

ПРАКТИЧЕСКОЕ РУКОВОДСТВО ПО SQL

Эта книга поможет даже в тех случаях, когда бесполезно самое лучшее руководство пользователя. Здесь раскрываются темы, которые часто пропускаются или очень кратко описаны в стандартных руководствах пользователя — структуры баз данных, индексация, подзапросы, виртуальные таблицы, производительность и целостность данных.

Вы шаг за шагом изучите основы SQL и научитесь с помощью этого языка создавать приложения для работы с базами данных.

Книга предназначена для пользователей реляционных баз данных — независимо от того, работают ли они на больших многопользовательских компьютерных системах или на персональных компьютерах.

Содержание

Предисловие	5
Предисловие ко второму и третьему изданиям	7
Введение	9
Глава 1. SQL и управление реляционными базами данных	15
УПРАВЛЕНИЕ РЕЛЯЦИОННЫМИ БАЗАМИ ДАННЫХ	15
Реляционная модель: одни таблицы	16
Независимость	17
Язык высокого уровня	17
Реляционные операции	19
Альтернативный способ просмотра данных	22
Нули	23
Безопасность	23
Целостность	24
ПРИСТУПАЯ К ПРОЕКТИРОВАНИЮ БАЗЫ ДАННЫХ	24
Глава 2. Проектирование баз данных	25
СТРУКТУРА БАЗЫ ДАННЫХ	25
Как подходить к проектированию базы данных	26
Что такое "хорошая структура"	28
Описание нашей базы данных	29
ДАННЫЕ И ВЗАИМОСВЯЗИ	30
Объекты	30
Отношение один-ко-многим	32
Отношение многих-ко-многим	34
Отношение один-к-одному	35
Последние замечания к объектному подходу	35
РУКОВОДСТВО ПО НОРМАЛИЗАЦИИ	36
Первая нормальная форма	37
Вторая нормальная форма	38
Третья нормальная форма	38
Четвертая и пятая нормальные формы	40
ОБЗОР БАЗЫ ДАННЫХ	41

Последние замечания о базе данных bookbiz	42
Проверка структуры базы данных	44
Рассмотрение других понятий из области баз данных	44
Реализация структуры	44
Глава 3. Создание и заполнение базы данных	45
СИНТАКСИС SQL	45
Обработка ошибок	47
СОЗДАНИЕ БАЗ ДАННЫХ	48
Выбор базы данных	49
СОЗДАНИЕ ТАБЛИЦ	49
Выбор типа данных	51
Назначение нулевого статуса	53
Процесс создания таблицы	54
СОЗДАНИЕ ИНДЕКСОВ	55
Оператор CREATE INDEX	55
Как, что и зачем нужно индексировать	57
СОЗДАНИЕ ТАБЛИЦ С ПОМОЩЬЮ ОГРАНИЧЕНИЙ SQL-92	58
ИЗМЕНЕНИЕ И УДАЛЕНИЕ БАЗ ДАННЫХ И ИХ ОБЪЕКТОВ	62
Изменение баз данных	62
Изменение определений таблицы	62
Удаление базы данных	63
Удаление таблиц	63
Удаление индекса	63
ДОБАВЛЕНИЕ, ИЗМЕНЕНИЕ И УДАЛЕНИЕ ДАННЫХ	64
Добавление новой строки	64
Использование оператора SELECT в команде INSERT	66
ИЗМЕНЕНИЕ СУЩЕСТВУЮЩИХ ДАННЫХ	68
Оператор UPDATE	68
Предложение SET	68
Предложение WHERE	69
УДАЛЕНИЕ ДАННЫХ: КОМАНДА DELETE	70
ПРИСТУПАЯ К ВЫБОРКЕ ДАННЫХ	71
Глава 4. Выборка информации из базы данных	72
ПЕРЕД ВЫБОРОМ	72
Синтаксис оператора SELECT	72
ВЫБОР СТОЛБЦОВ: СПИСОК ВЫБОРА	75
Выбор всех столбцов: SELECT *	75
Выбор отдельных столбцов	77
Выражения: больше, чем просто имена столбцов	77
УКАЗАНИЕ ТАБЛИЦ: СПИСОК ТАБЛИЦ	83
ВЫБОР СТРОК: ПРЕДЛОЖЕНИЕ WHERE	84
Операторы сравнения	84
Совместное использование условных и логических операторов	86
Диапазоны (BETWEEN и NOT BETWEEN)	90

