, возвращающая значение истинности; высказывание – это предикат без параметров. Некоторые примеры имеются в тексте главы, а в главе 10 приведены дополнительные примеры и общее обсуждение этих понятий.

Упражнение 5.18. Переменная-отношение P: *деталь PNO используется на предприятии, называется PNAME, имеет цвет COLOR и вес WEIGHT и хранится на складе в городе CITY.* Переменная-отношение SP: *поставщик SNO поставляет деталь PNO в количестве QTY.*

Упражнение 5.19. Интенцией переменной-отношения *R* называется ее предполагаемая интерпретация. Экстенцией переменной-отношения

R в некоторый момент времени называется множество кортежей, составляющих *R* в этот момент.

Упражнение 5.20. *Без ответа.*

Упражнение 5.21. *Допущение замкнутости мира* говорит (неформально), что все утверждаемое или вытекающее из содержимого базы данных истинно, а все остальное ложно. А *допущение открытости мира* (да, есть и такое) говорит, все утверждаемое или вытекающее из содержимого базы данных истинно, а все остальное неизвестно. Как легко видеть, *допущение открытости мира* прямо приводит к требованию разрешить null-значения и к трехзначной логике, поэтому всячески осуждается. Дополнительную информацию можно найти в статье «*The Closed World Assumption*» (см. приложение D).

Упражнение 5.22. Сказать, что переменная-отношение *R* имеет пустой ключ, – все равно, что сказать, что *R* не может содержать более одного кортежа (поскольку у всех кортежей одинаковы значения пустого множества атрибутов – пустое множество; поэтому если бы *R* содержала два или более кортежей, то мы имели бы нарушение уникальности ключа). А ограничение, состоящее в недопустимости более одного кортежа, безусловно, может оказаться полезным. Придумывание такой ситуации оставляю вам в качестве дополнительного упражнения.

Упражнение 5.23. См. ответ к упражнению 5.17.

Упражнение 5.24. Вопрос, несомненно, имеет смысл, учитывая, что с каждой переменной-отношением ассоциирован некоторый предикат. Однако, какой именно предикат соответствует конкретной переменной-отношению, известно лишь тому, кто ее определил (и, хочется надеяться, пользователю). Например, если я определю переменную-отношение C следующим образом:

```
VAR C BASE RELATION { CITY CHAR } KEY { CITY } ;
```

то ей может соответствовать чуть ли не любой предикат! Например: *CITY – город в Калифорнии*; *CITY – город, в котором находится хотя бы один поставщик*; *CITY – столица некоторого государства* и так далее[1]. Точно так же предикатом переменной-отношения степени 0

```
VAR Z BASE RELATION { } KEY { } ;
```

может быть «почти все, что угодно», с единственным условием (поскольку эта переменная-отношение не имеет атрибутов и, стало быть, соответствующий ей предикат не имеет параметров), что предикат

[1] Или даже *CITY – прозвище чьего-то любимого плюшевого медвежонка*. В определении переменной-отношения CITY нет ничего, свидетельствующего о том, что это город.

должен быть вырожденным, то есть высказыванием. Это высказывание должно принимать значение TRUE, если Z равно TABLE_DEE, и FALSE, если Z равно TABLE_DUM.

Кстати, отмечу, что переменная-отношение Z имеет пустой ключ; однако не следует делать вывод, будто пустые ключи могут быть только у переменных-отношений степени 0 (см. ответ к упражнению 5.22).

Упражнение 5.25. Конечно, нет. На самом деле, «большинство» отношений не являются значениями некоторой переменной-отношения. В качестве тривиального примера отметим, что отношение, обозначенное {CITY}, то есть проекция S на CITY, не является значением никакой переменной-отношения в базе данных о поставщиках и деталях. Поэтому в этой книге, говоря об отношениях, я, конечно, не имею в виду только отношения, являющиеся значениями некоторой переменной-отношения.

Глава 6

Прежде всего, приведу ответы на два упражнения, которые были включены в текст главы. В первом спрашивалось, в чем разница (при наших обычных тестовых данных) между выражениями S JOIN (P{PNO}) и (S JOIN P){PNO}. *Ответ*: Так как S и P{PNO} не имеют общих атрибутов, то первое выражение представляет собой декартово произведение; для нашего примера результат содержит все возможные комбинации кортежа поставщика с кортежем номера детали (говоря неформально), и его кардинальность равна 30. Второе выражение дает номера тех деталей, который находятся в одном городе с некоторым поставщиком (опять же, говоря неформально); для нашего примера результат содержит все номера деталей, кроме P3.

Во втором упражнении спрашивалось, в чем разница между эквисоединением и естественным соединением. *Ответ*: Пусть соединяемые отношения называются *r1* и *r2*, и предположим для простоты, что *r1* и *r2* имеют ровно один общий атрибут *A*. Прежде чем выполнить эквисоединение, необходимо произвести некоторое переименование. Для определенности применим переименование к *r2*: *r3* = *r2* RENAME (*A* AS *B*). Тогда эквисоединение определяется как (*r1* TIMES *r3*) WHERE *A* = *B*. Отметим, в частности, что и *A*, и *B* – атрибуты результата, и что в каждом кортеже результата значения этих атрибутов одинаковы. Если спроецировать результат на множество всех атрибутов, кроме *B*, то получится естественное соединение *r1* JOIN *r2*.

Упражнение 6.1. a. Результат содержит столбцы с повторяющимися именами (а также столбцы упорядочены слева направо). **b.** Столбцы результата упорядочены слева направо. **c.** Результат содержит неименованный столбец (а также столбцы упорядочены слева направо). **d.** Результат содержит строки-дубликаты. **e.** Синтаксическая ошиб-

ка SQL: в S NATURAL JOIN P нет столбца с именем S.CITY.[1] **f.** Все правильно. **g.** Результат содержит строки-дубликаты, и столбцы упорядочены слева направо; кроме того, отсутствует столбец с именем SNO, что может вызвать недоумение.

Упражнение 6.2. Нет! В частности, некоторые операции реляционного деления, которые могли бы предположительно завершиться ошибкой, на самом деле выполняются успешно. Вот несколько примеров (для понимания которых необходимо сначала прочитать соответствующий раздел в главе 7):

a. Пусть отношение PZ имеет тип RELATION {PNO PNO}, и пусть его тело пусто. Тогда выражение

```
SP { SNO , PNO } DIVIDEBY PZ { PNO }
```

сводится к проекции SP{SNO} отношения SP на SNO.

b. Пусть *z* – либо TABLE_DEE, либо TABLE_DUM. Тогда выражение

```
r DIVIDEBY z
```

сводится к *r* JOIN *z*.

c. Пусть отношения *r* и *s* одного типа. Тогда выражение

```
r DIVIDEBY s
```

дает TABLE_DEE, если *r* непусто и каждый кортеж *s* встречается в *r*, и TABLE_DUM – в противном случае.

d. Наконец, *r* DIVIDEBY *r* дает TABLE_DUM, если *r* пусто, и TABLE_DEE – в противном случае.

Упражнение 6.3. Соединение производится по атрибутам SNO, PNO и CITY. Результат выглядит следующим образом:

SNO	SNAME	STATUS	CITY	PNO	QTY	PNAME	COLOR	WEIGHT
S1	Smith	20	London	P1	300	Nut	Red	12.0
S1	Smith	20	London	P4	200	Screw	Red	14.0
S1	Smith	20	London	P6	100	Cog	Red	19.0
S2	Jones	10	Paris	P2	400	Bolt	Green	17.0
S3	Blake	30	Paris	P2	200	Bolt	Green	17.0
S4	Clark	20	London	P4	200	Screw	Red	14.0

Предикат звучит так: *поставщик SNO связан контрактом, называется SNAME, имеет статус STATUS и находится в городе CITY; деталь PNO используется на предприятии, называется PNAME,*

[1] Нет также и столбца с именем S.SNO (есть столбец с неквалифицированным именем SNO); однако существует странное синтаксическое правило, позволяющее обращаться к этому столбцу без уточнения имени. (Я говорю *странное*, потому что его очень трудно точно сформулировать, а, кроме того, оно противоречит интуиции и логически некорректно.)

имеет цвет COLOR и вес WEIGHT и хранится на складе в городе CITY; поставщик SNO поставляет деталь PNO в количестве QTY. Отметим, что оба вхождения SNO в этот предикат относятся к одному и тому же параметру, равно как оба вхождения PNO и оба вхождения CITY.

Простейшая формулировка на SQL выглядит так:

```
S NATURAL JOIN SP NATURAL JOIN P
```

(хотя в зависимости от контекста может оказаться необходимым поставить впереди «SELECT * FROM»).

Упражнение 6.4. В декартовых координатах на двумерной плоскости точки (x,0) и (0,y) являются проекциями точки (x,y) на оси X и Y соответственно; эквивалентно, (x) и (y) – проекции точки (x,y) двумерного пространства на некоторые одномерные пространства. Эти понятия легко обобщаются на *n* измерений (напомню (см. главу 3), что отношения в действительности *n*-мерны).

Упражнение 6.5.

a. SQL-аналог:

```
SELECT DISTINCT CITY
FROM   S NATURAL JOIN SP
WHERE  PNO = 'P2'
```

Примечание

Здесь и далее приводимые мной SQL-выражения не обязательно являются прямыми транслитерациями соответствующих алгебраических аналогов, а чаще «более естественными» формулировками запроса в терминах SQL.

Предикат: *город CITY таков, что в нем находится некоторый поставщик, поставляющий деталь P2.*

CITY
London
Paris

b. SQL-аналог:

```
SELECT *
FROM   P
WHERE  PNO NOT IN
     ( SELECT PNO
       FROM   SP
       WHERE  SNO = 'S2' )
```

Предикат: *деталь PNO используется на предприятии, называется PNAME, имеет цвет COLOR и вес WEIGHT, хранится на складе в городе CITY и не поставляется поставщиком S2.*

PNO	PNAME	COLOR	WEIGHT	CITY
P3	Screw	Blue	17.0	Oslo
P4	Screw	Red	14.0	London
P5	Cam	Blue	12.0	Paris
P6	Cog	Red	19.0	London

c. SQL-аналог:

```
SELECT CITY
FROM   S
EXCEPT CORRESPONDING
SELECT CITY
FROM   P
```

Предикат: *город CITY таков, что в нем находится некоторый поставщик, но не хранится ни одной детали.*

CITY
Athens

d. SQL-аналог:

```
SELECT SNO , PNO
FROM   S NATURAL JOIN P
```

Примечание

В версии на **Tutorial D** тоже нет необходимости проецировать S и P. Как вы думаете, может ли оптимизатор игнорировать их?

Предикат: *поставщик SNO и деталь PNO находятся в одном городе.*

SNO	PNO
S1	P1
S1	P4
S1	P6
S2	P2
S2	P5
S3	P2
S3	P5
S4	P1
S4	P4
S4	P6

e. SQL-аналог:

```
SELECT S.CITY AS SC , P.CITY AS PC
FROM   S , P
```

Предикат: *некоторый поставщик находится в городе SC, и некоторая деталь находится в городе PC.*

SC	PC
London	London
London	Paris
London	Oslo
Paris	London
Paris	Paris
Paris	Oslo
Athens	London
Athens	Paris
Athens	Oslo

Упражнение 6.6. Пересечение и декартово произведение – частные случаи соединения, поэтому здесь их можно игнорировать. Коммутативность объединения и соединения следует непосредственно из того, что оба отношения встречаются в определении симметричным образом. Далее я докажу, что объединение ассоциативно. Пусть t – кортеж. Считая, что символ «≡» обозначает «тогда и только тогда», а символ «∈» – «входит в», имеем:

```
t ∈ r UNION (s UNION u) ≡ t ∈ r OR t ∈ (s UNION u)
                        ≡ t ∈ r OR (t ∈ s OR t ∈ u)
                        ≡ (t ∈ r OR t ∈ s) OR t ∈ u
                        ≡ t ∈ (r UNION s) OR t ∈ u
                        ≡ t ∈ (r UNION s) UNION u
```

Обратите внимание, что в третьей строчке используется ассоциативность операции OR. Доказательство ассоциативности соединения аналогично.

Что касается SQL, то, прежде всего, забудем о существовании null и строк-дубликатов (а что случится, если не забудем?). Тогда:

- SELECT A, B FROM T1 UNION CORRESPONDING SELECT B, A FROM T2 и SELECT B, A FROM T2 UNION CORRESPONDING SELECT A, B FROM T1 не эквивалентны, поскольку дают разные результаты с точки зрения упорядочения столбцов слева направо. Таким образом, объединение в SQL, вообще говоря, не коммутативно (и то же самое относится к пересечению).
- T1 JOIN T2 и T2 JOIN T1 не эквивалентны (в общем случае), поскольку дают разные результаты с точки зрения упорядочения столбцов слева направо. Таким, образом соединение в SQL, вообще говоря, не коммутативно (и то же самое относится к произведению).

Однако все операторы ассоциативны.

Упражнение 6.7. Только RENAME.

Упражнение 6.8. Декартово произведение одной таблицы t по определению совпадает с t. Однако вопрос о том, что такое произведение *t1* и *t2* в случае, когда *t1* и *t2* содержат строки-дубликаты, не прост! Дополнительное обсуждение см. в ответе к упражнению 4.4.

Упражнение 6.9. Как обычно, **Tutorial D** слева, SQL справа:[1]

a.
```
SP
```
```
SELECT * FROM SP
or
TABLE SP /* см. главу 12 */
```

b.
```
( SP WHERE PNO = 'P1' )
                    { SNO }
```
```
SELECT SNO
FROM   SP
WHERE  PNO = 'P1'
```

c.
```
S WHERE STATUS ≥ 15
      AND STATUS ≤ 25
```
```
SELECT *
FROM   S
WHERE  S.STATUS BETWEEN
       15 AND 25
```

d.
```
( ( S JOIN SP )
  WHERE CITY = 'London' )
                    { PNO }
```
```
SELECT DISTINCT PNO
FROM   SP, S
WHERE  SP.SNO = S.SNO
AND    S.CITY = 'London'
```

e.
```
P { PNO } MINUS
( ( S JOIN SP )
  WHERE CITY = 'London' )
                    { PNO }
```
```
SELECT PNO
FROM   P
EXCEPT CORRESPONDING
SELECT PNO
FROM   SP , S
WHERE  SP.SNO = S.SNO
AND    S.CITY = 'London'
```

f.
```
WITH SP { SNO , PNO } AS Z :
( ( Z RENAME ( PNO AS X ) )
  JOIN
  ( Z RENAME ( PNO AS Y ) ) )
{ X , Y }
```
```
SELECT DISTINCT XX.PNO AS X ,
                YY.PNO AS Y
FROM   SP AS XX , SP AS YY
WHERE  XX.SNO = YY.SNO
```

g.
```
( S WHERE STATUS <
  STATUS FROM ( TUPLE FROM ( S
  WHERE SNO = 'S1' ) ) ) { SNO }
```
```
SELECT SNO
FROM   S
WHERE  STATUS <
       ( SELECT STATUS
         FROM   S
         WHERE  SNO = 'S1' )
```

Напомню (см. главу 3), что в **Tutorial D** выражение STATUS FROM (TUPLE FROM *r*) извлекает значение STATUS из единственного кортежа отношения *r* (это отношение *r* должно иметь кардинальность 1). А в SQL-версии этого запроса производится двойное приведение типа: сначала таблица из одной строки приводится

[1] Вообще говоря, эти решения не единственные. В частности, решение на **Tutorial D** часто можно улучшить с помощью операторов, описанных в главе 7.

к типу этой строки, а затем строка приводится к типу того единственного скалярного значения, которое содержит.

h.
```
WITH ( S WHERE CITY =                  |  SELECT DISTINCT SPX.PNO
           'London' ) AS RX ,          |  FROM   SP AS SPX
       ( SP RENAME ( PNO AS Y ) )      |  WHERE  NOT EXISTS (
                     AS RY :           |  SELECT S.SNO FROM S
  ( P WHERE RX { SNO } ⊆               |  WHERE     S.CITY = 'London'
      ( RY WHERE Y = PNO )             |  AND    NOT EXISTS (
                 { SNO } ) { PNO } )   |  SELECT SPY.*
                                       |  FROM   SP AS SPY
                                       |  WHERE  SPY.SNO = S.SNO
                                       |  AND    SPY.PNO =
                                       |         SPX.PNO ) )
```

Обратите внимание на реляционное сравнение в выражении **Tutorial D.** В SQL-версии используется оператор EXISTS (см. главу 10).

i.
```
( S { SNO } JOIN P { PNO } )  |  SELECT SNO , PNO
  MINUS SP { SNO , PNO }      |  FROM   S, P
                              |  EXCEPT CORRESPONDING
                              |  SELECT SNO , PNO
                              |  FROM   SP
```

j.
```
WITH ( SP WHERE SNO = 'S2' )       |  SELECT S.SNO FROM S
                     AS RA ,       |  WHERE  NOT EXISTS (
       ( SP RENAME ( SNO AS X ) )  |  SELECT P.* FROM P
                     AS RB :       |  WHERE  NOT EXISTS (
  S WHERE ( RB WHERE X = SNO )     |  SELECT SP.* FROM SP
          { PNO } ⊇ RA { PNO }      |  WHERE  SP.PNO = P.PNO
                                   |  AND    SP.SNO = 'S2' )
                                   |  OR     EXISTS (
                                   |  SELECT SP.*
                                   |  FROM   SP
                                   |  WHERE  SP.PNO = P.PNO
                                   |  AND    SP.SNO = S.SNO ) )
```

Упражнение 6.10. Интуитивно очевидно, что все три утверждения верны. *Более подробного ответа не будет.*

Упражнение 6.11. Объединение не является идемпотентной операцией в SQL, потому что выражение SELECT *R*.* FROM *R* UNION CORRESPONDING SELECT *R*.* FROM *R* не является тождественно равным SELECT *R*.* FROM *R*. Дело в том, что если *R* содержит дубликаты, то из объединения они удаляются. (А что, если *R* содержит null-значения? Хороший вопрос!)

Соединение идемпотентно, а потому идемпотентны также пересечение и декартово произведение – во всех случаях. Но только в реляционной модели, а не в SQL, опять же благодаря дубликатам и null-значениям.

Упражнение 6.12. Выражение *r{}* обозначает проекцию *r* на пустое множество атрибутов; оно возвращает TABLE_DUM, если *r* пусто и TABLE_DEE в противном случае. Ответ на вопрос «Каков соответствующий предикат?» зависит от предиката *r*. Например, предикат SP{} (не вполне формально) звучит так: *существует поставщик SNO, существует деталь PNO и существует количество QTY такие, что поставщик с номером SNO поставляет деталь PNO в количестве QTY*. Отметим, что этот предикат, по существу, является высказыванием; если SP пусто (и в этом случае SP{} совпадает с TABLE_DUM), то он принимает значение FALSE, в противном случае (и тогда SP{} совпадает с TABLE_DEE) он принимает значение TRUE.