Списки (IN и NOT IN)	92
Выборка нулевых значений	94
Поиск по подстрокам: предложение LIKE	96
ЧТО ДАЛЬШЕ	99
Глава 5. Сортировка данных и другие методы выбора	100
ДОПОЛНИТЕЛЬНЫЕ ВОЗМОЖНОСТИ ОПЕРАТОРА SELECT	100
СОРТИРОВКА РЕЗУЛЬТАТОВ ЗАПРОСА: ПРЕДЛОЖЕНИЕ ORDER BY	100
Порядок сортировки	100
Как выполняется сортировка	101
Синтаксис предложения ORDER BY	102
Сортировка внутри сортировки	102
Сортировка по возрастанию и по убыванию	104
А как насчет выражений?	105
Как сортировать нулевые значения	107
УСТРАНЕНИЕ ПОВТОРЯЮЩИХСЯ СТРОК: ПРЕДЛОЖЕНИЯ DISTINCT И ALL	108
Синтаксис предложения DISTINCT	109
Почувствуйте разницу!	109
АГРЕГИРУЮЩИЕ ФУНКЦИИ	112
Синтаксис агрегирующих функций	114
СКАЛЯРНЫЕ И ВЕКТОРНЫЕ ФУНКЦИИ	120
Глава 6. Группировка данных и построение отчетов	121
ГРУППИРОВКА	121
ПРЕДЛОЖЕНИЕ GROUP BY	121
Синтаксис предложения GROUP BY	122
Упорядоченные группы	131
ПРЕДЛОЖЕНИЕ HAVING	132
Разновидности предложения HAVING	132
Предложения HAVING и WHERE	133
ЕЩЕ О НУЛЕВЫХ ЗНАЧЕНИЯХ	135
Нули и проектирование баз данных	136
Сравнение нулевых значений	136
Нули и вычисления	137
Нули и группы	138
Значения по умолчанию в качестве альтернативы нулевым значениям	138
РАБОТА С НЕСКОЛЬКИМИ ТАБЛИЦАМИ	140
Глава 7. Объединение таблиц и сложный анализ данных	141
ЧТО ТАКОЕ ОБЪЕДИНЕНИЕ	141
Синтаксис операции объединения	141
ПОЧЕМУ НЕОБХОДИМО ОБЪЕДИНЕНИЕ	142
Объединения и реляционная модель	142
ПРИМЕР ОБЪЕДИНЕНИЯ	143
Проверка правильности объединения	143

КАК ПОЛУЧИТЬ ХОРОШЕЕ ОБЪЕДИНЕНИЕ	144
Объединения и нулевые значения	144
УЛУЧШЕНИЕ ЧИТАЕМОСТИ РЕЗУЛЬТАТОВ ОБЪЕДИНЕНИЯ	144
Выбор столбцов для запросов на объединение	145
Псевдонимы в списке таблиц улучшают читаемость запросов	146
ОПРЕДЕЛЕНИЕ УСЛОВИЙ ОБЪЕДИНЕНИЯ	146
Объединения, основанные на равенстве	146
Объединения, не основанные на равенствах	147
Объединение таблицы с самой собой: самообъединение	148
Использование при самообъединении оператора неравенства	150
Объединение нескольких таблиц	151
Внешние объединения	152
КАК ОБЪЕДИНЕНИЯ ОБРАБАТЫВАЮТСЯ СИСТЕМОЙ	154
ОПЕРАТОР UNION	155
Полезный трюк с оператором UNION	157
ПОДЗАПРОСЫ	158
Глава 8. Структурированные запросы и подзапросы	159
ЧТО ТАКОЕ ПОДЗАПРОС	159
Упрощенный синтаксис подзапроса	159
КАК РАБОТАЮТ ПОДЗАПРОСЫ	160
Некоррелированная обработка	161
Коррелированная обработка	162
ОБЪЕДИНЕНИЯ ИЛИ ПОДЗАПРОСЫ?	162
Подзапросы!	162
Объединения!	164
Подзапросы или самообъединения?	164
Что лучше?	165
ПРАВИЛА ПОДЗАПРОСОВ	165
ПОДЗАПРОСЫ, НЕ ВОЗВРАЩАЮЩИЕ ЗНАЧЕНИЙ ИЛИ ВОЗВРАЩАЮЩИЕ НЕСКОЛЬКО ЗНАЧЕНИЙ	166
Подзапросы, начинающиеся с IN	166
Подзапросы, начинающиеся с NOT IN	167
Коррелированные подзапросы с IN	168
Подзапросы, начинающиеся с операторов сравнения и включающие ключевые слова ANY или ALL	170
ПОДЗАПРОСЫ, ВОЗВРАЩАЮЩИЕ ЕДИНСТВЕННОЕ ЗНАЧЕНИЕ	174
Агрегирующие функции гарантируют единственное значение	175
Предложения GROUP BY и HAVING должны возвращать единственное значение	176
Коррелированные подзапросы с операторами сравнения	176
ПОДЗАПРОСЫ, ВЫПОЛНЯЮЩИЕ ПРОВЕРКУ НА СУЩЕСТВОВАНИЕ	177
NOT EXISTS отыскивает пустой набор	179
Использование EXISTS для поиска пересечения и разности	180

ПОДЗАПРОСЫ С РАЗНЫМИ УРОВНЯМИ ВЛОЖЕНИЯ	181
ПОДЗАПРОСЫ В ОПЕРАТОРАХ UPDATE, DELETE И INSERT	181
В ПОЛЕ ЗРЕНИЯ КУРСОРА	182
Глава 9. Создание и использование виртуальных таблиц (курсоров)	183
КУРСОР ОБЕСПЕЧИВАЕТ ГИБКОСТЬ	183
СОЗДАНИЕ КУРСОРОВ	183
Удаление курсоров	184
ПРЕИМУЩЕСТВА КУРСОРОВ	184
Почему же все-таки курсор?	187
КАК РАБОТАЮТ КУРСОРЫ	189
Правила присвоения имен столбцам курсора	190
Создание курсоров с объединениями и подзапросами	191
Ограничения на создание курсоров	192
Предложение Check Option	192
Разборка курсора	194
Переопределение курсоров	194
МОДИФИКАЦИЯ ДАННЫХ ПОСРЕДСТВОМ КУРСОРОВ	196
Правила в соответствии с ANSI	196
СОЗДАНИЕ КОПИЙ ДАННЫХ	198
ВОПРОСЫ АДМИНИСТРИРОВАНИЯ БАЗ ДАННЫХ	199
Глава 10. Безопасность, транзакции, производительность и целостность	200
УПРАВЛЕНИЕ БАЗАМИ ДАННЫХ В РЕАЛЬНОМ МИРЕ	200
БЕЗОПАСНОСТЬ ДАННЫХ	201
Идентификация пользователя и особые пользователи	201
Команды GRANT и REVOKE	202
Курсоры как механизм обеспечения безопасности	206
ТРАНЗАКЦИИ	207
Транзакции и совпадения	208
Транзакции и восстановление	209
Транзакции, определяемые пользователем	209
Получение резервной копии и восстановление	210
ПРОИЗВОДИТЕЛЬНОСТЬ	211
Сравнение с эталоном	211
Структура и индексация	212
Запросы	213
Другие инструменты для мониторинга и повышения производительности	213
ЦЕЛОСТНОСТЬ ДАННЫХ	215
Ограничения на домен	215
Целостность объекта	216
Ссылочная целостность	217
ОТ АБСТРАКЦИЙ SQL К РЕАЛЬНОМУ МИРУ	220
Глава 11. Разрешение проблем	221
КАК ИСПОЛЬЗОВАТЬ SQL В СВОЕЙ РАБОТЕ	221