Выражение *r*{ALL BUT} обозначает проекцию *r* на все свои атрибуты (другими словами, тождественную проекцию *r*); оно возвращает *r*.

Упражнение 6.13. Полагаю, что DB2 и Ingres выполняют такую оптимизацию. Возможно, другие продукты тоже.

Упражнение 6.14. Это выражение означает «Получить поставщиков, которые поставляют все фиолетовые детали». Конечно, смысл в том, что фиолетовых деталей нет вовсе (в наших тестовых данных). Выражение правильно возвращает всех пятерых поставщиков. Дальнейшее объяснение (в частности, того, почему этот ответ правилен) см. в главе 11.

Упражнение 6.15. Приближенный эквивалент на SQL мог бы выглядеть так:

```
SELECT CITY
FROM ( SELECT CITY FROM S
       UNION  CORRESPONDING
       SELECT CITY FROM P ) AS TEMP
WHERE  NOT EXISTS
     ( SELECT    CITY FROM S
       INTERSECT CORRESPONDING
       SELECT    CITY FROM P )
```

Однако это SQL-выражение не является точным эквивалентом реляционной версии. Точнее, если множества городов поставщиков и городов деталей пересекаются, то выражение будет вычислено успешно, но просто вернет пустой результат. *Примечание*: В этом примере спецификации CORRESPONDING можно было бы опустить без ущерба для смысла (почему?), но проще всегда задавать CORRESPONDING; даже если это логически излишне, – вреда не будет.

Упражнение 6.16. Два отношения *r1* и *r2* можно соединить тогда и только тогда, когда они соединяемые, то есть в случае, когда одноименные атрибуты имеют одинаковые типы (эквивалентно, тогда и только тогда, когда теоретико-множественное объединение их заголов-

ков является корректным заголовком). Это двуместный случай. Обобщить на *n*-местный случай тоже несложно: отношения *r1*, *r2*, ..., *rn* ($n > 0$) называются соединяемыми тогда и только тогда, когда для любых *i*, *j* ($1 \le i \le n$, $1 \le j \le n$) отношения *ri* и *rj* соединяемые.

Упражнение 6.17. Определение *n*-местных вариантов возможно благодаря коммутативности и ассоциативности операций JOIN, UNION и D_UNION.

SQL поддерживает *n*-местные варианты соединения и объединения, правда, не для случая $n < 2$. Синтаксис выглядит так:

```
t1 NATURAL JOIN t2
   NATURAL JOIN t3
    ..........
   NATURAL JOIN tn

SELECT * FROM t1 UNION CORRESPONDING SELECT * FROM t2
                 UNION CORRESPONDING SELECT * FROM t3
                   ..............................
                 UNION CORRESPONDING SELECT * FROM tn
```

n-местный вариант MINUS не имеет смысла, так как операция MINUS не является ни коммутативной, ни ассоциативной.

Упражнение 6.18. Краткое обоснование см. в ответе к упражнению 6.12. Развернутое таково. Рассмотрим проекцию переменной-отношения «поставщики» S на SNO – S{SNO}. Назовем результат этой проекции *r*; для наших тестовых данных *r* состоит из пяти кортежей. Теперь рассмотрим проекцию отношения *r* на пустое множество атрибутов – *r*{}. Ясно, что проецирование любого *кортежа* «ни на что» дает пустое множество; следовательно, любой кортеж *r* порождает пустой кортеж при проецировании *r* на пустое множество атрибутов. Но все пустые кортежи – дубликаты друг друга, поэтому проецирование отношения *r*, содержащего 5 кортежей, на пустое множество атрибутов дает отношение без атрибутов с одним (пустым) кортежем, иначе говоря, TABLE_DEE.

Теперь вспомним, что с любой переменной-отношением ассоциирован предикат. Для переменной-отношения S этот предикат звучит так:

> *Поставщик SNO связан контрактом, называется SNAME, имеет статус STATUS и находится в городе CITY.*

Для проекции *r* = S{SNO} предикат такой:

> *Существует некоторое название SNAME и некоторый статус STATUS и некоторый город CITY такие, что поставщик SNO связан контрактом, называется SNAME, имеет статус STATUS и находится в городе CITY.*

А для проекции *r*{} – такой:

Существует некоторый номер поставщика SNO, некоторое название SNAME и некоторый статус STATUS и некоторый город CITY такие, что поставщик SNO связан контрактом, называется SNAME, имеет статус STATUS и находится в городе CITY.

Обратите внимание, что последний предикат фактически является высказыванием: он безусловно принимает значение TRUE или FALSE. В рассматриваемом случае *r*{} совпадает с TABLE_DEE, и предикат (высказывание) обращается в TRUE. Но предположим, что в некоторый момент времени в базе данных вообще нет поставщиков. Тогда S{SNO} даст пустое отношение *r*, *r*{} будет равно TABLE_DUM, и предикат (высказывание) обратится в FALSE.

Глава 7

Упражнение 7.1.

a. SQL-аналог:

```
SELECT *
FROM   S
WHERE  SNO IN
     ( SELECT SNO
       FROM   SP
       WHERE  PNO = 'P2' )
```

Примечание

Здесь и далее приводимые мной SQL-выражения не обязательно являются прямыми транслитерациями соответствующих алгебраических аналогов, а чаще «более естественными» формулировками запроса в терминах SQL.

Предикат: *поставщик SNO связан контрактом, называется SNAME, имеет статус STATUS, находится в городе CITY и поставляет деталь P2.*

SNO	SNAME	STATUS	CITY
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London

b. SQL-аналог:

```
SELECT *
FROM   S
WHERE  SNO NOT IN
     ( SELECT SNO
       FROM   SP
       WHERE  PNO = 'P2' )
```

Предикат: *поставщик SNO связан контрактом, называется SNAME, имеет статус STATUS, находится в городе CITY и не поставляет деталь P2.*

SNO	SNAME	STATUS	CITY
S5	Adams	30	Athens

c. SQL-аналог:

```
SELECT *
FROM   P AS PX
WHERE  NOT EXISTS
     ( SELECT *
       FROM   S AS SX
       WHERE  NOT EXISTS
            ( SELECT *
              FROM   SP AS SPX
              WHERE  SPX.SNO = SX.SNO
              AND    SPX.PNO = PX.PNO ) )
```

Предикат: *деталь PNO используется на предприятии, называется PNAME, имеет цвет COLOR и вес WEIGHT, хранится на складе в городе CITY и поставляется всеми поставщиками.*

PNO	PNAME	COLOR	WEIGHT	CITY

d. SQL-аналог:

```
SELECT *
FROM   P
WHERE  ( SELECT COALESCE ( SUM ( QTY ) , 0 )
         FROM   SP
         WHERE  SP.PNO = P.PNO ) < 500
```

Предикат: *деталь PNO используется на предприятии, называется PNAME, имеет цвет COLOR и вес WEIGHT, хранится на складе в городе CITY и ее суммарное количество, поставляемое всеми поставщиками, меньше 500.*

PNO	PNAME	COLOR	WEIGHT	CITY
P3	Screw	Blue	17.0	Oslo
P6	Cog	Red	19.0	London

e. SQL-аналог:

```
SELECT *
FROM   P
WHERE  CITY IN
     ( SELECT CITY
       FROM   S )
```

Предикат: деталь PNO используется на предприятии, называется PNAME, имеет цвет COLOR и вес WEIGHT, хранится на складе в городе CITY и находится в том же городе, что некоторый поставщик.

PNO	PNAME	COLOR	WEIGHT	CITY
P1	Nut	Red	12.0	London
P2	Bolt	Green	17.0	Paris
P4	Screw	Red	14.0	London
P5	Cam	Blue	12.0	Paris
P6	Cog	Red	19.0	London

f. SQL-аналог:

```
SELECT S.* , 'Supplier' AS TAG
FROM   S
```

Предикат: *поставщик SNO связан контрактом, называется SNAME, имеет статус STATUS, находится в городе CITY, поставляет деталь P2 и имеет метку TAG «Supplier».*

SNO	SNAME	STATUS	CITY	TAG
S1	Smith	20	London	Supplier
S2	Jones	10	Paris	Supplier
S3	Blake	30	Paris	Supplier
S4	Clark	20	London	Supplier
S5	Adams	30	Athens	Supplier

g. SQL-аналог:

```
SELECT SNO , 3 * STATUS AS TRIPLE_STATUS
FROM   S
```

Предикат: *поставщик SNO связан контрактом, называется SNAME, имеет статус STATUS, находится в городе CITY, поставляет деталь P2 и имеет атрибут TRIPLE_STATUS, значение которого в три раза больше значения STATUS.*

SNO	SNAME	STATUS	CITY	TRIPLE_STATUS
S1	Smith	20	London	60
S2	Jones	10	Paris	30
S3	Blake	30	Paris	90
S4	Clark	20	London	60
S5	Adams	30	Athens	90

h. SQL-аналог:

```
SELECT PNO , PNAME, COLOR , WEIGHT , CITY , SNO , QTY
       WEIGHT * QTY AS SHIPWT
FROM   P NATURAL JOIN SP
```

Предикат: *деталь PNO используется на предприятии, называется PNAME, имеет цвет COLOR и вес WEIGHT, хранится на складе в городе CITY, поставляется поставщиком SNO в количе-*

стве QTY, общий вес этой поставки (детали PNO поставщиком SNO) SHIPWT равен произведению WEIGHT на QTY.

SNO	PNO	QTY	PNAME	COLOR	WEIGHT	CITY	SHIPWT
S1	P1	300	Nut	Red	12.0	London	3600.0
S1	P2	200	Bolt	Green	17.0	Paris	3400.0
S1	P3	400	Screw	Blue	17.0	Oslo	6800.0
S1	P4	200	Screw	Red	14.0	London	2800.0
S1	P5	100	Cam	Blue	12.0	Paris	1200.0
S1	P6	100	Cog	Red	19.0	London	1900.0
S2	P1	300	Nut	Red	12.0	London	3600.0
S2	P2	400	Bolt	Green	17.0	Paris	6800.0
S3	P2	200	Bolt	Green	17.0	Paris	3400.0
S4	P2	200	Bolt	Green	17.0	Paris	3400.0
S4	P4	300	Screw	Red	14.0	London	4200.0
S4	P5	400	Cam	Blue	12.0	Paris	4800.0

i. SQL-аналог:

```
SELECT P.* , WEIGHT * 454 AS GMWT , WEIGHT * 16 AS OZWT
FROM   P
```

Предикат: *деталь PNO используется на предприятии, называется PNAME, имеет цвет COLOR, вес WEIGHT, вес в граммах GMWT (= 454, умноженное на WEIGHT) и вес в унциях OZWT (= 16, умноженное на WEIGHT).*

PNO	PNAME	COLOR	WEIGHT	CITY	GMWT	OZWT
P1	Nut	Red	12.0	London	5448.0	192.0
P2	Bolt	Green	17.0	Paris	7718.0	204.0
P3	Screw	Blue	17.0	Oslo	7718.0	204.0
P4	Screw	Red	14.0	London	6356.0	168.0
P5	Cam	Blue	12.0	Paris	5448.0	192.0
P6	Cog	Red	19.0	London	8626.0	228.0

j. SQL-аналог:

```
SELECT P.* , ( SELECT COUNT ( SNO )
               FROM   SP
               WHERE  SP.PNO = P.PNO ) AS SCT
FROM   P
```

Предикат: *деталь PNO используется на предприятии, называется PNAME, имеет цвет COLOR, вес WEIGHT, находится на складе в городе CITY и поставляется SCT поставщиками.*

PNO	PNAME	COLOR	WEIGHT	CITY	SCT
P1	Nut	Red	12.0	London	2
P2	Bolt	Green	17.0	Paris	4
P3	Screw	Blue	17.0	Oslo	1
P4	Screw	Red	14.0	London	2
P5	Cam	Blue	12.0	Paris	2
P6	Cog	Red	19.0	London	1

k. SQL-аналог:

```
SELECT S.* , ( SELECT COUNT ( PNO )
               FROM   SP
               WHERE  SP.SNO = S.SNO ) AS NP
FROM   S
```

Предикат: поставщик SNO связан контрактом, называется SNAME, имеет статус STATUS, находится в городе CITY и поставляет деталь NP деталей.

SNO	SNAME	STATUS	CITY	NP
S1	Smith	20	London	6
S2	Jones	10	Paris	2
S3	Blake	30	Paris	1
S4	Clark	20	London	3
S5	Adams	30	Athens	0

l. SQL-аналог:

```
SELECT CITY , AVG ( STATUS ) AS AVG_STATUS
FROM   S
GROUP  BY CITY
```

Предикат: *средний статус поставщиков в городе CITY равен AVG_STATUS.*

CITY	AVG_STATUS
London	20
Paris	20
Athens	30

m. SQL-аналог:

```
SELECT COUNT ( SNO ) AS N
FROM   S
WHERE  CITY = 'London'
```

Предикат: *в Лондоне находится N поставщиков.*

N
2

Отсутствие здесь двойного подчеркивания – *не* ошибка.

n. SQL-аналог:

```
( SELECT *
  FROM   SP
  EXCEPT CORRESPONDING
  SELECT *
  FROM   SP
  WHERE  SNO = 'S1' )
UNION  CORRESPONDING
```

```
SELECT 'S7' AS SNO , PNO , QTY * 0.5 AS QTY
FROM   SP
WHERE  SNO = 'S1'
```

Предикат: *либо (а) поставщик SNO поставляет деталь PNO в количестве QTY (и SNO не равно S1), либо (б) SNO равно S7 и поставщик S1 поставляет деталь PNO в количестве, в два раза превышающем QTY. Кстати: что произойдет, если SP уже включает какие-нибудь кортежи для поставщика S7?*

SNO	PNO	QTY
S7	P1	150
S7	P2	100
S7	P3	200
S7	P4	100
S7	P5	50
S7	P6	50
S2	P1	300
S2	P2	400
S3	P2	200
S4	P2	200
S4	P4	300
S4	P5	400

Упражнение 7.2. Выражения *r1* MATCHING *r2* и *r2* MATCHING *r1* эквивалентны тогда и только тогда, когда *r1* и *r2* имеют один и тот же тип, при этом оба выражения сводятся просто к JOIN{*r1*,*r2*}, а оно, в свою очередь, сводится к INTERSECT{*r1*,*r2*}.

Упражнение 7.3. Оператор переименования не является примитивным, например потому, что выражения

```
S RENAME ( CITY AS SCITY )
```

и

```
( EXTEND S ADD ( CITY AS SCITY ) ) { ALL BUT CITY }
```

эквивалентны.

Упражнение 7.4. `EXTEND S { SNO } ADD ( COUNT ( !! SP ) AS NP )`

Упражнение 7.5. Определить, какие выражения эквивалентны, можно, посмотрев на результаты их вычисления. Отмечу, что SUM(1), вычисленное по *n* кортежам, равно *n*.

a.

r пусто: | CT | | *r* состоит из *n* кортежей (*n* > 0): | CT | | n |

b.

r пусто: | CT | | 0 | *r* состоит из *n* кортежей (*n* > 0): | CT | | n |

c.

r пусто:

CT

r состоит из *n* кортежей (*n* > 0):

CT
n

d.

r пусто:

CT
0

r состоит из *n* кортежей (*n* > 0):

CT
n

Иными словами, результатом в любом случае является отношение степени 1. Если *r* не пусто, то все четыре выражения эквивалентны, в противном случае (a) эквивалентно (c), а (b) эквивалентно (d). SQL-аналоги:

a.
```
SELECT COUNT ( * ) AS CT
FROM   r
EXCEPT CORRESPONDING
SELECT 0 AS CT
FROM   r
```

b.
```
SELECT COUNT ( * ) AS CT
FROM   r
```

Примечание

При желании можно было бы без ущерба для смысла добавить к этому выражению INTERSECT CORRESPONDING SELECT 0 AS CT FROM *r*.

---

c. То же, что (a).

d. То же, что (b).

Упражнение 7.6. SQL возвращает null во всех случаях, кроме COUNT, где правильно возвращается 0. О причинах можно только догадываться.

Упражнение 7.7. Существует достаточно прямолинейная формулировка: *поставщик SNO поставляет деталь PNO тогда и только тогда, когда деталь PNO встречается в отношении PNO_REL*. Кстати говоря, фраза «тогда и только тогда» важна (верно?).

Упражнение 7.8. Отношение *r* имеет такую же кардинальность, как SP, и такой же заголовок, если не считать одного дополнительного RVA-атрибута X. Отношения, являющиеся значениями X, имеют степень 0, причем каждое представляет собой TABLE_DEE, а не TABLE_DUM, потому что каждый кортеж *sp* в SP включает 0-кортеж в качестве значения того подкортежа *sp*, который соответствует пустому множеству атрибутов. Таким образом, каждый кортеж отношения *r* по сути дела состоит из соответствующего кортежа SP,

расширенного значением TABLE_DEE атрибута X, и исходное выражение GROUP логически эквивалентно следующему:

```
EXTEND SP ADD ( TABLE_DEE AS X )
```

Выражение *r* UNGROUP (X) дает исходное отношение SP.

Упражнение 7.9.

a.
```
N := COUNT ( SP
     WHERE SNO = 'S1' ) ;
```
```
SET N = ( SELECT COUNT ( * )
          FROM   S
          WHERE  SNO = 'S1' ) ;
```

b.
```
( S WHERE CITY =
  MIN ( S , CITY ) ) { SNO }
```
```
SELECT *
FROM   S
WHERE  CITY =
       ( SELECT MIN ( CITY )
         FROM   S )
```

c.
```
S { CITY }
WHERE COUNT ( !!S ) > 1
```
```
SELECT DISTINCT CITY
FROM   S AS SX
WHERE  ( SELECT COUNT ( * )
         FROM   S AS SY
         WHERE  SY.CITY = SX.CITY )
       > 1
```

d.
```
RESULT :=
IF AND ( S , SNAME < CITY )
THEN 'Y' ELSE 'N' END IF ;
```
```
SET RESULT = CASE WHEN
     ( SELECT COALESCE ( EVERY
     ( SNAME < CITY ) , TRUE ) )
       FROM   S ) THEN 'Y'
                  ELSE 'N'
              END ;
```

Примечание

EVERY – это SQL-аналог агрегатного оператора AND в **Tutorial D**, но он возвращает null, а не TRUE, если аргумент пуст.

Упражнение 7.10. Номера поставщиков, поставляющих деталь P2, и номера деталей, поставляемых поставщиком S2, соответственно. Обратите внимание, что формулировки на естественном языке симметричны, тогда как формальные – нет; это объясняется асимметричностью самого отношения R4 в том смысле, что номера поставщиков и номера деталей рассматриваются в нем совершенно по-разному. *Примечание*: Именно потому, что отношения с RVA-атрибутами (обычно) несимметричны, они и противопоказаны (обычно).

Упражнение 7.11. Оно обозначает отношение, которое выглядит примерно так, как показано ниже.

Здесь PNO_REL – RVA-атрибут. Отметим, в частности, что пустое множество деталей, поставляемых поставщиком S5, представлено пустым множеством, а не (как то имело бы место, если бы мы образовали внешнее соединение S и SP) null-значением. Представление пустого множества пустым множеством, очевидно, выглядит здравой

идеей; на самом деле, во внешнем соединении вообще не было бы нужды, если бы RVA-атрибуты были надлежащим образом поддержаны!

SNO	SNAME	STATUS	CITY	PNO_REL
S1	Smith	20	London	PNO P1 P2 ... P6
S2	Jones	10	Paris	PNO P1 P2
...	.....	..	.....	.....
S5	Adams	30	Athens	PNO

Попутно отметим, что если обозначить показанное выше отношение *r*, то выражение

```
( r UNGROUP ( PNO_REL ) ) { ALL BUT PNO }
```

не вернет наше привычное отношение «поставщики». Точнее, оно вернет отношение, идентичное отношению «поставщики» с тем отличием, что в нем не будет кортежа для поставщика S5.

Упражнение 7.12. С первым все ясно: оно вставляет новый кортеж, в котором SNO – S6, SNAME – Lopez, STATUS – 30, CITY – Madrid, а значением PNO_REL является отношение, содержащее единственный кортеж, в котором есть только один атрибут PNO со значением P5. Что касается второго, то думаю, было бы полезно показать соответствующую часть (хотя бы упрощенную) грамматики **Tutorial D**, описывающую реляционное присваивание (предполагается, что названия символов грамматики не требуют пояснения):

```
<relation assign>
    ::=    <relvar ref> := <relation exp>
         | <relation insert>
         | <relation delete>
         | <relation update>

<relation insert>
     ::=   INSERT <relvar ref> <relation exp>

<relation delete>
     ::=   DELETE <relvar ref> [ WHERE <boolean exp> ]

<relation update>
     ::=   UPDATE <relvar ref> [ WHERE <boolean exp> ] :
```

```
{ <attribute assign commalist> }
```

А символ *<attribute assign>* в случае, когда значениями рассматриваемого атрибута являются отношения, – это, по существу, просто *<relation assign>*, что нам и требуется. Таким образом, второе обновление в данном упражнении заменяет кортеж для поставщика S2 другим, в котором значение PNO_REL дополнительно включает кортеж для поставщика S5.

Отметим поэтому, что оба обновления суть формальные представления следующих фраз на естественном языке:

a. Добавить в базу данных тот факт, что поставщик S6 поставляет деталь P5.

b. Добавить в базу данных тот факт, что поставщик S2 поставляет деталь P5.

В случае принятой нами структуры базы о поставщиках и деталях (без RVA-атрибутов) между двумя обновлениями нет качественной разницы – оба сводятся к вставке одного кортежа в переменную-отношение SP следующим образом (для простоты игнорируем атрибут QTY):

a. `INSERT SP RELATION { TUPLE { SNO 'S6' , PNO 'P5' } } ;`

b. `INSERT SP RELATION { TUPLE { SNO 'S2' , PNO 'P5' } } ;`

Однако в случае SSP эти симметричные обновления обрабатываются несимметрично. Как было отмечено в ответе к упражнению 7.10, именно такое отсутствие симметрии и составляет (обычно, но не всегда) проблему при работе с переменными-отношениями, включающими RVA-атрибуты.

Упражнение 7.13. Запрос (a) простой:

```
WITH ( SSP RENAME ( SNO AS XNO ) ) { XNO, PNO_REL } AS X ,
     ( SSP RENAME ( SNO AS YNO ) ) { YNO, PNO_REL } AS Y :
( X JOIN Y ) { XNO, YNO }
```

Отметим, что здесь соединение производится по RVA-атрибуту (поэтому неявно выполняются реляционные сравнения).

Однако запрос (b) не столь очевиден. С запросом (a) мы так легко справились, потому что в SSP «детали вложены в поставщиков», а для запроса (b) хотелось бы, чтобы поставщики были вложены в детали. Так сделаем это:[1]

[1] Этот пример подчеркивает важное различие между RVA-атрибутами в реляционной системе и иерархиями в системах типа IMS (или XML?). В IMS иерархии «зашиты» в базу данных, иными словами, нам ничего не остается, как работать с иерархиями, которые придумал проектировщик базы данных. Напротив, в реляционной системе мы можем динамически строить любые удобные нам иерархии с помощью подходящих операторов реляционной алгебры.

```
WITH ( SSP UNGROUP ( PNO_REL ) ) GROUP ( { SNO } AS SNO_REL )
                                                AS PPS ,
     ( PPS RENAME ( PNO AS XNO ) ) { XNO, SNO_REL } AS X ,
     ( PPS RENAME ( PNO AS YNO ) ) { YNO, SNO_REL } AS Y :
( X JOIN Y ) { XNO, YNO }
```

Упражнение 7.14.

```
WITH ( P RENAME ( WEIGHT AS WT ) ) AS R1 ,
     ( EXTEND P ADD ( COUNT ( R1 WHERE WT > WEIGHT )
                                      AS N_HEAVIER ) AS R2 :
( R2 WHERE N_HEAVIER < 2 ) { ALL BUT N_HEAVIER }
```

```
SELECT *
FROM   P AS PX
WHERE ( SELECT COUNT ( * )
        FROM   P AS PY
        WHERE  PX.WEIGHT > PY.WEIGHT ) < 2
```

Этот запрос возвращает кортежи для деталей P2, P3 и P6 (то есть отношение кардинальности 3, хотя была задана квота 2). Квотированные запросы могут возвращать и меньше кортежей, чем заказано; рассмотрим, к примеру, запрос «Получить десять самых тяжелых деталей».

Примечание

Квотированные запросы часто встречаются на практике. Поэтому в книге «Databases, Types, and the Relational Model: The Third Manifesto» (см. приложение D) мы с Хью Дарвеном предлагаем для них «дружественную» сокращенную нотацию, в которой показанный выше запрос выглядел бы так:

```
( ( RANK P BY ( DESC WEIGHT AS W ) ) WHERE W J 2 ) { ALL BUT W }
```

В SQL есть нечто похожее.

Упражнение 7.15. Этот запрос демонстрирует одно слабое место оператора SUMMARIZE. Вот требуемая нам формулировка:

```
SUMMARIZE SP PER ( S { SNO } ) ADD ( SUMD ( QTY ) AS SDQ )
```

Вся хитрость в операторе «SUMD» – здесь буква «D» происходит от *distinct* и означает «устранить избыточные дубликаты перед тем, как вычислять сумму». Но это не более чем трюк (ситуативный, на мой взгляд), поэтому (наряду с многими другими причинами) лучше прибегнуть к следующему выражению, в котором используются оператор EXTEND и отношения-образы:

```
EXTEND S { SNO } ADD ( SUM ( !!SP , QTY ) AS SDQ )
```

Упражнение 7.16.

```
EXTEND S ADD ( COUNT ( !!SP ) AS NP , COUNT ( !!SJ ) AS NJ )
```

```
( SUMMARIZE SP PER ( S { SNO } ) ADD ( COUNT ( PNO ) AS NP ) )
  JOIN
( SUMMARIZE SJ PER ( S { SNO } ) ADD ( COUNT ( JNO ) AS NJ ) )

SELECT SNO , ( SELECT COUNT ( PNO )
               FROM   SP
               WHERE  SP.SNO = S.SNO ) AS NP ,
             ( SELECT COUNT ( JNO )
               FROM   SJ
               WHERE  SJ.SNO = S.SNO ) AS NJ
FROM   S
```

Упражнение 7.17. Легко видеть, что операция «!!» идемпотентна, следовательно, !!(!!SP) совпадает с !!SP, и выражение семантически эквивалентно такому:

```
S WHERE (!!SP) { PNO } = P { PNO }
```

(«Получить поставщиков, которые поставляют все детали»).

Упражнение 7.18. Нет никакого логического различия.

Упражнение 7.19. S JOIN SP не является полусоединением; S MATCHING SP не является соединением (это проекция соединения). Выражения *r1* JOIN *r2* и *r1* MATCHING *r2* эквивалентны тогда и только тогда, когда отношения *r1* и *r2* одного типа (тогда конечная проекция становится тождественной и все выражение вырождается в *r1* INTERSECTION *r2*).

Упражнение 7.20. Если *r1* и *r2* одного типа и *t1* – кортеж *r1*, то выражение !!*r2* (для *t1*) обозначает отношение степени 0: TABLE_DEE, если *t1* встречается в *r2*, и TABLE_DUM в противном случае. А если *r1* и *r2* – одно и то же отношение, то !!*r2* обозначает TABLE_DEE для любого кортежа *r1*.

Упражнение 7.21. Они одинаковы, если таблица S не пуста, а в этом случае первое дает таблицу из одной строки и одного столбца, содержащую нуль, а второе – таблицу из одной строки и одного столбца, «содержащую null».

Глава 8

Упражнение 8.1. Ограничение типа – это определение множества значений, составляющих данный тип. Ограничение типа для типа *T* проверяется в момент вызова селектора типа *T*; если проверка не проходит, то вызов селектора завершается ошибкой нарушения ограничения типа.

Ограничение базы данных – это ограничение на значения, которые могут встречаться в этой базе. Ограничения базы данных проверяются в «месте появления точки с запятой», точнее, в конце любого предложения, которое присваивает значение любой из относящихся

к делу переменных-отношений. Если проверка не проходит, то присваивание завершается ошибкой нарушения ограничения базы данных. *Примечание*: ограничения базы данных должны проверяться также в момент определения. Если проверка не проходит, то определение ограничения должно быть отвергнуто.

Упражнение 8.2. Золотое правило утверждает, что никакая операция обновления не должна приводить к обращению в FALSE ни одного ограничения базы данных, и тем более операция обновления не должна приводить к обращению в FALSE никакого ограничения переменной-отношения. Однако полное ограничение переменной-отношения может обращаться в FALSE – не потому, что нарушено какое-то ограничение с одной переменной-отношением, а из-за нарушения ограничения с несколькими переменными-отношениями. Впрочем, этот момент носит академический характер, учитывая, что разделение на ограничения с одной и с несколькими переменными-отношениями все равно произвольно.

Упражнение 8.3. «Утверждением» в SQL называется ограничение, созданное с помощью предложения CREATE ASSERTION. Под ограничением атрибута понимается тот факт, что атрибут имеет определенный тип. Ограничением базовой таблицы в SQL называется ограничение, которое задается в составе определения базовой таблицы (а не в составе определения столбца внутри определения базовой таблицы). Ограничением столбца в SQL называется ограничение, которое задается в составе определения столбца. Ограничение с несколькими переменными-отношениями – это ограничение базы данных, в котором упоминаются две или более различных переменных-отношения. Ссылочное ограничение сводится к тому, что если *B* ссылается на *A*, то *A* должно существовать. Ограничением переменной-отношения для переменной-отношения *R* называется ограничение базы данных, в котором упоминается *R*. Ограничением строки в SQL называется ограничение, обладающее тем свойством, что его можно проверить для любой строки, исследуя только эту строку. Ограничение с одной переменной-отношением – это ограничение базы данных, в котором упоминается только одна переменная-отношение. Ограничение состояния – это ограничение базы данных, не являющееся ограничением перехода. Полное ограничение базы данных *DB* – это логическое AND всех полных ограничений переменных-отношение в базе данных *DB* и значения TRUE. Полное ограничение переменной-отношения *R* – это логическое AND всех ограничений базы данных, в которых упоминается *R*, и значения TRUE. Ограничением перехода называется ограничение на допустимые переходы из одного «состояния» (то есть значения) базы данных в другое. Ограничением кортежа называется ограничение переменной-отношения, обладающее тем свойством, что его можно проверить для данного кортежа, исследуя только сам этот кортеж. В какие из этих категорий попада-

ют (а) ограничения ключа, (б) ограничения внешнего ключа? *Вопрос оставлен без ответа.*

Упражнение 8.4. См. текст главы.

Упражнение 8.5. a. 345. **b.** QTY.

Упражнение 8.6. См. текст главы.

Упражнение 8.7.

```
TYPE CITY POSSREP { C CHAR CONSTRAINT C = 'London'
                                      OR C = 'Paris'
                                      OR C = 'Rome'
                                      OR C = 'Athens'
                                      OR C = 'Oslo'
                                      OR C = 'Stockholm'
                                      OR C = 'Madrid'
                                      OR C = 'Amsterdam' } ;
```

Теперь для атрибута CITY в переменных-отношениях S и P можно задать тип CITY, а не просто CHAR.

Упражнение 8.8. По определению, не существует способа наложить ограничение, эквивалентное приведенному в ответе на предыдущее упражнение, не определяя явно тип. Но можно потребовать, чтобы города поставщиков могли принимать только перечисленные выше восемь значений, воспользовавшись подходящим ограничением базы данных. И точно так же ограничить города деталей. Например, можно было бы определить базовую таблицу следующим образом:

```
CREATE TABLE C ( CITY CHAR , UNIQUE ( CITY ) ) ;
```

Затем заполнить эту таблицу, поместив в нее восемь значений городов:

```
INSERT INTO C VALUES 'London'    ,
                     'Paris'     ,
                     'Rome'      ,
                     'Athens'    ,
                     'Oslo'      ,
                     'Stockholm' ,
                     'Madrid'    ,
                     'Amsterdam' ;
```

После чего определить внешние ключи:

```
CREATE TABLE S ( ... ,
                 FOREIGN KEY ( CITY ) REFERENCES C ( CITY ) ) ;

CREATE TABLE P ( ... ,
                 FOREIGN KEY ( CITY ) REFERENCES C ( CITY ) ) ;
```

Преимущество такого подхода в том, что множество допустимых городов при необходимости легко можно изменить.

Другой подход – задать ограничения базовой таблицы (или столбца) в составе определений базовых таблиц S и P. Третий – воспользоваться предложениями CREATE ASSERTION. И еще один – определить триггерные процедуры.

Все эти подходы требуют определенной работы, причем первый, пожалуй, – самой неприятной.

Упражнение 8.9.

```
TYPE SNO POSSREP
{ C CHAR CONSTRAINT
         CHAR_LENGTH ( C ) ≥ 2 AND CHAR_LENGTH ( C ) ≤ 5
         AND SUBSTR ( C, 1, 1 ) = 'S'
         AND CAST_AS_INTEGER ( SUBSTR ( C, 2 ) ) ≥ 0
         AND CAST_AS_INTEGER ( SUBSTR ( C, 2 ) ) ≤ 9999 } ;
```

Я предполагаю, что операторы CHAR_LENGTH, SUBSTR и CAST_AS_INTEGER имеются и наделены очевидной семантикой.

Упражнение 8.10.

```
TYPE LINESEG POSSREP { BEGIN POINT, END POINT } ;
```

Я предполагаю, что пользовательский тип POINT определен, как показано в тексте главы.

Упражнение 8.11. Один из примеров – тип POINT, но есть и много других. Так, можете подумать о типе PARALLELOGRAM, который можно было бы представить многими разными способами (сколько вы сможете предложить?). Что касается ограничений типа: концептуально спецификация каждого допустимого представления должна включать ограничение типа, и все эти ограничения должны быть логически эквивалентны. Например:

```
TYPE POINT
     POSSREP CARTESIAN { X FIXED , Y FIXED
             CONSTRAINT SQRT ( X ** 2 + Y ** 2 ) ≤ 100.0 }
     POSSREP POLAR { R FIXED , THETA FIXED
             CONSTRAINT R ≤ 100.0 } ;
```

Вопрос о том, можно ли предложить некую сокращенную нотацию, которая позволила бы задавать ограничение только один раз, выходит за рамки настоящей книги.

Упражнение 8.12. Отрезок прямой можно, например, представить начальной и конечной точками или средней точкой, длиной и углом наклона.

Упражнение 8.13. Я дам ответы в терминах сокращений INSERT, DELETE и UPDATE, а не самих реляционных присваиваний:

CX1: INSERT в S, UPDATE атрибута STATUS в S

CX2: INSERT в S, UPDATE атрибута CITY или STATUS в S

CX3: INSERT в S, UPDATE атрибута SNO в S

CX4: INSERT в S, UPDATE атрибута SNO или CITY в S

CX5: UPDATE атрибута STATUS в S, INSERT в SP, UPDATE атрибута SNO или PNO в SP (здесь я предполагаю, что действует ограничение CX6 – ограничение внешнего ключа в направлении SP к S)

CX6: DELETE из S, UPDATE атрибута SNO в S, INSERT в SP, UPDATE атрибута SNO в SP

CX7: INSERT в LS или NLS, UPDATE атрибута SNO в LS или NLS

CX8: INSERT в S или P, UPDATE атрибута SNO или CITY в S, UPDATE атрибута PNO или CITY в P

CX9: UPDATE атрибута SNO или STATUS в S

Упражнение 8.14. Нет, хотя не видно причин, по которым его нельзя было бы при желании добавить.

Упражнение 8.15. (Ниже приводится несколько упрощенный ответ, но смысл происходящего в нем отражен.) Пусть *c* – ограничение базовой таблицы *T*; тогда эквивалентное *c* предложение CREATE ASSERTION логически должно было бы иметь вид FORALL *r* (*c*) – или, чуть ближе к реальному синтаксису SQL, NOT EXISTS *r* (NOT *c*), – где *r* обозначает строку *T*. Иными словами, логически необходимый квантор всеобщности неявно подразумевается в ограничении базовой таблицы, но должен быть явно задан в утверждении.

Упражнение 8.16. Формальная причина связана с определением квантора FORALL в случае, когда областью применимости является пустое множество; дополнительные пояснения см. в главе 10. В языке **Tutorial D** нет прямого аналога ограничениям базовых таблиц, поэтому и такое поведение в нем не наблюдается.