ФОРМАТИРОВАНИЕ И ОТОБРАЖЕНИЕ ДАННЫХ	222
Отображение одного поля в виде двух	222
Выравнивание строки символов по правому краю	224
Как указать число разрядов после десятичной точки	227
РАБОТА С ШАБЛОНАМИ	229
Сопоставление прописных и строчных букв	230
Поиск символьных данных заданного размера	231
Как найти данные типа дат	232
Замена пробелов на нули	234
ПОИСК ДАННЫХ С ПОМОЩЬЮ СЛОЖНЫХ ОБЪЕДИНЕНИЙ И ПОДЗАПРОСОВ	236
Сопоставление пар столбцов в разных таблицах	236
Поиск данных в определенном диапазоне, если вам не известны точные значения	238
Отображение данных в формате электронной таблицы	239
ПРЕДЛОЖЕНИЕ GROUP BY	243
Отображение данных по времени	243
ПОСЛЕДОВАТЕЛЬНЫЕ НОМЕРА	244
Нахождение максимального значения и добавление 1	245
Использование отдельной таблицы ключей	246
Использование произвольного значения	247
КАК ИЗБЕЖАТЬ ОШИБОК	247
Глава 12. Ошибки, и как их избежать	248
НЕТ, ВЫ НЕ ДУРАК	248
ПРЕДЛОЖЕНИЕ GROUP BY	248
Подсчет по единицам	249
ПРЕДЛОЖЕНИЯ WHERE И HAVING	249
Почему столько строк?	249
Сочетание значений строк и агрегирующих функций	253
Как избежать проблем с предложением HAVING	256
КЛЮЧЕВОЕ СЛОВО DISTINCT	258
DISTINCT со столбцами и выражениями	259
DISTINCT с агрегирующими функциями	260
DISTINCT и DISTINCT?	261
ДРУГИЕ НЕДОРАЗУМЕНИЯ	262
Удаление дубликатов	262
Нахождение "первого" входа	263
Приложение А. Краткое описание синтаксиса SQL, используемого в книге	265
СОГЛАШЕНИЯ ПО СИНТАКСИСУ	265
ФОРМАТИРОВАНИЕ	265
Регистр	265
СПИСОК ОПЕРАТОРОВ	266
Приложение Б. Аналогии между ключевыми словами разных	267

диалектов SQL

СРАВНЕНИЕ СИНТАКСИСОВ	267
ОПРЕДЕЛЕНИЕ ДАННЫХ	267
Операторы базы данных	268
Создание и удаление объектов базы данных	269
МАНИПУЛЯЦИИ С ДАННЫМИ	271
АДМИНИСТРИРОВАНИЕ ДАННЫХ	273
Приложение В. Словарь терминов	276
Приложение Г. Описание базы данных bookbiz	284
ТАБЛИЦЫ	284
ОПЕРАТОРЫ CREATE И INSERT ДЛЯ БАЗЫ ДАННЫХ BOOKBIZ	290
Приложение Д. Список литературы	309
Предметный указатель	311

Предметный указатель

А	домен, 215
агрегирующая функция, 100	Е
администратор базы данных, 23	естественное объединение, 147
администрирование данных, 17	З
аргумент, 114	запись, 16
атрибут, 16	запрос, 18
Б	значение, 16
база данных, 16	И
базовая таблица, 20; 183	идентификатор, 45
блокировка, 208	индекс, 17
В	К
виртуальная таблица, 22	команда, 17
включающий диапазон, 90	команды управления данными, 19
владелец, 23	контроль совпадений, 200
вложенная сортировка, 103	кортеж, 16
вложенный запрос, 72	курсор, 22
внешнее объединение, 146	кэш данных, 213
внешний ключ, 28	Л
восстановление, 200	логическая независимость, 17
вспомогательная таблица, 37	логические операторы, 86
выборка данных, 17	М
выражение, 67	моделирование зависимостей, 26
Г	модификация данных, 17
главная таблица, 37	Н
групповой индекс, 56	набор символов, 100
Д	назначение полномочий, 200
декартово произведение, 143	немодифицированный оператор
декомпозиция без потерь, 27	сравнения, 174
диаграмма зависимостей между	непроцедурный язык
объектами, 27	программирования, 19

нормализация, 26
нормальная форма, 36
О
общий подязык данных, 17
объект, 16
объектная целостность, 24
ограничение, 19
оператор, 17
определение данных, 17
оптимизатор запросов, 213
отношение, 16
П
первичный ключ, 16
подзапрос, 159
поисковая таблица, 42
поле, 16
пользовательская таблица, 16
порядок сортировки, 100
права на доступ и модификацию
данных, 23
правило, 58
проектирование базы данных, 25
производная таблица, 20
просматриваемая таблица, 183
псевдоним, 83
Р
различаемый нуль, 136
С
сгруппированный курсор, 197
системная таблица, 16
системный администратор, 23
системный журнал транзакций, 209
системный каталог, 16
сканирование таблицы, 58

составной индекс, 56
список выбора, 75
список таблиц, 83
сравнение с эталоном, 211
ссылочная целостность, 24
столбец, 16
столбец соединения, 141
стратегия доступа, 17
строка, 16
структура данных, 27
сущность, 16
схема, 48
Т
таблица, 16
терминатор, 47
транзакция, 207
триггерные действия, 218
триггерные условия, 218
У
уникальный индекс, 56
управление транзакциями, 24; 207
устройство базы данных, 48
Ф
файл, 16
физическая независимость данных,
17
фиктивное значение, 67
форма, 64
Ц
целостность, 24
Ш
шаблон, 96
Э
экземпляр, 31

Предисловие

Язык SQL вырос из языков баз данных, известных только компьютерным специалистам, в широко используемый мировой стандарт индустрии ПК. Число SQL-92-совместимых баз данных увеличивается с каждым годом и сегодня, наверное, достигло миллиона. Если вы получаете какую-либо информацию из Internet или из внутренних сетей, то почти с полной уверенностью можно сказать, что в этом принимает участие SQL. Однако то, что сегодня кажется столь очевидным, было таким далеко не всегда.

В начале эры реляционных баз данных, когда SQL только разрабатывался в исследовательской лаборатории IBM, я участвовал в создании языка управления базами данных, который должен был стать конкурентом SQL. Наш язык, в отличие от SQL, мог обрабатывать более сложные запросы, но был труднее в изучении. Спустя семь лет, в течение которых на основе нашего языка были разработаны два готовых программных продукта, я стал полным сторонником языка SQL.

Причина того тривиальна: язык SQL достаточно мощен и прост в изучении, постоянно улучшается и, что наиболее важно, поддерживается всеми поставщиками систем управления базами данных. Именно поэтому свою третью программу (для Sybase) я уже написал на языке SQL.

Не следует также забывать, что предыдущее поколение баз данных было нереляционным и каждая база данных использовала свой собственный язык. Поэтому разработка приложений для них требовала больших финансовых расходов, и переход на новую базу данных всегда был суровым испытанием и для пользователей, и для программистов.

Сегодня все существующие системы управления базами данных (СУБД) поддерживают язык SQL. Кроме того, с помощью специальных трансляторов SQL, которые соединяют данные из разных баз данных, можно получить доступ и к устаревшим базам данных. В системах типа клиент/сервер доступ к базам данных также осуществляется на основе SQL. Поэтому знание языка SQL сегодня просто необходимо, независимо от того, какую СУБД вы используете.

Принимая во внимание академическое происхождение SQL, можно понять, почему для многих людей SQL и реляционные базы данных являются чем-то непостижимым. Термины типа “нормальные формы”, “корреляционные переменные” и “ссылочная целостность” совершенно не проясняют заложенный в них абсолютно простой смысл. Поэтому процесс управления базой данных может показаться чем-то из области высшей математики, доступным только высокоученым специалистам.

Несмотря на ряд заложенных в его основу упрощений, SQL — очень мощный язык. Эта мощность иногда может привести к результатам, весьма неожиданным даже для профессионалов. Семь лет назад, например, я принимал участие в сравнительном тестировании ряда СУБД различных производителей. Все проходило нормально, но однажды мы в течение шести часов ожидали результата SQL-запроса, не-SQL-аналог которого на других системах выполнялся всего за несколько минут.

Мы терялись в догадках и не могли понять, как такое вообще может быть, пока кто-то не предложил нам сформулировать свой запрос к базе данных на обычном “человеческом” языке. В результате до нас дошло, что правая скобка в запросе стоит не на том месте. После исправления запрос выполнялся меньше, чем за минуту.

Никогда не забывайте, что реляционные языки более чувствительны к ошибкам в расстановке скобок, чем FORTRAN. Книга *Практическое руководство по SQL* позволит вам избежать подобных ошибок и сэкономит вам много нервов и времени.

Работая с реляционными базами данных с 1976 года, я с удовольствием прочитал *Практическое руководство по SQL*. В нем описывается не только сам язык

SQL, но и рассматриваются общие вопросы построения баз данных и нормализации, причем в соответствии с названием, именно с практической точки зрения.

Многочисленные примеры в *Практическом руководстве по SQL* проясняют важность объясняемых понятий. Например, вам станет ясно, что при соблюдении ссылочной целостности вы не сможете заказать несуществующий товар или обратиться к несуществующему пользователю, другими словами, вашу базу данных нельзя будет “поставить на колени”.

Кроме того, *Практическое руководство по SQL* подготовит вас к будущему. SQL не является “мертвым” языком, разработчики СУБД постоянно расширяют его возможности, которые со временем становятся общепринятыми стандартами. Многие изменения связаны с необходимостью разработки чрезвычайно сложных систем, в которых ведущую роль играют именно базы данных. Расширения SQL, упрощающие администрирование баз данных, особенно важны в системах типа клиент/сервер, в которых персональные компьютеры и рабочие станции общаются с СУБД по сети. В книге *Практическое руководство по SQL* описаны некоторые из таких расширений, в частности язык Transact-SQL, созданный компанией Sybase для своих систем SQL Server и SQL Anywhere.