Упражнение 8.17.

```
CREATE TABLE S
  ( ... ,
    CONSTRAINT CX5 CHECK
      ( STATUS >= 20 OR SNO NOT IN ( SELECT SNO
                                     FROM   SP
                                     WHERE  PNO = 'P6' ) ) ;

CREATE TABLE P
  ( ... ,
    CONSTRAINT CX5 CHECK
      ( NOT EXISTS ( SELECT *
                     FROM   S NATURAL JOIN SP
                     WHERE  STATUS < 20
                     AND    PNO = 'P6' ) ) ;
```

Отметим, что во второй формулировке спецификация ограничения не содержит ссылок на базовую таблицу, в состав определения ко-

торой входит. Следовательно, точно такая же спецификация могла бы входить в состав определения абсолютно любой базовой таблицы. (По существу, она идентична версии с CREATE ASSERTION.)

Упражнение 8.18. Булево выражение в ограничении CX1 – это простое условие выборки (restriction); выражение в ограничении CX5 сложнее. Одним из следствий является тот факт, что для кортежа, вставляемого в S, ограничение CX1 можно проверить, даже не рассматривая уже имеющиеся в базе данных значения, тогда как для ограничения CX5 это не верно.

Упражнение 8.19. Да, конечно, это возможно; пример дает ограничение CX3. Однако отметим, что в общем случае ни ограничение типа CX3, ни явная спецификация KEY не могут гарантировать, что заданная комбинация атрибутов удовлетворяет требованию неприводимости. (Хотя можно было бы хотя бы ввести синтаксическое правило о том, что если для одной и той же переменной-отношения заданы два разных ключа, то ни один из них не должен быть собственным подмножеством другого. Такое правило было бы полезно, но полностью проблему все равно не решило бы.)

Упражнение 8.20.

```
CREATE ASSERTION CX8 CHECK
   ( ( SELECT COUNT ( * )
       FROM ( SELECT CITY
              FROM   S
              WHERE  SNO = 'S1'
              UNION  CORRESPONDING
              SELECT CITY
              FROM   P
              WHERE  PNO = 'P1' ) AS POINTLESS ) < 2 ) ;
```

Упражнение 8.21. Из-за длины строк слишком трудно привести формулировки на **Tutorial D** и SQL рядом, поэтому в каждом случае сначала я привожу первую, а вторую – вслед за ней. Подробное описание того, какие операции могут привести к нарушению этих ограничений, я опускаю.

a.
```
CONSTRAINT CXA IS_EMPTY
    ( P WHERE COLOR = 'Red' AND WEIGHT ≥ 50.0 ) ;
```

```
CREATE ASSERTION CXA CHECK ( NOT EXISTS (
   SELECT *
   FROM   P
   WHERE  COLOR = 'Red'
   AND    WEIGHT >= 50.0 ) ) ;
```

b.
```
CONSTRAINT CXB IS_EMPTY (
    ( S WHERE CITY = 'London' )
        WHERE TUPLE { PNO 'P2' } ∉ (!!SP) { PNO } ) ;
```

```
CREATE ASSERTION CXB CHECK (
   NOT EXISTS ( SELECT * FROM S
                WHERE  CITY = 'London'
                AND    NOT EXISTS
                     ( SELECT * FROM SP
                       WHERE  SP.SNO = S.SNO
                       AND    SP.PNO = 'P2' ) ) ) ;
```

c.
```
CONSTRAINT CXC COUNT ( S ) = COUNT ( S { CITY } ) ;

CREATE ASSERTION CXC CHECK ( UNIQUE ( SELECT CITY FROM S ) ) ;
```

d.
```
CONSTRAINT CXD COUNT ( S WHERE CITY = 'Athens' ) < 2 ;

CREATE ASSERTION CXD CHECK
     ( UNIQUE ( SELECT CITY FROM S WHERE CITY = 'Athens' ) ) ;
```

e.
```
CONSTRAINT CXE COUNT ( S WHERE CITY = 'London' ) > 0 ;

CREATE ASSERTION CXE CHECK
     ( EXISTS ( SELECT * FROM S WHERE CITY = 'London' ) ) ;
```

f.
```
CONSTRAINT CXF COUNT ( P WHERE COLOR = 'Red'
                         AND WEIGHT < 50.0 ) > 0 ;

CREATE ASSERTION CXF CHECK
     ( EXISTS ( SELECT * FROM P
                WHERE  COLOR = 'Red'
                AND    WEIGHT < 50.0 ) ) ;
```

g.
```
CONSTRAINT CXG CASE
                  WHEN IS_EMPTY ( S ) THEN TRUE
                  ELSE AVG ( S , STATUS ) > 10
               END CASE ;

CREATE ASSERTION CXG CHECK
     ( CASE
          WHEN NOT EXISTS ( SELECT * FROM S ) THEN TRUE
          ELSE ( SELECT AVG ( STATUS ) FROM S ) > 10
       END ) ;
```

h.
```
CONSTRAINT CXH
   CASE
      WHEN IS_EMPTY ( SP ) THEN TRUE
      ELSE IS_EMPTY ( SP WHERE QTY > 2 * AVG ( SP , QTY ) )
   END CASE ;
CREATE ASSERTION CXH CHECK
     ( CASE
          WHEN NOT EXISTS ( SELECT * FROM SP ) THEN TRUE
          ELSE NOT EXISTS ( SELECT * FROM SP
                            WHERE  QTY > 2 *
```

```
                                        ( SELECT AVG ( QTY )
                                          FROM   SP ) )
          END ) ;
```

i.
```
CONSTRAINT CXI CASE
       WHEN COUNT ( S ) < 2 THEN TRUE
       ELSE IS_EMPTY ( JOIN
         { ( S WHERE STATUS = MAX ( S { STATUS } ) ) { CITY } ,
           ( S WHERE STATUS = MIN ( S { STATUS } ) ) { CITY } } )
                END CASE ;

CREATE ASSERTION CXI CHECK ( CASE
   WHEN ( SELECT COUNT ( * ) FROM S ) < 2 THEN TRUE
   ELSE NOT EXISTS
      ( SELECT * FROM S AS X , S AS Y
        WHERE  X.STATUS = ( SELECT MAX ( STATUS ) FROM S )
        AND    Y.STATUS = ( SELECT MIN ( STATUS ) FROM S )
        AND    X.CITY = Y.CITY )
                             END ) ;
```

j.
```
CONSTRAINT CXJ P { CITY } ⊆ S { CITY } ;

CREATE ASSERTION CXJ CHECK ( NOT EXISTS
     ( SELECT * FROM P
       WHERE  NOT EXISTS
            ( SELECT * FROM S
              WHERE  S.CITY = P.CITY ) ) ) ;
```

k.
```
CONSTRAINT CXK IS_EMPTY (
         ( ( EXTEND P ADD ( (!!SP) JOIN S ) { CITY } AS SC )
           WHERE TUPLE { CITY CITY } ∉ SC ) ) ;

CREATE ASSERTION CXK CHECK ( NOT EXISTS
     ( SELECT * FROM P
       WHERE  NOT EXISTS
            ( SELECT * FROM S
              WHERE  S.CITY = P.CITY
              AND    EXISTS
                   ( SELECT * FROM SP
                     WHERE  S.SNO = SP.SNO
                     AND    P.PNO = SP.PNO ) ) ) ) ;
```

l.
```
CONSTRAINT CXL
     COUNT ( ( ( S WHERE CITY = 'London' ) JOIN SP ) { PNO } ) >
     COUNT ( ( ( S WHERE CITY = 'Paris'  ) JOIN SP ) { PNO } ) ;

CREATE ASSERTION CXL CHECK (
  ( SELECT COUNT ( DISTINCT PNO ) FROM S , SP
    WHERE  S.SNO = SP.SNO
    AND    S.CITY = 'London' ) >
  ( SELECT COUNT ( DISTINCT PNO ) FROM S , SP
```

```
      WHERE  S.SNO = SP.SNO
      AND    S.CITY = 'Paris' ) ) ;
```

m.
```
CONSTRAINT CXM
      SUM ( ( ( S WHERE CITY = 'London' ) JOIN SP ) , QTY ) >
      SUM ( ( ( S WHERE CITY = 'Paris'  ) JOIN SP ) , QTY ) ;
```

```
   CREATE ASSERTION CXM CHECK (
    ( SELECT COALESCE ( SUM ( QTY ) , 1 ) FROM S , SP
      WHERE  S.SNO = SP.SNO
      AND    S.CITY = 'London' ) >
    ( SELECT COALESCE ( SUM ( QTY ) , 0 ) FROM S , SP
      WHERE  S.SNO = SP.SNO
      AND    S.CITY = 'Paris' ) ) ;
```

n.
```
CONSTRAINT CXN IS_EMPTY
      ( ( SP JOIN P ) WHERE QTY * WEIGHT > 20000.0 ) ;
```

```
   CREATE ASSERTION CXN CHECK
        ( NOT EXISTS ( SELECT * FROM SP NATURAL JOIN P
                       WHERE  QTY * WEIGHT > 20000.0 ) ) ;
```

Упражнение 8.22. Оно гарантирует, что ограничение удовлетворяется для пустой базы данных (то есть не содержащей никаких переменных-отношений).

Упражнение 8.23. Предположим, что требуется определить переменную-отношение SC с атрибутами SNO и CITY и предикатом *Поставщик SNO не имеет офиса в городе CITY*. Предположим далее, что поставщик S1 имеет офисы всего в десяти городах. Тогда из *допущения замкнутости мира* следовало бы, что в переменной-отношении SC должно быть n-10 кортежей для поставщика S1, где n – общее количество возможных городов (быть может, вообще всех городов в мире)!

Упражнение 8.24. Нам понадобится множественное присваивание (если требуется выполнить удаление в одном предложении):

```
DELETE S WHERE SNO = x , DELETE SP WHERE SNO = x ;
```

Упражнение 8.25. Эти ограничения нельзя выразить декларативно (ни SQL, ни **Tutorial D** в настоящее время не поддерживают ограничения перехода; придется воспользоваться триггерными процедурами, но детали выходят за рамки настоящей книги). Однако ниже приведены формулировки, в которых используется соглашение об «именах переменных-отношений со штрихом», обсуждаемое в разделе «Разное»:

a.
```
CONSTRAINT CXA
      IS_EMPTY ( ( ( S′ WHERE CITY = 'Athens' ) { SNO } ) JOIN S )
                 WHERE CITY ≠ 'Athens'
                 AND   CITY ≠ 'London'
                 AND   CITY ≠ 'Paris' )
```

```
AND IS_EMPTY ( ( ( S' WHERE CITY = 'London' ) { SNO } ) JOIN S )
               WHERE CITY ≠ 'London'
               AND   CITY ≠ 'Paris' ) ;
```

b.
```
CONSTRAINT CXB IS_EMPTY
    ( P WHERE SUM ( !!SP , QTY ) > SUM ( !!SP' , QTY ) ) ;
```

c.
```
CONSTRAINT CXC IS_EMPTY
    ( S WHERE SUM ( !!SP' , QTY ) < 0.5 * SUM ( !!SP , QTY ) ) ;
```

Условие «за одну операцию обновления» важно, чтобы мы не пытались уменьшить общее количество деталей в поставке сначала, скажем на треть, а потом еще на треть.

Упражнение 8.26. См. текст главы.

Упражнение 8.27. *Без ответа.*

Упражнение 8.28. В SQL ограничения типа не поддерживаются по причине, связанной с наследованием типов. Детали выходят за рамки этой книги; если вам интересно, можете найти подробное обсуждение в книге «Databases, Types, and the Relational Model: The Third Manifesto», написанной Хью Дарвеном и мной (см. приложение D). Что касается последствий, то одно из них состоит в том, что при определении типа в SQL вы даже не можете указать, какие значения составляют данный тип! – разве что в виде априорного ограничения, налагаемого представлением. А потому – в отсутствие каких-либо иных средств контроля – в базе данных могут оказаться неправильные данные (даже бессмысленные, например, размер обуви 1000).

Упражнение 8.29. В принципе, все применимы, хотя в языке **Tutorial D** намеренно не предоставляется способ определить для нескалярных или системных типов какие-либо ограничения типа кроме априорных.

Упражнение 8.30. См. обсуждение потенциально недетерминированных типов в главе 12. *Другого ответа не будет.*

Глава 9

Упражнение 9.1.

```
VAR NON_COLOCATED VIRTUAL
  ( S { SNO } JOIN P { PNO } ) MINUS ( S JOIN P ) { SNO , PNO }
    KEY { SNO , PNO } ;

CREATE VIEW NON_COLOCATED AS
       SELECT SNO , PNO
       FROM   S , P
       WHERE  S.CITY <> P.CITY
            /* UNIQUE ( SNO , PNO ) */ ;
```

Упражнение 9.2. Подставляя определение представления вместо ссылки на представление во внешней фразе FRON (и явно показывая все квалификаторы имен), получаем:

```
SELECT DISTINCT LSSP.STATUS , LSSP.QTY
FROM ( SELECT S.SNO , S.SNAME , S.STATUS , SP.PNO , SP.QTY
       FROM   S NATURAL JOIN SP
       WHERE  S.CITY = 'London' ) AS LSSP
WHERE  LSSP.PNO IN
     ( SELECT P.PNO
       FROM   P
       WHERE  P.CITY <> 'London' )
```

Это выражение можно упростить:

```
SELECT DISTINCT STATUS , QTY
FROM   S NATURAL JOIN SP
WHERE  CITY = 'London'
AND    PNO IN
     ( SELECT PNO
       FROM   P
       WHERE  CITY <> 'London' )
```

Упражнение 9.3. Единственный ключ – {SNO,PNO}. Предикат: *Поставщик SNO связан контрактом, называется SNAME, имеет статус STATUS, находится в некотором городе и поставляет деталь PNO в количестве QTY.*

Упражнение 9.4.

a.
```
( ( P WHERE WEIGHT > 14.0 ) { PNO , WEIGHT , COLOR } )
                                          WHERE COLOR = 'Green'
```

b.
```
( EXTEND ( ( P WHERE WEIGHT > 14.0 ) { PNO , WEIGHT , COLOR } )
           ADD ( WEIGHT + 5.3 AS WTP ) ) { PNO , WTP }
```

c.
```
INSERT ( ( P WHERE WEIGHT > 14.0 ) { PNO , WEIGHT , COLOR } )
 RELATION { TUPLE { PNO 'P99', WEIGHT 12.0, COLOR 'Purple' } } ;
```

Отметим, что этот оператор INSERT логически эквивалентен реляционному присваиванию, в котором целевое выражение определено иначе, чем просто ссылка на переменную-отношение. Из возможности обновлять представления следует, что такие присваивания должны быть допустимы как синтаксически, так и семантически. Аналогичные замечания относятся также к пунктам (d) и (e) ниже.

d.
```
DELETE ( ( P WHERE WEIGHT > 14.0 ) { PNO , WEIGHT , COLOR } )
                                          WHERE WEIGHT < 9.0 ;
```

e.
```
UPDATE ( ( P WHERE WEIGHT > 14.0 ) { PNO , WEIGHT , COLOR } )
                     WHERE WEIGHT = 18.0 : { COLOR := 'White' } ;
```

Дополнительное упражнение: Какие еще упрощения можно применить к рассмотренным выше подстановкам?

Упражнение 9.5.

a.
```
( ( ( EXTEND P ADD ( WEIGHT * 454 AS WT ) ) WHERE WT > 6356.0 )
              { PNO , WT , COLOR } ) WHERE COLOR = 'Green'
```

b.
```
( EXTEND
( ( ( EXTEND P ADD ( WEIGHT * 454 AS WT ) ) WHERE WT > 6356.0 )
{ PNO , WT , COLOR } ) ADD ( WT + 5.3 AS WTP ) ) { PNO , WTP }
```

c.
```
INSERT
( ( ( EXTEND P ADD ( WEIGHT * 454 AS WT ) ) WHERE WT > 6356.0 )
    { PNO , WT , COLOR } )
RELATION { TUPLE { PNO 'P99' , WT 12.0 , COLOR 'Purple' } } ;
```

d.
```
DELETE
( ( ( EXTEND P ADD ( WEIGHT * 454 AS WT ) ) WHERE WT > 6356.0 )
    { PNO , WT , COLOR } ) WHERE WT < 9.0 ;
```

e.
```
UPDATE
( ( ( EXTEND P ADD ( WEIGHT * 454 AS WT ) ) WHERE WT > 6356.0 )
{ PNO , WT , COLOR } ) WHERE WT = 18.0 : { COL := 'White' } ;
```

Упражнение 9.6. Сначала SQL-версия определения представления из упражнения 9.4:

```
CREATE VIEW HEAVYWEIGHT AS
       SELECT PNO , WEIGHT AS WT , COLOR AS COL
       FROM   P
       WHERE  WEIGHT > 14.0 ;
```

Для пунктов a–e я сначала привожу SQL-аналог формулировки на **Tutorial D**, а затем результат, получающийся после подстановки:

a.
```
SELECT *
FROM   HEAVYWEIGHT
WHERE  COL = 'Green'

SELECT *
FROM ( SELECT PNO , WEIGHT AS WT , COLOR AS COL
       FROM   P
       WHERE  WEIGHT > 14.0 ) AS POINTLESS
WHERE  COL = 'Green'
```

Здесь и далее оставляю вам дальнейшее упрощение в качестве дополнительного упражнения (если явно не оговорено противное).

b.
```
SELECT PNO , WT + 5.3 AS WTP
FROM   HEAVYWEIGHT

SELECT PNO , WT + 5.3 AS WTP
```

```
FROM ( SELECT PNO , WEIGHT AS WT , COLOR AS COL
       FROM   P
       WHERE  WEIGHT > 14.0 ) AS POINTLESS
```

c.
```
INSERT INTO HEAVYWEIGHT ( PNO , WT , COL )
            VALUES ( 'P99' , 12.0 , 'Purple' ) ;

INSERT INTO ( SELECT PNO , WEIGHT AS WT , COLOR AS COL
              FROM   P
              WHERE  WEIGHT > 14.0 ) ( PNO , WT , COL )
              VALUES ( 'P99' , 12.0 , 'Purple' ) ;
```

Интересно отметить, что это предложение INSERT нарушает синтаксические правила SQL (потому что целевое выражение не является простой ссылкой на таблицу); таким образом, в данном случае обработка представлений в SQL в терминах одной лишь подстановки выражена быть не может. Поэтому приведенное выше предложение INSERT должно быть трансформировано в нечто иное, но во что именно, сказать трудно.

d.
```
DELETE FROM ( SELECT PNO , WEIGHT AS WT , COLOR AS COL
              FROM   P
              WHERE  WEIGHT > 14.0 ) WHERE WT < 9.0 ;
```

И снова в трансформированной версии применен недопустимый синтаксис SQL, но на этот раз отыскать допустимый эквивалент немного проще:

```
DELETE FROM P WHERE WEIGHT > 14.0 AND WEIGHT < 9.0 ;
```

(Это, как легко видеть, «пустая операция». Как вы думаете, сумеет оптимизатор распознать этот факт?)

e.
```
UPDATE ( SELECT PNO , WEIGHT AS WT , COLOR AS COL
         FROM   P
         WHERE  WEIGHT > 14.0 )
SET    COL = 'White'
WHERE  WT = 18.0 ;
```

Синтаксически допустимый эквивалент:

```
UPDATE P
SET    COLOR = 'White'
WHERE  WEIGHT = 18.0 AND WEIGHT > 14.0 ;
```

SQL-версии ответов к упражнению 9.5 *не приводятся*.