Наконец, эта книга может послужить прекрасным введением в SQL как для новичков, так и для имеющих некоторый опыт работы с СУБД.

Роберт Эпштейн (Robert Epstein)
Исполнительный вице-президент Sybase, Inc.

Предисловие ко второму и третьему изданиям

ПОЧЕМУ НОВОЕ ИЗДАНИЕ

Многое изменилось с момента выхода в свет этой книги в 1989 году, включая и SQL. Он вырос во всех отношениях — как в количестве пользователей, так и в количестве новых команд. Реляционные базы данных сегодня переживают эпоху своего расцвета.

Когда мы готовили первое издание *Практического руководства по SQL*, ANSI (American National Standards Institute — Американский национальный институт стандартов) предложил стандарт SQL 1986. ISO (International Standards Organization — Международная организация по стандартизации) одобрила его в 1987 году. Этот стандарт был максимально упрощен, не учитывал целый ряд предложений поставщиков коммерческих баз данных и продержался до 1989 года. Стандарт 1989 года учел некоторые пожелания, но все равно в нем отсутствовали многие важные элементы.

На практике в это время стандарты устанавливались *de facto*: каждый поставщик фактически закрывал глаза на то, чем занимались другие поставщики, стараясь просто переманить или привлечь новых пользователей, рекламируя предоставляемые им новые удобные возможности и не заботясь при этом о какой-либо совместимости. Поэтому тогда мы просто оставили эти разные стандарты на суд специалистов и сконцентрировались на общих принципах SQL. Нашей целью было помочь непрофессионалу разобраться в имеющихся тогда разновидностях SQL.

Стандарт ANSI 1992 года (часто называемый SQL2 или SQL-92) отражает новый этап в развитии SQL. Он значительно полнее стандарта 1989 года — его описание в четыре раза больше описания предыдущей версии. Кроме того, он был практически полностью одобрен коммерческими поставщиками СУБД. Таким образом, удалось в основном примирить промышленные стандарты и стандарты ANSI/ISO.

Однако принятие и внедрение стандартов не происходит мгновенно. Между различными реализациями SQL все еще могут быть достаточно значительные различия. Поэтому, прежде чем изучать конкретную реализацию, просто необходимо разобраться в основах SQL.

ВТОРОЕ ИЗДАНИЕ

Изучая литературу, доступную пользователям SQL в 1993 году, мы обнаружили явную нехватку практических рекомендаций по использованию языка. К этому времени уже было достаточно книг, описывающих конкретные реализации, и книг, рассматривающих общие вопросы SQL.

Общаясь с новыми пользователями SQL, мы не раз слышали, что им хотелось бы изучать SQL на практических примерах, на основе которых можно было бы создавать свои собственные системы. Поэтому во второе издание были включены две новые главы, содержащие конкретные примеры использования SQL. Кроме того, были устранены неточности, замеченные в первом издании.

Глава 11 “Разрешение проблем” в основном содержит ответы на вопросы, заданные нашими читателями через USENET. Мы отвечали на них, используя простую базу данных *bookbiz*. Эта глава содержит примеры по организации форматированного вывода результатов, поиску данных, работе с запросами, использованию предложения GROUP BY и индексированию.

Несколько примеров были выделены в отдельную категорию. Они не столько отражают решение определенной задачи, сколько описывают характерные ошибки. Эти примеры собраны в главе 12 “Ошибки, и как их избежать”.

ТРЕТЬЕ ИЗДАНИЕ

Третье издание преследует другую цель. В нем мы описываем основные команды языка SQL, включая расширения стандарта SQL-92, принятые большинством поставщиков СУБД. Они включают новые типы данных, модификации оператора CREATE TABLE, позволяющие контролировать целостность, измененные предложения ORDER BY и GROUP BY, а также новый синтаксис команд GRANT и REVOKE. Мы опишем синтаксис самых общих команд Sybase SQL Server, Sybase Anywhere, Microsoft SQL Server, Informix и Oracle. Все примеры из этой книги, если специально не оговорено обратное, гарантированно работают как на Sybase SQL Server, так и на Sybase Anywhere.

Третье издание *Практического руководства по SQL* — это больше, чем просто книга. На прилагаемом компакт-диске содержится рабочая версия Sybase Anywhere вместе с базой данных *bookbiz*. Это означает, что все примеры из этой книги вы можете выполнить на своем ПК. Успешно изучить SQL можно только на практике. Теперь вы можете экспериментировать, запуская на выполнение примеры из книги и наблюдая за получаемыми результатами. Если какой-то пример покажется вам слишком сложным, разбивайте его исходный код на отдельные части и запускайте их по отдельности, пока точно не уясните, что делает каждая из них. Затем снова соберите их вместе и наслаждайтесь полученными результатами!

БЛАГОДАРНОСТИ

Мы выражаем искреннюю благодарность людям, внесшим свой вклад в создание этой книги:

Донне Джекер (Donna Jeker) и Сту Шустер (Stu Schuster) за постоянную помощь и поддержку.

Джефу Личтману (Jeff Lichtman) и Говарду Торфу (Howard Torf) за советы, примеры, анекдоты и исправление ошибок.

Тому Бонду (Tom Bondur), Сюзи Боуман (Susie Bowman), Джону Куперу (John Cooper) и Вайн Дюкюсне (Wayne Duquesne) за предоставленные материалы и информацию.

Полу Винсбергу (Paul Winsberg) за главу с обзором структур баз данных, вошедшую в первое издание.

Роберту Гарвею (Robert Garvey) за редактирование второго издания.

Карен Эли (Karen Ali) за работу над компакт-диском.

Тео Посселту (Teo Posselt) за редактирование третьего издания.

Введение

НАЧАЛА SQL

Вначале была IBM, и IBM создала SQL.

SQL — сокращение от Structured Query Language (Язык структурированных запросов) — это универсальный язык для создания, модификации и управления данными в реляционных базах данных.

Реляционная модель была предложена в 1970 году И.Ф. Коддом (E.F. Codd), работавшим в исследовательской лаборатории IBM в Сан-Хосе, Калифорния, и развивалась последующие десять лет в университетах и научных организациях. SQL — один из нескольких языков, выросших из этой идеи, в настоящее время практически полностью господствует в мире реляционных баз данных. Производители систем управления реляционными базами данных, первоначально использовавшие другие языки, сегодня полностью переориентировались на SQL.

В этот период (1970—1980) коммерческому успеху систем управления реляционными базами данных препятствовала их невысокая производительность. Преимущества реляционных моделей — математическая строгость и интуитивная простота — в ранних системах нивелировались сложностью, а иногда просто невозможностью управления большими базами данных. Ситуацию изменили два фактора — появление более мощных компьютеров и разработка новых методов хранения и доступа к данным.

В 1981 году IBM объявила о своем первом, основанном на SQL, программном продукте, SQL/DS. В начале 80-х годов о создании собственных реляционных СУБД заявили компании Oracle, Relational Technology и ряд других поставщиков. К 1989 году насчитывалось более 25 SQL-подобных СУБД, работающих на самых разных компьютерах, — от однопользовательских микрокомпьютеров до машин с сотнями пользователей. Сегодня SQL широко применяется в коммерческих, государственных и общественных организациях для работы с базами данных, содержащими самую различную информацию.

Появление высококонкурентного рынка реляционных СУБД было обусловлено созданием целого ряда SQL-приложений, в которых воплотились годы работы по созданию полного и выразительного языка для реляционных моделей. Но проблема остается. Количество диалектов SQL сегодня равно количеству имеющихся на рынке СУБД. И хотя во всех диалектах без труда узнается SQL, все они отличаются друг от друга синтаксисом и семантикой. Более того, многие поставщики постоянно расширяют свои версии SQL, внося этим дополнительную путаницу.

SQL продолжает развиваться — частично из-за того, что его исходная структура не удовлетворяет требованиям некоторых предметных областей, а частично из-за того, что производители СУБД стремятся выжать максимум возможностей из самой реляционной модели.

Гибкость и простота расширения SQL позволяет ему удовлетворять все текущие требования рынка.

Коммерциализация SQL

Первые коммерческие реализации SQL имели тот же налет экспериментальности, что и версии, разрабатываемые в университетах и исследовательских лабораториях. Это во многом было связано с отсутствием в то время каких-либо общепринятых стандартов SQL. Сегодня версии Oracle, SQL/DS и DB2 существенно отличаются от своих родителей. Кроме того, коммерческие реализации по-прежнему отличаются друг от друга во многих отношениях и не полностью соответствуют стандарту ANSI SQL (как выпущенному в 1983 году, так и расширенному в 1992 году).

Ситуация еще более усугубляется из-за появления на рынке сложных диалектов SQL. К счастью, идеи стандартизации и соответствующие тенденции развития

рынка программного обеспечения заставляют производителей выпускать коммерческие версии, совместимые с версиями конкурентов.

Такое взаимное схождение объясняется двумя причинами. Во-первых, компании стараются предлагать своим потенциальным клиентам системы с аналогичными списками функций и возможных расширений. Во-вторых, они постоянно стремятся привлечь к себе внимание пользователей других реляционных систем и поэтому вынуждены уменьшать стоимость перехода с их SQL-систем на свои.

Таким образом, коммерческие интересы заставляют производителей создавать совместимые системы и добавлять новые возможности, чтобы при этом обеспечивалась совместимость с предыдущими версиями.

Эта книга (и прилагаемый компакт-диск с версией Sybase Anywhere) помогут вам самостоятельно изучить основы SQL. Полученные знания вы сможете затем применить для освоения других диалектов SQL. Разобравшись в основах SQL, вы с легкостью сможете переходить на его различные версии.

ДЛЯ КОГО ПРЕДНАЗНАЧЕНА ЭТА КНИГА

Книга *Практическое руководство по SQL* предназначена для пользователей реляционных баз данных — независимо от того, работают ли они на больших многопользовательских компьютерных системах или на персональных компьютерах. Мы не считаем вас компьютерным профессионалом — вы можете быть конечным пользователем в большой компании, государственном учреждении, некоммерческой компании, мелким бизнесменом, менеджером в небольшой организации, домашним пользователем, работающим над собственным проектом, или студентом, изучающим технологии баз данных. Вы могли перейти на реляционные системы баз данных с обычного диспетчера файлов ПК, с другой системы управления базами данных, не основанной на SQL, либо вообще можете их видеть в первый раз.

Мы надеемся, что вы имеете небольшой опыт работы с компьютером и программным обеспечением. Конечно, было бы полезно, если бы вы имели начальные представления и о базах данных.