Упражнение 9.7. Пожалуй, только для HEAVYWEIGHT (первый вариант), потому что только его и можно рассматривать как обновляемое.

Упражнение 9.8. В обоих случаях единственный ключ {PNO}. Предикат представления из упражнения 9.4: *Деталь PNO используется на предприятии, имеет некоторое название и хранится на складе*

в некотором городе, имеет цвет COL и вес в фунтах WT (и WT больше 14). Предикат представления из упражнения 9.5: *Деталь PNO используется на предприятии, имеет некоторое название и хранится на складе в некотором городе, имеет цвет COL и вес в граммах WT (и WT больше 6356).* Что касается полных ограничений переменных-отношений: для представления из упражнения 9.4 полное ограничение в основном такое же, как для переменной-отношения P, но модифицировано с учетом переименования и проекции И ограничения, что вес должен быть больше 14. Полное ограничение для представления из упражнения 9.5 аналогично.

Упражнение 9.9. Вот несколько причин:

- Если пользователю разрешено оперировать представлениями вместо базовых переменных-отношений, то понятно, что представления должны выглядеть по возможности так же, как базовые переменные-отношения. На самом деле, в соответствии с *принципом взаимозаменяемости* пользователь вообще не должен знать, что работает с представлениями, и обращаться с ними может точно так же, как с базовыми переменными-отношениями. И если пользователь базовой переменной-отношения должен знать, какие для нее определены ключи (в общем случае), то и пользователь представления тоже должен знать ключи этого представления (опять же в общем случае).
- СУБД не всегда может вывести ключи (так обстоит дело почти во всех SQL-системах, представленных сегодня на рынке). Поэтому явные объявления, по-видимому, остаются единственным средством (для администратора базы данных – АБД) сообщить СУБД – и пользователям тоже – о существовании таких ключей.
- Даже если бы СУБД умела выводить ключи самостоятельно, явные объявления, по крайней мере, помогли бы системе проверить, что ее выводы и явные спецификации, заданные АБД, согласуются между собой.
- АБД может быть известна некоторая информация, которой СУБД не располагает, поэтому он может уточнить выводы СУБД.
- В тексте главы отмечалось, что с помощью такого средства можно было бы легко и удобно задавать некоторые важные ограничения, которые иначе пришлось бы формулировать очень многословно.

Дополнительное упражнение: Как вы думаете, какие из вышеперечисленных причин относятся не только к ограничениям ключей, но и к ограничениям целостности вообще?

Упражнение 9.10. Вот один такой пример: переменная-отношение «поставщики» равна соединению своих проекций на {SNO,SNAME}, {SNO,STATUS} и {SNO,CITY} – при условии, конечно, что действуют подходящие ограничения (верно?). Поэтому мы могли бы сделать эти

проекции базовыми переменными-отношениями, а их соединение – представлением.

Упражнение 9.11. Вот несколько замечаний на эту тему. Во-первых, сам процесс замены состоит из нескольких шагов, которые можно вкратце охарактеризовать следующим образом:

```
/* определить новые базовые переменные-отношения */

VAR LS BASE RELATION
  { SNO CHAR , SNAME CHAR , STATUS INTEGER , CITY CHAR }
  KEY { SNO } ;

VAR NLS BASE RELATION
  { SNO CHAR , SNAME CHAR , STATUS INTEGER , CITY CHAR }
  KEY { SNO } ;

/* скопировать данные в новые базовые переменные-отношения */

INSERT LS  ( S WHERE CITY = 'London' ) ;

INSERT NLS ( S WHERE CITY ≠ 'London' ) ;

/* удалить старую переменную-отношение */

DROP VAR S ;

/* создать требуемое представление */

VAR S VIRTUAL ( LS D_UNION NLS ) ;
```

Далее нужно как-то поступить с внешним ключом переменной-отношения SP, который ссылался на старую переменную-отношение S. Очевидно, лучше всего было бы сделать так, чтобы этот внешний ключ теперь ссылался на представление S;[1] если это невозможно (а так оно и есть в большинстве современных продуктов), то можно было бы определить еще одну базовую переменную-отношение следующим образом:

```
VAR SS BASE RELATION { SNO CHAR } KEY { SNO } ;
```

И скопировать данные (понятно, что это следует делать до удаления переменной-отношения S):

```
INSERT SS S { SNO } ;
```

[1] На самом деле, логическая независимость от данных – сильный аргумент в пользу того, чтобы разрешить определять любые ограничения не только для базовых переменных-отношений, но и для представлений. См. дополнительное упражнение в конце ответа к упражнению 9.9.

Теперь следует добавить такую спецификацию внешнего ключа в определения переменных-отношений LS и NLS:

```
FOREIGN KEY { SNO } REFERENCES SS
```

И наконец, необходимо так изменить спецификацию внешнего ключа {SNO} в переменной-отношении SP, чтобы он ссылался на SS вместо S.

Упражнение 9.12. a. *Без ответа* – отмечу лишь, что если дать ответ на этот вопрос для конкретного продукта затруднительно, то сам этот факт и составляет смысл упражнения. **b.** То же, что для **(a)**. **c.** *Без ответа*.

Упражнение 9.13. Описание различий см. в тексте главы. В настоящее время SQL не поддерживает снимки. (Поддерживается предложение CREATE TABLE AS – см. ответ к упражнению 1.16 в главе 1, – которое позволяет инициализировать базовую таблицу в момент создания, но у этого предложения нет опции REFRESH.)

Упражнение 9.14. Термин «материализованное представление» – не рекомендованный синоним термина *снимок*. Он не рекомендуется, потому что смешивает понятия, которые логически различны и таковыми должны оставаться. По определению представления с точки зрения модели не материализуются, так что мы оказываемся в ситуации, когда отсутствует четкий термин для понятия, которое исходно было определено вполне однозначно. Поэтому термина «материализованное представление» следует всячески избегать.[1] На самом деле меня так и подмывает сделать следующий напрашивающийся вывод: мне кажется, что поддержка термина «материализованное представление» выдает непонимание реляционной модели в частности и различия между моделью и реализацией вообще.

Упражнение 9.15. Сначала приведу определение варианта (b) в терминах варианта (a):

```
VAR SSP VIRTUAL ( S JOIN SP )
    KEY { SNO , PNO } ;

VAR XSS VIRTUAL ( S NOT MATCHING SP )
    KEY { SNO } ;
```

А вот как выглядит определение варианта (a) в терминах варианта (b):

```
VAR S VIRTUAL ( XSS D_UNION SSP { ALL BUT PNO , QTY } )
    KEY { SNO } ;

VAR SP VIRTUAL ( SSP { SNO , PNO , QTY } )
    KEY { SNO , PNO } ;
```

[1] Понимаю, что эту битву я, скорее всего, уже проиграл, но остаюсь вечным оптимистом.

К обоим вариантам применимы следующие ограничения базы данных:

```
CONSTRAINT DESIGN_A IS_EMPTY ( SP { SNO } MINUS S { SNO } ) ;

CONSTRAINT DESIGN_B IS_EMPTY ( SSP { SNO } JOIN XSS { SNO } ) ;
```

При таких ограничениях эти варианты представления информации эквивалентны. Но вариант (a) лучше, потому что обе переменные-отношения в нем полностью нормализованы. В варианте же (b) переменная-отношение SSP нормализована не до конца (она даже не находится во второй нормальной форме), а это значит, что она избыточна и потенциально подвержена различным «аномалиям при обновлении». Подумайте, что произойдет в варианте (b), если какой-нибудь поставщик вообще прекратит поставку деталей или же поставщик, который раньше ничего не поставлял, начнет поставлять какие-то детали. Более пристальное обсуждение проблем, присущих варианту (b), выходит за рамки этой книги, но в некоторых публикациях, упоминаемых в приложении D, они рассматриваются подробно (см. также приложение B).

Замечу мимоходом, что, учитывая тот факт, что {SNO} – ключ для переменной-отношения S, ограничение DESIGN_A дает пример еще одной формулировки ссылочного ограничения.

Упражнение 9.16. (Советую еще раз прочитать раздел «Зависимость от имен атрибутов» в главе 6 перед тем, как знакомиться с ответом.) Да, представлений действительно должно быть достаточно для решения проблемы логической независимости от данных. Но беда в том, что определение представления обычно воспринимают как взгляд приложения на некоторую часть базы данных и одновременно соответствие между этим взглядом и «настоящей» базой данных. Чтобы добиться такой независимости от данных, которую я здесь имею в виду, эти две спецификации следует отделять друг от друга.

Глава 10

Упражнение 10.1. См. ответ к упражнению 4.10 в главе 4.

Упражнение 10.2. Прежде всего, легко видеть, что одновременно OR и AND не нужны, так как

```
p AND q  ≡  NOT ( NOT ( p ) OR NOT ( q ) )
```

(Эта эквиваленция легко устанавливается с помощью таблиц истинности.) Отсюда следует, что ниже мы можем свободно пользоваться OR и AND.

Далее рассмотрим связки, в которых участвует единственное высказывание *p*. Пусть *c(p)* – рассматриваемая связка. Существуют следующие возможности:

```
c(p)  ≡  p OR NOT ( p )    /* всегда TRUE  */
c(p)  ≡  p AND NOT ( p )   /* всегда FALSE */
c(p)  ≡  p                 /* тождество    */
c(p)  ≡  NOT ( p )         /* NOT          */
```

Теперь рассмотрим связки с двумя высказываниями *p* и *q*. Пусть *c(p,q)* – рассматриваемая связка. Существуют следующие возможности:

```
c (p,q)  ≡  p OR NOT ( p ) OR q OR NOT ( q )
c (p,q)  ≡  p AND NOT ( p ) AND q AND NOT ( q )
c (p,q)  ≡  p
c (p,q)  ≡  NOT ( p )
c (p,q)  ≡  q
c (p,q)  ≡  NOT ( q )
c (p,q)  ≡  p OR q
c (p,q)  ≡  p AND q
c (p,q)  ≡  p OR NOT ( q )
c (p,q)  ≡  p AND NOT ( q )
c (p,q)  ≡  NOT ( p ) OR q
c (p,q)  ≡  NOT ( p ) AND q
c (p,q)  ≡  NOT ( p ) OR NOT ( q )
c (p,q)  ≡  NOT ( p ) AND NOT ( q )
c (p,q)  ≡  ( NOT ( p ) OR q ) AND ( NOT ( q ) OR p )
c (p,q)  ≡  ( NOT ( p ) AND q ) OR ( NOT ( q ) AND p )
```

В качестве дополнительного упражнения и чтобы убедиться, что приведенные выше определения действительно покрывают все возможности, можете построить соответствующие таблицы истинности.

Теперь обратимся к части (b) упражнения. На самом деле существует два таких примитива: NOR и NAND, которые часто обозначаются, соответственно, направленной вниз стрелкой, «↓» (стрелка Пирса), и вертикальной черточкой, «|» (штрих Шеффера). Вот их таблицы истинности:

NOR	T	F
T	F	F
F	F	T

NAND	T	F
T	F	T
F	T	T

Как следует из этих таблиц, $p \downarrow q$ («*p* NOR *q*») эквивалентно NOT (*p* OR *q*), а $p|q$ («*p* NAND *q*») эквивалентно NOT (p AND q). Далее я буду рассматривать только NOR (NAND оставляю вам). Замечу, что эту связку можно назвать «ни-ни» («ни первый, ни второй опе-

ранд не являются истинными»). Теперь я покажу, как в терминах этого оператора определить NOT, OR и AND:

```
NOT ( p )    ≡  p ↓ p
p OR q       ≡  ( p ↓ q ) ↓ ( p ↓ q )
p AND q      ≡  ( p ↓ p ) ↓ ( q ↓ q )
```

К примеру, рассмотрим более внимательно случай «p AND q»:

p	*q*	*p*↓*p*	*q*↓*q*	(*p*↓*p*) ↓ (*q*↓*q*)
T	T	F	F	T
T	F	F	T	F
F	T	T	F	F
F	F	T	T	F

Эта таблица истинности доказывает, что выражение (*p*↓*p*)↓(*q*↓*q*) эквивалентно *p* AND *q*, поскольку первый, второй и последний столбец как раз и образуют таблицу истинности для AND:

p	*q*	*p* AND *q*
T	T	T
T	F	F
F	T	F
F	F	F

Поскольку ранее мы видели, что все остальные связки можно выразить через NOT, OR и AND, то требуемое утверждение доказано.

Упражнение 10.3. «Солнце – звезда» и «Луна – звезда» – высказывания, хотя первое истинно, а второе ложно.

Упражнение 10.4. *Солнце* удовлетворяет этому предикату, а *Луна* нет.

Упражнение 10.5. Вместо параметра можно подставить любой аргумент, лишь бы он был подходящего типа. А вместо обозначения вообще ничего нельзя подставить; оно – как ссылка на переменную в языке программирования – просто «обозначает» значение соответствующей переменной в определенный момент времени (в нашем случае, при проверке ограничения).

Упражнение 10.6. «Получить названия таких поставщиков, что существует поставка – единственная поставка, – связывающая их со всеми деталями». Отметим, что этот запрос вернет либо (а) все названия поставщиков, если кардинальность переменной-отношения P меньше двух, либо (б) пустой результат в противном случае.

Упражнение 10.7. Следующие SQL-выражения являются наиболее близкими аналогами соответствующих реляционных выражений:

Пример 1. Получить все пары номеров поставщиков такие, что оба поставщика находятся в одном городе.

```
SELECT SX.SNO AS SA , SY.SNO AS SB
FROM   S AS SX , S AS SY
WHERE  SX.CITY = SY.CITY
AND    SX.SNO < SY.SNO
```

Пример 2. Получить названия поставщиков, которые поставляют хотя бы одну красную деталь.

```
SELECT DISTINCT SX.SNAME
FROM   S AS SX
WHERE  EXISTS
     ( SELECT *
       FROM   SP AS SPX
       WHERE  EXISTS
            ( SELECT *
              FROM   P AS PX
              WHERE  SX.SNO = SPX.SNO
              AND    SPX.PNO = PX.PNO
              AND    PX.COLOR = 'Red' ) )
```

Пример 3. Получить названия поставщиков, которые поставляют хотя бы одну деталь, поставляемую поставщиком S2.

```
SELECT DISTINCT SX.SNAME
FROM   S AS SX
WHERE  EXISTS
     ( SELECT *
       FROM   SP AS SPX
       WHERE  EXISTS
            ( SELECT *
              FROM   SP AS SPY
              WHERE  SX.SNO = SPX.SNO
              AND    SPX.PNO = SPY.PNO
              AND    SPY.SNO = 'S2' ) )
```

Пример 4. Получить названия поставщиков, которые не поставляют деталь P2.

```
SELECT DISTINCT SX.SNAME
FROM   S AS SX
WHERE  NOT EXISTS
         ( SELECT *
           FROM   SP AS SPX
           WHERE  SPX.SNO = SX.SNO
           AND    SPX.PNO = 'P2' )
```

Пример 5. Для каждой поставки получить всю информацию о поставке, а также ее общий вес.

```
SELECT SPX.* , PX.WEIGHT * SPX.QTY AS SHIPWT
FROM   P AS PX , SP AS SPX
WHERE  PX.PNO = SPX.PNO
```

Пример 6. Для каждой детали получить номер детали и общее поставленное количество.

```
SELECT PX.PNO ,
     ( SELECT COALESCE ( SUM ( ALL SPX.QTY ) , 0 )
       FROM   SP AS SPX
       WHERE  SPX.PNO = PX.PNO ) AS TOTQ
FROM   P AS PX
```

Пример 7. Получить города, в которых хранится более пяти красных деталей.

```
SELECT DISTINCT PX.CITY
FROM   P AS PX
WHERE  ( SELECT COUNT ( * )
         FROM   P AS PY
         WHERE  PY.CITY = PX.CITY
         AND    PY.COLOR = 'Red' ) > 5
```

Упражнение 10.8. Следующая таблица истинности доказывает, что операция AND ассоциативна; доказательство для OR аналогично.

p	*q*	*r*	*p* AND *q*	(*p* AND *q*) AND *r*	(*q* AND *r*)	*p* AND (*q* AND *r*)
T	T	T	T	T	T	T
T	T	F	T	F	F	F
T	F	T	F	F	F	F
T	F	F	F	F	F	F
F	T	T	F	F	T	F
F	T	F	F	F	F	F
F	F	T	F	F	F	F
F	F	F	F	F	F	F

Упражнение 10.9. a. Неверно (предположим, что x пробегает пустое множество и q равно TRUE, тогда EXISTS x (q) равно FALSE). **b.** Неверно (предположим, что x пробегает пустое множество и q равно FALSE, тогда FORALL x (q) равно TRUE). **c.** Верно. **d.** Верно. **e.** Неверно (предположим, что x пробегает пустое множество, тогда FORALL x ($p(x)$) равно TRUE, но EXISTS x ($p(x)$) равно FALSE, а TRUE $\Rightarrow$ FALSE равно FALSE). **f.** Неверно (предположим, что x пробегает пустое множество, тогда EXISTS x (TRUE) равно FALSE). **g.** Неверно (предположим, что x пробегает пустое множество, тогда FORALL x (FALSE) равно TRUE). **h.** Верно. **i.** Неверно (например, сказать, что ровно одно целое число равно нулю, не то же самое, что сказать, что все целые числа равны нулю). **j.** Неверно (например, из того, что во всех сутках 24 часа, и из того, что существуют хотя бы одни сутки, в которых 24 часа, не следует, что ровно в одних сутках 24 часа). **k.** Верно. Отметим, что (правильные!) эквиваленции и импликации, подобные рассматриваемым, можно положить в основу правил трансформации выражений исчисления, похожих на правила трансформации алгебраических выражений, которые обсуждались в главе 6.

Упражнение 10.10. a. Верно. **b.** Верно. **c.** Верно. **d.** Верно. **e.** Неверно (например, сказать, что для каждого целого числа y существует большее целое число x, не то же самое, что сказать, что существует целое число x, большее всех целых чисел y). **f.** Верно.