Если вы планируете разрабатывать сложные приложения для баз данных, вам, возможно, потребуется встраивать команды SQL в программы на других языках программирования или использовать так называемые языки приложений четвертого поколения (4GL). Однако для успешного изучения основ SQL и освоения более сложных тем с помощью книги *Практическое руководство по SQL* вовсе не требуется, чтобы когда-нибудь вами была написана хотя бы одна строка программного кода.

Мы не предназначаем эту книгу для теоретиков, полагая, что тонкости реляционной теории и дебатов ISO—ANSI по поводу стандартов SQL не являются основными интересами наших читателей.

Мы считаем, что вы находитесь в стороне от полемики об SQL — по крайней мере, по поводу его основных структур — и готовы к его использованию. Короче говоря, мы предполагаем: вы хотите узнать, что и как действительно работает или, по крайней мере, получить об этом минимальные представления.

Эта книга построена так, чтобы поэтапно научить вас использовать SQL — вы вводите команду и немедленно получаете ее результат на экране, а не занимаетесь разработкой каких-то сложных программ. Любая коммерческая версия SQL содержит диалоговый интерфейс, который помогает освоить основы языка и позволяет создавать запросы любой сложности. Многие версии предлагают генераторы отчетов или 4GL-языки, которые могут использоваться совместно с командами SQL для построения сложных приложений.

ЦЕЛЬ ЭТОЙ КНИГИ

Исходя из вышесказанного, мы акцентируем ваше внимание на реальном мире коммерческих реализаций SQL, которые мы также будем называть “промышленными SQL”, а не на текущих стандартных ANSI-версиях SQL. Мы делаем это потому, что промышленные версии SQL включают в себя удобные средства обучения, хорошо документированы и обладают большими функциональными возможностями.

- Документация по промышленным версиям SQL обычно написана на простом языке и интуитивно понятна. В документации ISO–ANSI, в противоположность этому, для описания синтаксиса языка используется BNF (Backus Naur Form — форма Бэкуса–Наура), математически строгая, но сложная для чтения и понимания.
- Все промышленные SQL предоставляют диалоговый интерфейс для начинающих и неопытных пользователей, а стандартные версии больше тяготеют к расширениям SQL для программистов и разработчиков программного обеспечения.
- Поставщики систем реляционных баз данных стараются удовлетворить все потребности своих клиентов, учитывая при этом опыт своих конкурентов. Реализация же команд ANSI SQL может различаться в разных версиях. Поэтому в этой книге мы будем опираться на промышленные стандарты (а не на стандарты ANSI) и расскажем о реализованных на сегодня возможностях SQL.

Описание ANSI SQL само по себе является весьма непривлекательным документом, напичканным примечаниями и ссылками. BNF-форма к тому же больше подходит для описания функций каждого элемента языка, чем для представления его синтаксиса. Но если вы все же хотите окупиться в подобные самостоятельные исследования, мы рекомендуем вам начать с книги C.J. Date *A Guide to SQL Standard*. К. Дейт — один из ведущих теоретиков в области реляционных моделей и один из самых известных авторов по этой тематике. С помощью этой книги вы научитесь читать BNF-формы и узнаете мнение профессионала о стандартах SQL.

После “переваривания” документации ISO–ANSI, даже воспользовавшись для этого руководством Дейта, для прояснения некоторых деталей вы, вероятно, захотите обратиться к руководству пользователя по конкретной системе управления базами данных. Но подобные руководства всегда имеют свои недостатки и ограничения.

В то время как в одних руководствах пользователя достаточно хорошо описываются основы SQL, во многих других применяется слишком упрощенный подход, или наоборот, они переполнены ненужными подробностями. Часто руководства пользователя (включая ряд руководств, написанных авторами этой книги) фокусируются на деталях синтаксиса конкретного диалекта в ущерб общей концептуальной картине.

Книга *Практическое руководство по SQL* призвана помочь вам в тех случаях, когда будет бесполезным даже самое лучшее руководство пользователя. С ее помощью вы шаг за шагом изучите основы SQL и научитесь на его основе создавать приложения для работы с базами данных. В *Практическом руководстве по SQL* раскрываются темы, которые обычно пропускаются или написаны мелким шрифтом в стандартных руководствах пользователя, — структуры баз данных, индексация, объединения, подзапросы, виртуальные таблицы (курсоры), производительность и целостность данных.

КАК ИЗУЧАТЬ SQL С ПОМОЩЬЮ ЭТОЙ КНИГИ

Начнем с нескольких предположений. Во-первых, мы надеемся, что большую часть этой книги вы прочтете, сидя за терминалом. Во-вторых, мы рассчитываем, что вы внимательно изучите и выполните все примеры, приведенные в этой книге. Наконец, мы надеемся, что вы будете много экспериментировать. Ничто не может заменить практическую работу, даже очень внимательное изучение программного кода какого-нибудь сверхсложного приложения.

Все остальное зависит от вас. Некоторые люди усваивают новый материал, внимательно вчитываясь в объяснения, другим достаточно просто взглянуть на соответствующие картинки. Мы считаем, что эта книга одинаково полезна и тем, кто вчитывается в каждое слово, и тем, кто бегло просматривает текст, останавливаясь только на примерах. Термины, впервые встречающиеся в тексте и помещенные в словарь терминов, выделяются полужирным начертанием.

Некоторые из сотен примеров книги *Практическое руководство по SQL* будут очень простыми — они служат только для иллюстрации основных понятий. Другие будут более сложными. Некоторые примеры могут послужить моделью для ваших собственных приложений. Наиболее сложные примеры будут объясняться более детально.