Упражнение 10.11. a. Верно. **b.** Верно.

Упражнение 10.12. Я привожу лишь такие решения, для которых можно отметить что-то интересное. Упражнения из главы 6:

6.12. Следующие выражения реляционного исчисления обозначают соответственно TABLE_DEE и TABLE_DUM:

```
{ } WHERE TRUE

{ } WHERE FALSE
```

А следующее выражение обозначает проекцию текущего значения переменной-отношения S на пустое множество атрибутов:

```
{ } WHERE EXISTS ( SX )
```

Обычно не считают, что в реляционном исчислении имеется прямой аналог выражения *r*{ALL BUT ...} языка **Tutorial D**, но в принципе нет причин, по которым его не могло бы быть.

6.15. Обычно не считают, что в реляционном исчислении имеется прямой аналог выражения D_UNION языка **Tutorial D**, но в принципе нет причин, по которым его не могло бы быть.

Упражнение 10.12 (продолжение). Упражнения из главы 7:

7.1.

d.
```
{ PX } WHERE SUM ( SPX WHERE SPX.PNO = PX.PNO , QTY ) < 500
```
e.
```
{ PX } WHERE EXISTS ( SX WHERE SX.CITY = PX.CITY )
```
j.
```
{ PX , COUNT ( SPX WHERE SPX.PNO = PX.PNO ) AS SCT }
```

7.8. В реляционном исчислении аналогом выражения SP GROUP ({} AS X) языка **Tutorial D** является выражение:

```
{ SPX , { } AS X }
```

7.10. В реляционном исчислении аналогом выражения (R4 WHERE TUPLE {PNO 'P2'} ∈ PNO_REL){SNO} языка **Tutorial D** является выражение:

```
RANGEVAR RX RANGES OVER R4 ,
RANGEVAR RY RANGES OVER RX.PNO_REL ;

RX.SNO WHERE EXISTS ( RY WHERE RY.PNO = 'P2' )
```

Отметим, что определение RY выше зависит от RX (определения разделяются запятой, а не точкой с запятой, и потому считаются одной операцией).

А вот аналог выражения ((R4 WHERE SNO = 'S2') UNGROUP (PNO_REL)){PNO}:

```
RY.PNO WHERE RX.SNO = 'S2'
```

7.11. `{ SX , { SPX.PNO WHERE SPX.SNO = SX.SNO } AS PNO_REL }`

7.12. На практике нам необходимы аналоги традиционных операторов INSERT, DELETE и UPDATE (и реляционного присваивания), которые соответствовали бы духу реляционного исчисления, а не алгебры (и это замечание остается справедливым вне зависимости от того, говорим мы о переменных-отношениях с RVA-атрибутами, как в данном контексте, или без них). Детали выходят за рамки этой книги, но в любом случае не вызывают затруднений. *Более подробного ответа не будет.*

Упражнение 10.12 (продолжение). *Упражнения из главы 8: без ответа.*

Упражнение 10.12 (продолжение). *Упражнения из главы 9: без ответа.*

Упражнение 10.13. В тексте главы отмечалось, что множество, которое пробегает переменная кортежа, всегда является телом некоторого отношения – обычно, *но не всегда*, отношения, которое является текущим значением некоторой переменной-отношения (обратите внимание на курсив). В данном примере переменная кортежа пробегает множество, которое, по существу, является объединением:

```
RANGEVAR CX RANGES OVER { SX.CITY } , { PX.CITY } ;

{ CX } WHERE TRUE
```

Отметим, что в определении переменной кортежа CX используются переменные кортежа SX и PX, которые, как я предполагаю, были определены ранее.

Упражнение 10.14. Для доказательства реляционной полноты SQL достаточно[1] показать, что (а) существуют SQL-выражения для каждой из алгебраических операций ограничения, проекции, произведения, объединения и разности и что (б) операндами этих SQL-выражений могут быть произвольные SQL-выражения.

Прежде всего, как мы знаем, SQL поддерживает оператор реляционной алгебры RENAME благодаря наличию необязательной спецификации AS для элементов фразы SELECT.[2] Поэтому можно гаранти-

---

[1] Достаточно, но необходимо ли?

[2] Немного более точно: SQL-аналогом алгебраического выражения T RENAME (A AS B) служит (крайне неудобное!) SQL-выражение SELECT A AS B, X, Y, . . . , Z FROM T (где X, Y, ..., Z – все столбцы T за исключением A, и я закрываю глаза на то, что это SQL-выражение дает таблицу, столбцы которой упорядочены слева направо).

ровать, что столбцы во всех таблицах имеют надлежащие имена, а, значит, операнды произведения, объединения и разности удовлетворяют требованиям алгебры, предъявляемым к именам столбцов. Далее – при условии, что требования к именам столбцов действительно удовлетворяются, – правила наследования имен столбцов в SQL совпадают с правилами алгебры, которые были описаны в главе 6.

Вот SQL-выражения, приблизительно соответствующие пяти примитивным операторам:

```
Алгебра                 SQL

R WHERE p               SELECT * FROM R WHERE p

R { A , B , ... , C }   SELECT DISTINCT A , B , ... , C FROM R

R1 TIMES R2             SELECT * FROM R1 , R2

R1 UNION R2             SELECT * FROM R1
                        UNION  CORRESPONDING
                        SELECT * FROM R2

R1 MINUS R2             SELECT * FROM R1
                        EXCEPT CORRESPONDING
                        SELECT * FROM R2
```

Далее, (а) *R1* и *R2* в приведенных выше SQL-выражениях фактически являются ссылками на таблицы, и (б) если взять любое из пяти показанных SQL-выражений и заключить его в скобки, то снова получится ссылка на таблицу.[1] Отсюда следует, что SQL – действительно реляционно полный язык.

Примечание

В предыдущем рассуждении есть один изъян – SQL не поддерживает проецирование на пустое множество столбцов (поскольку не поддерживает пустые фразы SELECT). Следовательно, не поддерживаются также TABLE_DEE и TABLE_DUM, а, значит, SQL все-таки не является на 100 процентов реляционно полным.

Упражнение 10.15. Пусть *TP* и *DC* обозначают, соответственно, высказывания «база данных содержит только истинные высказывания» и «база данных непротиворечива». Тогда первый пункт означает:

```
IF TP THEN DC
```

а второй:

```
IF NOT ( DC ) THEN NOT ( TP )
```

[1] Я игнорирую тот факт, что SQL в таких ссылках на таблицы настаивает на включении бессмысленного определения переменной кортежа.

Если вспомнить определение логической импликации в терминах NOT и OR, то легко видеть, что эти два высказывания на самом деле эквивалентны. (В действительности, второе – контрапозиция первого. Дополнительные объяснения см. в главе 11.)

Упражнение 10.16. Нет, хотя в учебниках по формальной логике обычно утверждается противоположное, и на практике она «обычно» достижима. В статье «A Remark on Prenex Normal Form» (см. приложение D) содержится дополнительная информация.

Глава 11

Упражнение 11.1. Сначала приведу формулировку на SQL запроса «Получить поставщиков SX таких, что для любых деталей PX и PY, если PX.CITY ≠ PY.CITY, то SX не поставляет обе детали». (Насколько эта формулировка отличается от данной в тексте главы?)

```
SELECT SX.*
FROM   S AS SX
WHERE  NOT EXISTS
     ( SELECT *
       FROM   P AS PX
       WHERE  EXISTS
            ( SELECT *
              FROM   P AS PY
              WHERE  PX.CITY <> PY.CITY
              AND    EXISTS
                   ( SELECT *
                     FROM   SP AS SPX
                     WHERE  SPX.SNO = SX.SNO
                     AND    SPX.PNO = PX.PNO )
              AND    EXISTS
                   ( SELECT *
                     FROM   SP AS SPX
                     WHERE  SPX.SNO = SX.SNO
                     AND    SPX.PNO = PY.PNO ) ) )
```

Далее требовалось дать SQL-формулировки (a) в которых используются GROUP BY и HAVING, (b) в которых не используются GROUP BY и HAVING, для следующих запросов:

- Получить номера поставщиков, которые поставляют N различных деталей для некоторого $N > 3$.
- Получить номера поставщиков, которые поставляют N различных деталей для некоторого $N < 4$.

Вот как выглядят формулировки с GROUP BY и HAVING:

```
SELECT SNO
FROM   SP
```

```
GROUP  BY SNO
HAVING COUNT ( * ) > 3

SELECT SNO
FROM   SP
GROUP  BY SNO
HAVING COUNT ( * ) < 4
UNION  CORRESPONDING
SELECT SNO
FROM   S
WHERE  SNO NOT IN
     ( SELECT SNO
       FROM   SP )
```

А вот без GROUP BY и HAVING:

```
SELECT SNO
FROM   S
WHERE  ( SELECT COUNT ( * )
         FROM   SP
         WHERE  SP.SNO = S.SNO ) > 3

SELECT SNO
FROM   S
WHERE  ( SELECT COUNT ( * )
         FROM   SP
         WHERE  SP.SNO = S.SNO ) < 4
```

Также требовалось ответить на вопрос, какие выводы можно сделать из этого упражнения. Лично я заключаю, что нужно проявлять большую осмотрительность при использовании GROUP BY и HAVING. В частности, отметим, что формулировки на естественном языке симметричны, тогда как запросы с использованием GROUP-BY/HAVING, очевидно, несимметричны. А вот запросы без GROUP BY и HAVING, напротив, симметричны.

Упражнение 11.2. *Без ответа.*

Упражнение 11.3. *Без ответа.*

Упражнение 11.4. Разумеется, я не собираюсь давать сколько-нибудь полный ответ к этому упражнению, но, по крайней мере, замечу, что следующие эквиваленции позволяют преобразовать некоторые алгебраические выражения в выражения реляционного исчисления, и наоборот:

- *r* WHERE *bx1* AND *bx2* ≡ (*r* WHERE *bx1*) JOIN (*r* WHERE *bx2*)
- *r* WHERE bx1 OR bx2 ≡ (*r* WHERE *bx1*) UNION (*r* WHERE *bx2*)
- *r* WHERE NOT (*bx*) ≡ *r* MINUS (*r* WHERE *bx*)

В тексте книги (начиная с главы 6) упоминались и некоторые другие трансформации.

Глава 12

Упражнение 12.1.

A NATURAL JOIN *B* : Допустимо

A INTERSECT *B* : Допустимо

TABLE *A* NATURAL JOIN TABLE *B* : Допустимо

TABLE *A* INTERSECT TABLE *B* : Допустимо

SELECT * FROM *A* NATURAL JOIN SELECT * FROM *B* : Недопустимо

SELECT * FROM *A* INTERSECT SELECT * FROM *B* : Допустимо

(SELECT * FROM *A*) NATURAL JOIN (SELECT * FROM *B*) : Недопустимо

(SELECT * FROM *A*) INTERSECT (SELECT * FROM *B*) : Допустимо

(TABLE *A*) NATURAL JOIN (TABLE *B*) : Недопустимо

(TABLE *A*) INTERSECT (TABLE *B*) : Допустимо

(TABLE *A*) AS *AA* NATURAL JOIN (TABLE *B*) AS *BB* : Допустимо

(TABLE *A*) AS *AA* INTERSECT (TABLE *B*) AS *BB* : Недопустимо

((TABLE *A*) AS *AA*) NATURAL JOIN ((TABLE B) AS *BB*) : Недопустимо

((TABLE *A*) AS *AA*) INTERSECT ((TABLE *B*) AS *BB*) : Недопустимо

Из этого упражнения я делаю вывод, что эти правила очень трудно запомнить (это еще мягко сказано). В частности, SQL-выражения с INTERSECT не всегда можно напрямую трансформировать в эквивалентные выражения с JOIN.

Упражнение 12.2. *Без ответа.*

Упражнение 12.3. *Без ответа.*

Приложение B

Упражнение B.1. *Без ответа.*

Упражнение B.2. Полное множество ФЗ (формально оно называется *замыканием*, хотя не имеет ничего общего со свойством замкнутости реляционной алгебры) для переменной-отношения SP таково:

```
{ SNO , PNO , QTY } → { SNO , PNO , QTY }
{ SNO , PNO , QTY } → { SNO , PNO }
{ SNO , PNO , QTY } → { PNO , QTY }
{ SNO , PNO , QTY } → { SNO , QTY }
{ SNO , PNO , QTY } → { SNO }
{ SNO , PNO , QTY } → { PNO }
{ SNO , PNO , QTY } → { QTY }
{ SNO , PNO , QTY } → { }

{ SNO , PNO }       → { SNO , PNO , QTY }
{ SNO , PNO }       → { SNO , PNO }
```

```
{ SNO , PNO }       → { PNO , QTY }
{ SNO , PNO }       → { SNO , QTY }
{ SNO , PNO }       → { SNO }
{ SNO , PNO }       → { PNO }
{ SNO , PNO }       → { QTY }
{ SNO , PNO }       → { }

{ PNO , QTY }       → { PNO , QTY }
{ PNO , QTY }       → { PNO }
{ PNO , QTY }       → { QTY }
{ PNO , QTY }       → { }

{ SNO , QTY }       → { SNO , QTY }
{ SNO , QTY }       → { SNO }
{ SNO , QTY }       → { QTY }
{ SNO , QTY }       → { }

{ SNO }             → { SNO }
{ SNO }             → { }

{ PNO }             → { PNO }
{ PNO }             → { }

{ QTY }             → { QTY }
{ QTY }             → { }

{ }                 → { }
```

Упражнение В.3. Истинно (из утверждения «если в двух кортежах одинаковы значения A, то одинаковы и значения B» следует, что проекции этих кортежей на {A,B} равны). *Примечание*: В тексте приложения я заметил, что понятие проекции кортежа столь же осмысленно, как понятия проекции отношения, и аналогичное замечание справедливо для некоторых других реляционных операторов (например, переименования и расширения). Технически мы говорим, что такие операторы *перегружены*.

Упражнение В.4. *Без ответа* (легкое упражнение).

Упражнение В.5. Теорема Хита утверждает: если $R\{A,B,C\}$ удовлетворяет ФЗ $A \rightarrow B$, то R равно соединению своих проекций $R1$ на $\{A,B\}$ и $R2$ на $\{A,C\}$. В следующем простом доказательстве этой теоремы я пользуюсь неформальной сокращенной нотацией для кортежей, которая была введена в тексте приложения.

Сначала я покажу, что в результате взятия проекций и последующего их обратного соединения ни один кортеж R не теряется. Пусть $(a,b,c) \in R$. Тогда $(a,b) \in R1$ и $(a,c) \in R2$, поэтому $(a,b,c) \in R1$ JOIN $R2$.

Теперь я покажу, что любой кортеж соединения действительно является кортежем R (иначе говоря, соединение не порождает «лишних» кортежей). Пусть $(a,b,c) \in R1$ JOIN $R2$. Чтобы породить такой кортеж

в соединении, мы должны иметь $(a,b) \in R1$ и $(a,c) \in R2$. Следовательно, чтобы можно было породить кортеж $(a,c) \in$ R2, должен существовать кортеж $(a,b',c) \in R$ для некоторого b'. Тогда должно быть $(a,b') \in R1$. Итак, мы имеем $(a,b) \in R1$ и $(a,b') \in R1$; поэтому должно быть $b = b'$, поскольку $A \rightarrow B$. Отсюда $(a,b,c) \in R$.

Обратная теорема Хита утверждала бы, что если $R\{A,B,C\}$ равно соединению своих проекций на $\{A,B\}$ и на $\{A,C\}$, то R удовлетворяет ФЗ $A \rightarrow B$. Но это неверно. Чтобы убедиться в этом, достаточно привести контрпример; оставляю эту задачу вам. (Если сдаетесь, посмотрите ответ к упражнению B.20. Обратите также внимание на переменную-отношение SPJ на рис. B.5 в тексте этого приложения.)

Упражнение B.6. См. текст приложения.

Упражнение B.7. См. главу 5.

Упражнение B.8. Определения см. в тексте главы. Первая является частным случаем второй. Например, переменная-отношение S удовлетворяет тривиальной ФЗ {CITY,STATUS} → {STATUS}. Применяя теорему Хита, мы видим, что S удовлетворяет тривиальной ЗС *{*AB*,*AC*}, где *A* – {CITY,STATUS}, *B* – {STATUS}, а *C* – {SNO,SNAME}.

Упражнение B.9. ФЗ по сути дела представляет собой суждение (на самом деле, высказывание) вида $A \rightarrow B$, где A и B – подмножества заголовка R. Так как у множества из n элементов есть 2^n различных подмножеств, то A и B могут принимать 2^n значений, следовательно, количество возможных ФЗ сверху ограничено величиной 2^{2n}. Например, для переменной-отношения степени 5 верхняя граница составляет 1024.

Упражнение B.10. Пусть заданной ФЗ удовлетворяет переменная-отношение R. Для любого кортежа t, принадлежащего R, все его подкортежи, соответствующие пустому множеству атрибутов, принимают одно и то же значение (а именно, 0-кортеж). Если B пусто, то ФЗ $A \rightarrow B$ тривиально удовлетворяется для всех возможных множеств A атрибутов R; фактически это *тривиальная* ФЗ (и потому не очень интересная). С другой стороны, если A пусто, то ФЗ $A \rightarrow B$ означает, что все кортежи R принимают одно и то же значение для B (поскольку значения для A, очевидно, одинаковы). А если B при этом состоит из «всех атрибутов R» – иными словами, если R имеет пустой ключ, – то R не может иметь более одного кортежа.

Упражнение B.11. Предположим, что мы начали с переменной-отношения R с атрибутами D, P, S, L, T и C (имена атрибутов очевидным образом соответствуют параметрам предиката). Тогда R удовлетворяет следующим нетривиальным ФЗ:

```
{ L }         → { D , P , C , T }
{ D , P , C } → { L , T }
{ D , P , T } → { L , C }
{ D , P , S } → { L , C }
```

Один их возможных наборов переменных-отношений, находящихся в НФБК, выглядит так:

```
SCHEDULE { L , D , P , C , T }
         KEY { L }
         KEY { D , P , C }
         KEY { D , P , T }

STUDYING { S , L }
         KEY { S , L }
```

STUDYING находится в 6НФ; SCHEDULE – в 5НФ, но не в 6НФ. При такой декомпозиции «потерялась» ФЗ {D,P,T} → {L,C}; более того, если бы мы разложили SCHEDULE на 6НФ-проекции на {L,D}, {L,P}, {L,C} и {L,T} (а при желании мы, конечно, можем это сделать), то дополнительно «потеряли» бы ФЗ {D,P,C} → {L,T} и {D,P,T} → {L,C}. Поэтому продолжать декомпозицию, пожалуй, не стоит.

Упражнение В.12. Да, иногда, хотя и не очень часто; на самом деле, именно эта возможность и лежит в основе различия между 5НФ и 4НФ. В подразделе «Еще о 5НФ» в тексте этого приложения приводится пример переменной-отношения, которую можно без потерь разложить на три, но не на две проекции. Такую декомпозицию можно даже назвать желательной, потому что (повторюсь) она уменьшает избыточность и, стало быть, позволяет избежать аномалий при обновлении.

Упражнение В.13. Суррогатные ключи – *не* то же самое, что идентификаторы кортежей. Во-первых (начнем с очевидного), суррогатный ключ идентифицирует сущность, а идентификатор кортежа – кортежи, а между сущностями и кортежами, безусловно, не существует взаимно однозначного соответствия. (Вспомните, например, о производных кортежах, которые возникают в результате выполнения некоторого запроса. Собственно, даже неясно, а есть ли у производных кортежей идентификатор.) Далее, идентификаторы кортежей имеют непосредственное касательство к производительности, а суррогатные ключи – нет (обычно предполагается, что доступ к кортежу по его идентификатору производится быстро, но о суррогатных ключах этого не скажешь). Кроме того, идентификаторы кортежей обычно скрыты от пользователя, а суррогатные ключи должны быть ему видны (в силу *принципа информации* – см. приложение А); другими словами, идентификатор кортежа, скорее всего, невозможно (и нежелательно!) хранить в переменной-отношении базы данных, тогда как хранить там суррогатный ключ возможно и желательно. Резюмируем: суррогатные ключи относятся к логической структуре базы данных, а идентификаторы кортежей – к физической.

Можно ли назвать идею суррогатных ключей здравой? Для начала отметим, что сама реляционная модель об этом ничего не говорит; как и вообще все проектирование баз данных, вопрос о том, исполь-

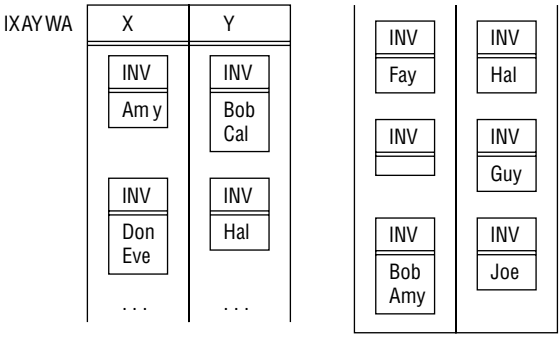
зовать суррогатные ключи или нет, относится к *применению* реляционной модели, а не к самой модели.

Однако я должен сказать, что этот вопрос далеко не прост. Существуют сильные доводы как за, так и против суррогатных ключей, причем их так много, что я не смогу разобрать все. Вместо этого я просто отсылаю вас к собственной подробной статье на эту тему «Composite Keys» (см. приложение D). *Примечание*: Эта статья называется «Составные ключи», потому что на практике суррогатные ключи чаще всего бывают полезны, когда имеющиеся ключи (и соответствующие им внешние ключи) являются составными.

И последнее замечание о суррогатах. Поскольку переменная-отношение, находящаяся в НФБК и не имеющая составных ключей, «автоматически» находится в 5НФ, многие, похоже, считают, что стоит ввести суррогатный ключ в НФБК-переменную, как она тут же окажется в 5НФ. Отнюдь нет. В частности, если переменная-отношение имела составной ключ до введения суррогатного, то будет иметь и после этого.

Упражнение B.14. В этом упражнении я буду использовать нотацию $X \rightarrow Y$ для обозначения того, что если каждый человек, принадлежащий множеству X, придет на вечеринку, то и каждый человек, принадлежащий множеству Y, тоже придет (такой выбор нотации не совсем произволен). А выражения вида $X \rightarrow Y$ я буду называть *утверждениями*.

Таким образом, начальный и самый очевидный проект состоит из единственной переменной-отношения IXAYWA («if X attends, Y will attend» – если X придет, то и Y придет), содержащий по одному кортежу для каждого из имеющихся утверждений (см. следующий рисунок, на котором INV означает «invitee» – приглашенный). Отметим, что X и Y в этой переменной-отношении – RVA-атрибуты.



Но можно пойти дальше. Разумеется, я выбрал нотацию ФЗ не случайно. На самом деле, утверждение «если X придет, то и Y придет», очевидно, изоморфно утверждению «ФЗ $X \rightarrow Y$ удовлетворяет-

ся» (представьте себе переменную-отношение с булевыми атрибутами для Эми, Боба, Кэла и всех остальных; при этом каждый кортеж представляет битовую карту присутствия на вечеринке, совместимую с заданными условиями). Теперь мы можем воспользоваться теорией ФЗ, чтобы уменьшить объем информации, явно хранимой в этой переменной-отношении. С помощью теории мы сумеем найти так называемый *неприводимый эквивалент* заданного множества утверждений, то есть (обычно небольшое) множество утверждений, которое не содержит избыточной информации и обладает тем свойством, что все исходные утверждения следуют из утверждений, вошедших в это множество.

Подробное рассмотрение того, как искать неприводимый эквивалент, выходит за рамки настоящей книги, я лишь хочу описать набросок алгоритма. Во-первых, в приложении B я уже отмечал, что если $X \rightarrow Y$ удовлетворяется, то и $X' \rightarrow Y'$ удовлетворяется для всех надмножеств $X' \supseteq X$ и всех подмножеств $Y' \subseteq Y$. Поэтому мы можем сократить объем явно сохраняемой информации, проверяя для каждого записываемого утверждения $X \rightarrow Y$, что множество X – наименьшее из возможных, а множество Y – наибольшее из возможных. Точнее:

- Если хотя бы одного человека удалить из X, то результирующее утверждение перестанет быть верным.
- Если хотя бы одного человека добавить в Y, то результирующее утверждение перестанет быть верным.

Кроме того, мы знаем, что ФЗ $X \rightarrow Y$ обязательно удовлетворяется для всех множеств Y таких, что Y – подмножество X (в частности, зависимость вида $X \rightarrow X$ всегда удовлетворяется, так как означает, например, что «если Эми придет, то Эми придет»). Эти случаи тривиальны, поэтому их явно сохранять вообще не нужно.

Отметим также, что если переменная-отношение IXAYWA содержит только те ФЗ, которые составляют неприводимый эквивалент, то {X} будет ее ключом.

Желающих почитать продолжение отсылаю к своей книге «An Introduction to Database Systems» (см. приложение D).

Упражнение B.15. Самый простой проект выглядит примерно так:

```
EMP  { ENO , ENAME , SALARY }
     KEY { ENO }

PGMR { ENO , LANG }
     KEY { ENO }
     FOREIGN KEY { ENO } REFERENCES EMP
```

Каждый служащий представлен кортежем в EMP (и никаких других кортежей в EMP нет). Для служащих-программистов дополнитель-

но имеется кортеж в PGMR (и никаких других кортежей в PGMR нет). Отметим, что соединение EMP и PGMR дает всю информацию (номер служащего, имя, зарплату и владение языком) о программистах (и только о них).

Если программист может владеть несколькими языками, то нужно будет внести всего одно изменение – сделать ключом переменной-отношения совокупность всех атрибутов, то есть множество {ENO,LANG}.

Упражнение В.16. Назовем проекции R на $\{A,B\}$ и на $\{A,C\}$ *R1* и *R2* соответственно, и пусть (*a,b1,c1*) ∈ R и (*a,b2,c2*) ∈ R. Тогда (*a,b1*) ∈ *R1* и (*a,b2*) ∈ *R1*, а (*a,c1*) ∈ *R2* и (a,c2) ∈ *R2*; таким образом, (*a,b1,c2*) ∈ J и (*a,b2,c1*) ∈ J, где J = *R1* JOIN *R2*. Но R удовлетворяет ЗС *{*AB,AC*}, поэтому $J = R$; следовательно, (*a,b1,c2*) ∈ R и (*a,b2,c1*) ∈ R.

Кстати, отметим, что определение МЗЗ симметрично относительно B и C; следовательно, R удовлетворяет МЗЗ $A \rightarrow\!\!\!\!\rightarrow B$ тогда и только тогда, когда она удовлетворяет МЗЗ $A \rightarrow\!\!\!\!\rightarrow C$. МЗЗ всегда встречаются такими парами, поэтому принято записывать их одним предложением:

```
A →→ B | C
```

Упражнение В.17. Пусть C – атрибуты R, не вошедшие ни в A, ни в B. По теореме Хита, если R удовлетворяет ФЗ $A \rightarrow B$, то она удовлетворяет и ЗС *{*AB,AC*}. Однако по определению (см. упражнение В.16), если R удовлетворяет ЗС *{*AB,AC*}, то она удовлетворяет МЗЗ $A \rightarrow\!\!\!\!\rightarrow B$ и $A \rightarrow\!\!\!\!\rightarrow C$. Таким образом, из $A \rightarrow B$ следует $A \rightarrow\!\!\!\!\rightarrow B$.

Упражнение В.18. Результат немедленно следует из определения многозначной зависимости (см. упражнение В.16).

Упражнение В.19. Упражнение В.17 показывает, что если $K \rightarrow A$, то $K \rightarrow\!\!\!\!\rightarrow A$. Но если K – ключ, то $K \rightarrow A$, откуда сразу же следует требуемое утверждение.

Упражнение В.20. Пусть C – некоторый клуб, и пусть переменная-отношение $R\{A,B\}$ такова, что кортеж (*a,b*) встречается в R тогда и только тогда, когда и *a*, и *b* являются членами клуба C. Тогда R равно декартову произведению своих проекций $R\{A\}$ и R{*B*}, поэтому она удовлетворяет ЗС $K\{A,B\}$ и, что эквивалентно, следующим МЗЗ:

```
{ } →→ A | B
```

Эти МЗЗ не тривиальны, так как им, конечно, удовлетворяют не все бинарные переменные-отношения, и они не обусловлены суперключом. Отсюда следует, что R не находится в 4НФ. Однако она находится в НФБК, потому что ключом является совокупность всех атрибутов (см. упражнение В.24).

Упражнение В.21. Сначала введем три переменные-отношения с очевидной интерпретацией:

```
REP     { REPNO ,  ... } KEY { REPNO  }
AREA    { AREANO , ... } KEY { AREANO }
PRODUCT { PRODNO , ... } KEY { PRODNO }
```

Далее, связи между (а) торговыми представителями и регионами продаж и (б) торговыми представителями и изделиями можно представить такими переменными-отношениями:

```
RA { REPNO , AREANO } KEY { REPNO , AREANO }
RP { REPNO , PRODNO } KEY { REPNO , PRODNO }
```

Каждое изделие продается в каждом регионе. Поэтому вводим переменную-отношение

```
AP { AREANO , PRODNO } KEY { AREANO , PRODNO }
```

для представления связи между регионами и изделиями и следующее ограничение:

```
CONSTRAINT C1 AP = AREA { AREANO } JOIN PRODUCT { PRODNO } ;
```

Отметим, что из этого ограничения следует, что AP не находится в 4НФ. На самом деле, AP не несет информации, которую нельзя было бы получить из других переменных-отношений. Точнее, мы имеем:

```
AP { AREANO } = AREA { AREANO }
AP { PRODNO } = PRODUCT { PRODNO }
```

Но предположим ненадолго, что мы все-таки включили переменную-отношение AP в проект.

Никакие два представителя не продают одно и то же изделие в одном и том же регионе. Иными словами, для данной комбинации {AREANO, PRODNO} существует ровно один отвечающий за нее торговый представитель REPNO, поэтому можно ввести переменную-отношение

```
APR { AREANO , PRODNO , REPNO } KEY { AREANO , PRODNO }
```

в которой (чтобы сделать ФЗ явной)

```
{ AREANO , PRODNO } → { REPNO }
```

(Задания {AREANO,PRODNO} в качестве ключа достаточно для выражения этой ФЗ.) Теперь, однако, все переменные-отношения RA, RP и AP стали избыточными, потому что являются проекциями APR; следовательно, их можно удалить. Вместо ограничения C1 нам необходимо ограничение C2:

```
CONSTRAINT C2 APR { AREANO , PRODNO } =
              AREA { AREANO } JOIN PRODUCT { PRODNO } ;
```

Это ограничение должно быть сформулировано явно и независимо (оно не «обусловлено ключами»).

Также, поскольку каждый представитель продает все изделия, за которые отвечает, во всех подведомственных ему регионах, то мы имеем дополнительное ограничение С3 на переменную-отношение APR:

```
{ REPNO } →→ { AREANO } | { PRODNO }
```

(Эти МЗЗ не тривиальны, и переменная-отношение APR не находится в 4НФ.) Это ограничение также должно быть сформулировано явно и независимо.

Таким образом, окончательно проект состоит из переменных-отношений REP, AREA, PRODUCT и APR и ограничений С2 и С3:

```
CONSTRAINT C2 APR { AREANO , PRODNO } =
              AREA { AREANO } JOIN PRODUCT { PRODNO } ;

CONSTRAINT C3 APR = APR { REPNO , AREANO } JOIN
              APR { REPNO , PRODNO } ;
```

Это упражнение наглядно демонстрирует тот факт, что в общем случае нормализации может быть достаточно для представления некоторых семантических аспектов задачи (в основном, ФЗ, МЗЗ и ЗС, обусловленных ключами), но для представления других аспектов могут понадобиться дополнительные ограничения. Кроме того, из него видно, что нормализация «до конца» не всегда желательна (переменная-отношение APR находится в НФБК, но не в 4НФ).

Примечание

В качестве дополнительного упражнения можете подумать над тем, подходит ли для этой задачи структура с RVA-атрибутами. Означает ли такая структура, что некоторые из высказанных в предыдущем абзаце замечаний не применимы?

Упражнение В.22.

```
CONSTRAINT SPJ_JD SPJ = JOIN { SPJ { SNO , PNO } ,
                               SPJ { PNO , JNO } ,
                               SPJ { JNO , SNO } ;
```

Упражнение В.23. Это пример «циклического ограничения». Подойдет такая структура:

```
REP     { REPNO ,  ... } KEY { REPNO  }
AREA    { AREANO , ... } KEY { AREANO }
PRODUCT { PRODNO , ... } KEY { PRODNO }

RA { REPNO ,  AREANO } KEY { REPNO ,  AREANO }
AP { AREANO , PRODNO } KEY { AREANO , PRODNO }
PR { PRODNO , REPNO  } KEY { PRODNO , REPNO  }
```

Кроме того, пользователя необходимо проинформировать о том, что соединение RA, AP и PR не приводит к «ловушке связи»:

```
CONSTRAINT NO_TRAP
        ( RA JOIN AP JOIN PR ) { REPNO ,  AREANO } = RA AND
        ( RA JOIN AP JOIN PR ) { AREANO , PRODNO } = AP AND
        ( RA JOIN AP JOIN PR ) { PRODNO , REPNO  } = PR ;
```

Примечание

Как и в упражнении B.21, можете подумать над тем, подходит ли для этой задачи структура с RVA-атрибутами

Упражнение B.24. a. Верно. **b.** Верно. **c.** Неверно, хотя «почти» верно. Вот контрпример: пусть имеется переменная-отношение USA {COUNTRY, STATE} («STATE (штат) является частью COUNTRY (страны)»), где во всех кортежах COUNTRY содержит U.S. Эта переменная-отношение удовлетворяет ФЗ {} → {COUNTRY}, которая не является ни тривиальной, ни обусловленной ключом, поэтому данная переменная-отношение не находится в НФБК (она может быть без потерь разложена на две унарных проекции).

Упражнение B.25. Да, относятся. Какой вывод вы из этого сделаете?

Упражнение B.26. Я не собираюсь здесь рисовать никакие диаграммы «сущность-связь», а смысл упражнения в том, чтобы придать вес следующим замечаниям из приложения B:

> Проблема в том, что диаграммы «сущность-связь» и аналогичные им картинки в состоянии представить не все ограничения, а только небольшую часть, пусть и важную. Поэтому, хотя их использование и допустимо для того, чтобы объяснить проект на высоком уровне абстракции, думать, что на такой диаграмме структура представлена *во всей полноте* – заблуждение, иногда весьма опасное. *Напротив:* структура базы данных состоит из переменных-отношений, которые на диаграммах отражены, и *ограничений*, которые не отражены [курсив добавлен].

D

Дополнительная литература

Как следует из названия, в этом приложении содержится список литературы для дополнительного чтения. Приношу извинения за то, что в большинстве публикаций я сам являюсь автором или соавтором. Работы упорядочены по имени автора, а работы одного и того же автора – в хронологическом порядке по возрастанию. *Примечание*: В этой книге не рассматриваются конкретные продукты на основе SQL, поэтому и публикаций, относящихся к таким продуктам, вы здесь не найдете. Однако таких работ много, и, возможно, вы захотите познакомиться с некоторыми из них, чтобы применить обсуждавшиеся в этой книге идеи к какому-то конкретному продукту.

1. Surajit Chaudhuri and Gerhard Weikum «Rethinking Database System Architecture: Towards a Self-tuning RISC-style Database System» Proc. 26th Int. Conf. on Very Large Data Bases, Cairo, Egypt (September 2000).

   Среди прочего в этой работе убедительно отстаивается одна из ключевых идей настоящей книги, а именно (как я писал в предисловии): «SQL сложен, запутан, при написании SQL-команд легко допустить ошибку – рискну предположить, что куда легче, чем можно судить по уверениям апологетов этого языка.» Приведу пространную цитату из введения к статье:

   > *SQL – источник мучений.* Одна из самых неприятных проблем, присущих системе баз данных, – это язык SQL. Он представляет собой конгломерат самой разнообразной функциональности (многие его средства используются редко или должны быть поставлены под запрет) и слишком сложен для типичного разработчика приложений. Его основа, отнесем к ней выборку-проекцию-соединение и агрегирование, чрезвычайно полезна, но мы сомневаемся, что у многочисленных «бантиков» есть широкое и разумное применение. Разбираться в семантике SQL (даже до SQL-92), включающей в себя сочетание вложенных (и коррелированных)

подзапросов, null-значений, триггеров и т. д., – сущий кошмар. Преподаватели SQL обычно концентрируют внимание на ядре, а изучение всевозможных «прибамбасов» оставляют слушателям для «освоения на практике». В некоторых профессиональных журналах время от времени публикуются конкурсы знатоков SQL, в которых требуется выразить сложный запрос одним SQL-предложением. Такие предложения занимают несколько страниц и, чтобы понять их, требуются значительные усилия. Если программист придерживается такого стиля в реальных приложениях, то – с учетом неизбежных сложностей отладки «декларативного» предложения сверхвысокого уровня – становится очень трудно, если вообще возможно, удостовериться в том, что запрос адекватно отражает потребности пользователей. На самом деле, искусство программирования на SQL во многих случаях состоит в декомпозиции сложных запросов на последовательность более простых предложений.

2. E. F. Codd «Derivability, Redundancy, and Consistency of Relations Stored in Large Data Banks» *IBM Research Report* RJ599 (August 19th, 1969); «A Relational Model of Data for Large Shared Data Banks,» CACM 13, No. 6 (June 1970).

   Написанная в 1969 году статья Кодда была первой работой по реляционной модели и по существу явилась предтечей статьи 1970 года, с рядом очень любопытных отличий (главное из них состоит в том, что в 1969 году были разрешены атрибуты, значениями которых являются отношения, а в 1970 – уже нет). Эта статья 1970 года – перепечатанная в *Milestones of Research*, CACM 26, No. 1 (January 1982) и в других изданиях – стала первой широкодоступной работой на данную тему. Обычно ее называют первопроходческой работой в этой области, хотя это, пожалуй, несправедливо по отношению к предвосхитившей ее статье 1969 года. Я считаю, что каждый, кто профессионально занимается базами данных, должен ежегодно перечитывать хотя бы одну, а то и обе эти статьи.

3. E. F. Codd «Relational Completeness of Data Base Sublanguages» в сборнике Randall J. Rustin (ed.), *Data Base Systems, Courant Computer Science Symposia Series 6*. Englewood Cliffs, N.J.: Prentice Hall (1972).

   В этой работе Кодд впервые формально определил реляционную алгебру и реляционное исчисление в их первозданном виде. Чтение нелегкое, но вдумчивое изучение окупится сторицей.

4. E. F. Codd and C. J. Date «Much Ado about Nothing» в сборнике C. J. Date, *Relational Database Writings* 1991–1994. Reading, Mass.: Addison-Wesley (1995).

   Кодд был, пожалуй, самым яростным приверженцем null-значений и трехзначной логики, как основы для обработки отсутствующей информации (забавно, учитывая, что null-значения нарушают сформулированный тем же Коддом *принцип информации*). В этой статье

приводится спор между Коддом и мной на эту тему. Здесь же вы найдете восхитительное замечание: «Управление базами данных стало бы куда проще, не будь этих отсутствующих значений» (Кодд). *Примечание*: Я включил именно эту ссылку (из огромного числа публикаций на эту тему), потому что здесь так или иначе затрагиваются почти все доводы обеих сторон.

5. Hugh Darwen «The Role of Functional Dependence in Query Decomposition» в сборнике C. J. Date and Hugh Darwen, *Relational Database Writings* 1989–1991. Reading, Mass.: Addison-Wesley (1992).

   В этой работе приводятся правила вывода, применяя которые можно вывести функциональные зависимости (ФЗ), которым удовлетворяет отношение *r*, являющееся результатом вычисления произвольного реляционного выражения, зная, каким ФЗ удовлетворяют переменные-отношения, участвующие в этом выражении. Проанализировав множество выведенных таким образом ФЗ, можно определить, каким ограничениям ключа удовлетворяет *r*, что составляет основу правил вывода ключа, которые мимоходом упоминались в главе 4.

6. Hugh Darwen «What a Database *Really* Is: Predicates and Propositions» в сборнике C. J. Date, *Relational Database Writings* 1994–1997. Reading, Mass.: Addison-Wesley (1998).

   Очень внятное пособие по предикатам переменных-отношений и смежным вопросам.

7. Hugh Darwen «How to Handle Missing Information Without Using Nulls» (слайды к презентации), *http://www.thethirdmanifesto.com* (May 9th, 2003; revised May 16th, 2005).

   Один из возможных подходов к проблеме, обозначенной в заголовке («Как обрабатывать отсутствие информации, не используя null»).

8. C. J. Date «Fifty Ways to Quote Your Query», *http://www.dbpd.com* (July 1998).

   На эту статью есть ссылка в главе 6.

9. C. J. Date «Composite Keys» в сборнике C. J. Date and Hugh Darwen, *Relational Database Writings* 1989-1991. Reading, Mass.: Addison-Wesley (1992).

   На эту статью есть ссылка в приложении B.

10. C. J. Date «An Introduction to Database Systems» (8th edition). Boston, Mass.: Addison-Wesley (2004).

    Учебник по всем аспектам управления базами данных для студентов младших курсов. SQL обсуждается на уровне стандарта SQL:1999 с некоторыми замечаниями, относящимися к SQL:2003; в частности, включено рассмотрение «объектно-реляционных» механизмов (типы REF, ссылочные значения и т. д.), и объясняется, почему они

нарушают реляционные принципы. *Примечание:* Эта тематика рассматривается также в работах [32], [41] и [42].

11. C. J. Date «The Relational Database Dictionary», Extended Edition. Berkeley, Calif.: Apress (2008).

    Многие определения в тексте книги основаны на приведенных в этой работе.

12. C. J. Date «Double Trouble, Double Trouble», в сборнике *Date on Database: Writings 2000–2006*. Berkeley, Calif.: Apress (2006).

    Всестороннее и подробное рассмотрение проблем, вызванных дубликатами. Обсуждение дубликатов в главе 4 основано на примере, взятом из этой работы.

13. C. J. Date «What First Normal Form Really Means», в сборнике *Date on Database: Writings 2000–2006*. Berkeley, Calif.: Apress (2006).

    На протяжении многих лет первая нормальная форма была источником многочисленных заблуждений. В этой статье сделана попытка исправить ситуацию – и даже служить авторитетным источником, насколько это возможно. В основу аргументации положена идея о том (см. главу 2), что понятие атомарности (в терминах которой и была первоначально определена первая нормальная форма) лишено всякого смысла.

14. C. J. Date «A Sweet Disorder», в сборнике *Date on Database: Writings 2000–2006*. Berkeley, Calif.: Apress (2006).

    Атрибуты отношений не упорядочены слева направо, но в SQL-таблицах такое упорядочение столбцов подразумевается. В этой статье исследуются последствия такого положения вещей, которые оказываются куда менее тривиальными, чем могло бы показаться.

15. C. J. Date «On the Notion of Logical Difference», «On the Logical Difference Between Model and Implementation» и «On the Logical Differences Between Types, Values, and Variables», все в сборнике *Date on Database: Writings 2000–2006*. Berkeley, Calif.: Apress (2006).

    Заголовками все сказано («О понятии логического различия», «О логическом различии между моделью и реализацией», «О логических различиях между типами, значениями и переменными»).

16. C. J. Date «Two Remarks on SQL's UNION» в сборнике *Date on Database: Writings 2000–2006*. Berkeley, Calif.: Apress (2006).

    В этой короткой статье описываются некоторые странности, возникающие в SQL-операторе UNION – и, как следствие, в операторах INTERSECT и EXCEPT – из-за (а) приведения типов и (б) строк-дубликатов.

17. C. J. Date «A Cure for Madness» в сборнике *Date on Database: Writings 2000–2006*. Berkeley, Calif.: Apress (2006).

Детальное исследование того факта, что – вопреки интуиции – SQL-выражения SELECT *нечто* FROM (SELECT * FROM *t* WHERE *p*) WHERE *q* и SELECT *нечто* FROM *t* WHERE *p* AND *q* логически не эквивалентны, хотя должны бы быть эквивалентны, и по крайней мере один современный продукт на основе SQL иногда трансформирует первое во второе.

18. C. J. Date «Why Three- and Four-Valued Logic Don't Work» в сборнике *Date on Database: Writings 2000–2006*. Berkeley, Calif.: Apress (2006).

    В книге отмечалось, что поддержка null в SQL базируется на трехзначной логике. Реализация этой логики в SQL страдает серьезными дефектами, но даже если бы их не было, использовать трехзначную логику все равно не рекомендуется, и в этой статье объясняется, почему.

19. C. J. Date «The Logic of View Updating» в сборнике *Logic and Databases: The Roots of Relational Theory*. Victoria, BC: Trafford Publishing (2007). См. *http://www.trafford.com/07-0690*.

    В этой работе, а также в пространном приложении на ту же тему в работе [28] приведены доводы в поддержку мысли о том, что представления логически всегда допускают обновления, а препятствием служат только нарушения *принципа присваивания* или **золотого правила**.

20. C. J. Date «The Closed World Assumption» в сборнике *Logic and Databases: The Roots of Relational Theory. Victoria*, BC: Trafford Publishing (2007). См. *http://www.trafford.com/07-0690*.

    На *допущении замкнутости мира* редко акцентируют внимание, а ведь оно лежит в основе почти всего, что мы делаем при использовании базы данных. В этой работе указанное допущение рассматривается подробно; в частности, показано, почему его следует предпочесть *допущению открытости мира* (на котором, кстати, основана «семантическая паутина», по крайней мере, так было заявлено). Здесь же объясняется, почему мы все-таки можем получать ответы «я не знаю» (если это необходимо) даже от базы данных, в которой нет null-значений.

21. C. J. Date «The Theory of Bags: An Investigative Tutorial» в сборнике *Logic and Databases: The Roots of Relational Theory*. Victoria, BC: Trafford Publishing (2007). См. *http://www.trafford.com/07-0690*.

    Среди прочего, в этой статье обсуждается, что происходит с операторами типа объединения, когда операндами являются не множества, а мультимножества.

22. C. J. Date «Inclusion Dependencies and Foreign Keys» (готовится к печати).

    Альтернативным названием этой статьи могло быть быть *Rethinking Foreign Keys* (Переосмысливая внешние ключи); среди прочего здесь

показано, что понятие внешнего ключа охватывает гораздо больше, чем это обычно представляют. Также обсуждаются логические различия между внешними ключами и указателями. (В главе 2 отмечалось, что некоторые авторы заявляют, будто внешние ключи – не более чем традиционные указатели «в овечьей шкуре», но это совсем не так.)

23. C. J. Date «Image Relations» (готовится к печати).

    Подробное обсуждение семантики и полезности отношений-образов (см. главу 7).

24. C. J. Date «Is SQL's Three-Valued Logic Truth Functionally Complete?» (готовится к печати).

    Среди прочего статья содержит полное и точное описание поддержки null-значений и трехзначной логики в SQL.

25. C. J. Date «A Remark on Prenex Normal Form» (готовится к печати).

    Обсуждение предваренной нормальной формы в контексте баз данных.

26. C. J. Date «Go Faster! The TransRelational™ Approach to DBMS Implementation» (готовится к печати).

    Подробное описание *The TransRelational™ Model* – новаторской технологии реализации, упомянутой в приложениях A и B. *Примечание*: Краткое введение в эту технологию можно найти также в приложении A к работе [10].

27. C. J. Date and Hugh Darwen «A Guide to the SQL Standard» (4th edition). Reading, Mass.: Addison-Wesley (1997).

    В этой книге полностью описан стандарт SQL по состоянию на 1997 год. С тех пор в стандарт было добавлено немало нового (в том числе так называемые объектно-реляционные средства (см. работу [28])), но к реляционному использованию SQL они имеют мало отношения. Поэтому на мой, несколько предвзятый взгляд, эта книга остается неплохим источником подробной информации практически обо всех аспектах SQL (по крайней мере, в стандартном воплощении), затронутых в настоящей книге.

28. C. J. Date and Hugh Darwen «Databases, Types, and the Relational Model: The Third Manifesto» (3rd edition). Boston, Mass.: Addison-Wesley (2006).

    В этой книге формулируется и объясняется *Третий Манифест* – детально проработанное предложение о будущем систем управлениями данными и базами данных. Включено также точное, хотя и несколько формальное, определение реляционной модели и подробное предложение, касающееся необходимой поддержки теории типов (в том числе и исчерпывающая модель наследования типов), а, кроме того, полное описание языка **Tutorial D**.

29. C. J. Date and Hugh Darwen «Multiple Assignment» в сборнике *Date on Database: Writings 2000–2006*. Berkeley, Calif.: Apress (2006).

См. главу 8.

30. C. J. Date, Hugh Darwen, and Nikos A. Lorentzos «Temporal Data and the Relational Model». San Francisco, Calif.: Morgan Kaufmann (2003).

Некоторое представление о том, чему посвящена эта книга, можно найти в приложении A.

31. C. J. Date and David McGoveran «Why Relational DBMS Logic Must Not Be Many-Valued,» в сборнике C. J. Date, *Logic and Databases: The Roots of Relational Theory. Victoria*, BC: Trafford Publishing (2007). See *http://www.trafford.com/07-0690*.

В этой статье аргументируется положение о том, что языки баз данных должны быть основаны на двузначной логике (как реляционная модель, а не как SQL).

32. Ramez Elmasri and Shamkant Navathe «Fundamentals of Database Systems» (4th edition). Boston, Mass.: Addison-Wesley (2004).

33. Stéphane Faroult with Peter Robson «The Art of SQL». Sebastopol, Calif.: O'Reilly Media Inc. (2006).

Практическое руководство на тему о том, как добиваться высокой производительности от SQL в современных продуктах. Некоторое представление о степени охвата материала можно получить из следующего, немного отредактированного, списка подзаголовков из двенадцати глав книги:

1. Проектирование баз данных с учетом производительности
2. Эффективный доступ к базе данных
3. Индексирование
4. Предложения SQL
5. Физическая реализация
6. Классические способы применения SQL
7. Иерархические данные
8. Трудные случаи
9. Конкурентный доступ
10. Большие объемы данных
11. Время реакции
12. Мониторинг производительности

В своих советах и рекомендациях авторы не сильно отклоняются от реляционных принципов и даже, по большей части, явно выступают в поддержку этих принципов. Но они также понимают, что со-

временные оптимизаторы далеки от идеала, а потому подсказывают, как из многих логически эквивалентных формулировок выбирать для конкретной задачи ту, которая будет работать быстрее (и объясняют почему). Описываются также некоторые приемы кодирования, способствующие повышению производительности, например, использование MIN для того, чтобы убедиться, что в столбце, допускающем только значения да/нет, все значения равны *да* (вместо того, чтобы явно проверять на *нет*). Говоря о советах[1] (*hint*) оптимизатору (многие продукты поддерживают такую возможность), авторы пишут следующие мудрые слова: «Проблема советов в том, что они более императивны, чем можно предположить по названию, и любой совет – азартная игра с будущим. Нет никаких гарантий, что в результате изменения обстоятельств, объемов, алгоритмов СУБД, аппаратуры и т. д. навязанный план выполнения останется не то, что абсолютно лучшим, но хотя бы приемлемым... Помните, что нужно тщательно документировать все, что вы силой навязываете СУБД».

34. Patrick Hall, Peter Hitchcock, and Stephen Todd «An Algebra of Relations for Machine Computation,» Conf. Record of the 2nd ACM Symposium on Principles of Programming Languages, Palo Alto, Calif. (January 1975).

    Эта работа, пожалуй немного «сложновата», но я думаю, что она важна. И язык **Tutorial D**, и та версия реляционной алгебры, которую я описал в настоящей книге, берут начало в этой статье.

35. Jim Gray and Andreas Reuter «Transaction Processing: Concepts and Techniques». San Mateo, Calif.: Morgan Kaufmann (1993).

    Стандартный учебник по управлению транзакциями.

36. Lex de Haan and Toon Koppelaars «Applied Mathematics for Database Professionals». Berkeley, Calif.: Apress (2007).

    Среди прочего вы найдете в этой книге обширный набор тождеств (здесь они называются *правилами перезаписи*), который можно использовать, как это было сделано в главе 11, для формулирования сложных SQL-выражений. Здесь же показано, как реализовывать ограничения целостности с помощью процедурного кода (см. главу 8). Рекомендую.

37. Wilfrid Hodges «Logic». London, England: Penguin Books (1977).

    Введение в формальную логику для начинающих.

38. International Organization for Standardization (ISO) Database Language SQL, Document ISO/IEC 9075:2003 (2003).

---

[1] Обычно *optimizer hint* переводят как *указание оптимизатору* (что, кстати, точнее отражает их природу), но в данном случае контекст требует перевести более мягко и ближе к смыслу английского слова (*намек*). – *Прим. перев.*

Официальный стандарт SQL. Отмечу, что это действительно международный стандарт, а не просто (как многие, похоже, думают) американский или «ANSI».

39. Jim Melton and Alan R. Simon «SQL:1999 – Understanding Relational Components»; Jim Melton «Advanced SQL:1999 – Understanding Object-Relational and Other Advanced Features». San Francisco, Calif.: Morgan Kaufmann (2002 and 2003, respectively).

    Это единственные известные мне книги, в которых рассмотрены все аспекты стандарта SQL:1999 (непосредственного предшественника текущей версии стандарта SQL:2003). Мелтон в течение многих лет был редактором стандарта SQL.

40. Fabian Pascal «Practical Issues in Database Management: A Reference for the Thinking Practitioner». Boston, Mass.: Addison-Wesley (2000).

    В некоторых отношениях эту книгу можно считать дополнением к той, что вы держите в руках. Здесь рассматриваются многие типичные вопросы баз данных: нормализация, избыточность, целостность, отсутствующая информация и другие, и обсуждаются теоретически правильные подходы к их решению, с упором на практическое применение теории.

41. Raghu Ramakrishnan and Johannes Gehrke «Database Management Systems» (3rd edition). New York, N.Y.: McGraw-Hill (2003).

42. Avi Silberschatz, Henry F. Korth, and S. Sudarshan «Database System Concepts» (5th edition). New York, N.Y.: McGraw-Hill (2005).

43. Robert R. Stoll «Sets, Logic, and Axiomatic Theories». San Francisco, Calif.: W. H. Freeman and Company (1961).

    Корни реляционной модели глубоко уходят в формальную логику и теорию множеств. Эта книга является достаточно формальным, но не слишком трудным введением в эти предметы. См. также книгу Hodges [37].

44. Dave Voorhis Rel. *http://db@builder.sourceforge.net/rel.html*.

    Отсюда можно скачать код эталонной реализации диалекта **Tutorial D**.

45. Moshé M. Zloof «Query-By-Example», Proc. NCC 44, Anaheim, Calif. (May 1975). Montvale, N.J.: AFIPS Press (1977).

    Query-By-Example (QBE) – отличная иллюстрация того факта, что можно создать очень «дружественный пользователю» язык, основанный на реляционном исчислении, а не на реляционной алгебре. Злуф придумал и спроектировал язык QBE, и эта статья – одна из многих его работ на эту тему.

Алфавитный указатель