Все реализации SQL, по крайней мере в мелких деталях, отличаются друг от друга, поэтому ни одна книга по SQL не может гарантировать, что все приведенные в ней примеры будут работать одинаково на всех системах. Хорошая новость состоит в том, что благодаря нашему решению опираться в книге *Практическое руководство по SQL* на промышленные версии SQL, а не на официальные стандарты, процесс “перевода” с одной версии на другую не будет для вас слишком болезненным. Используя документацию к используемой вами реализации SQL, Приложение Б, описывающее соответствия между ключевыми словами в различных реализациях SQL, и немного поработав детективом, вы сможете успешно выполнить все примеры из этой книги на своей системе.

Все примеры, если специально не оговорено обратное, гарантированно работают в Sybase SQL Server и в Sybase Anywhere. Мы не будем окунаться в самые глубины реализаций SQL и рассуждать о преимуществах и ограничениях того или иного диалекта. Наша цель — рассказать об основных возможностях, имеющихся в большинстве промышленных реализаций SQL.

За исключением ряда случаев, все примеры основаны на нашей базе данных под названием *bookbiz*. Она описывается в главах 2 и 3. Вам не обязательно использовать нашу базу, однако это будет самым простым способом проверки правильности получаемых вами результатов.

Bookbiz — совсем небольшая база данных. Вы можете воспользоваться ее копией, находящейся на прилагаемом компакт-диске, или воссоздать ее на своей системе. Однако эта база достаточно сложна, чтобы проиллюстрировать на ней все важные моменты в технологии реляционных баз данных.

СТРУКТУРА КНИГИ

В каждой главе книги *Практическое руководство по SQL* описывается либо один оператор SQL, либо целый набор логически связанных между собой операторов и конструкций.

Объяснение ведется по следующей схеме.

- Определение — что есть что.
- Минимальный синтаксис — упрощенный “скелет” команды (без расширений и предложений, которые могут изменяться от одной версии к другой).
- Простой пример.

После описания синтаксиса и правил использования мы расскажем вам о роли этого оператора в реляционной модели и о его возможных применениях в приложениях баз данных.

По мере необходимости мы будем приводить расширенный синтаксис команды (необязательные предложения, обеспечивающие дополнительные функции) и соответственно более сложные примеры. В любом случае, каждая новая конструкция будет подробно описываться на содержательных примерах.

Там, где это возможно, каждый новый пример будет основываться на предыдущем. Однако примеры в разных главах не зависят друг от друга, поэтому вы можете читать книгу по отдельным главам. Изучение SQL напоминает изучение иностранного языка. Процесс изучения начинается с подражания, затем выходит на стадию понимания, и наконец заканчивается свободным владением. На каждой стадии ключом к успеху является практика.

Процесс изучения будет для вас более приятным и эффективным, если вы будете выполнять следующее.

Сохраняйте написанные вами команды SQL в файлах. (Ваша система должна позволять это делать.) Если вы не уверены, что в результате выполнения запроса вы получили требуемые результаты, сохраните их для дальнейшего изу-

чения. Запишите, что работает, а что — нет, и если это возможно, сохраните сообщения об ошибках.

Сохраняйте удачно отработавшие запросы. Сохраняйте их в отдельных файлах, удачные решения могут пригодиться вам в будущем.

Группируйте запросы в отдельные модули или процедуры. В соответствии с современными методами структурного программирования приложения SQL должны состоять из набора отдельных процедур, что значительно упрощает их модификацию и повторное использование.

Заведите специальную тетрадку. Даже если в документацию к вашей системе включен краткий справочник команд, все равно сделайте это. Это будет способствовать лучшему усвоению материала. Вскоре вы обнаружите, что одни команды SQL используются значительно чаще других.

Старайтесь улучшать предлагаемые нами решения. Наш опыт свидетельствует, что чем больше вы будете самостоятельно работать над освоением SQL, тем более простыми и элегантными будут становиться написанные вами операторы.

Изучив SQL на практике, вы сможете правильно использовать его операторы, получая требуемые результаты. Чтобы достичь такого уровня профессионализма, вы, кроме того, должны изучить и проверить вашу реляционную СУБД. Проверьте, чтобы выполнение на ней 50 транзакций в секунду не приводило к появлению сообщения об ошибке SQL (логической ошибке).

SQL требует практики, потому что это фактически иностранный язык, причем говорить нужно исключительно четко, так как общаться вы будете с машиной. Несмотря на то, что в SQL содержится заведомо конечное число легко читаемых операторов и ключевых слов, это достаточно запутанная область. Подобно другим языкам программирования высокого уровня, SQL имеет строго определенную грамматику и структуру и полный набор синтаксических правил. Он во многом напоминает английский язык, но на самом деле весьма далек от естественного языка. Рано или поздно вы обязательно столкнетесь с операциями, выполнение которых будет SQL не под силу.

Практическое руководство по SQL поможет вам разобраться в сильных и слабых областях SQL. Оно уберезет вас от потенциальных неприятностей, вызываемых ошибками в структурах баз данных, и сделает максимально комфортным процесс изучения SQL.

КРАТКИЙ ОБЗОР КНИГИ

Глава 1. SQL и управление реляционными базами данных. В этой главе кратко описывается и иллюстрируется реляционная модель, рассказывается об основных возможностях языка SQL.

Глава 2. Проектирование баз данных. Проектирование баз данных часто вызывает значительные затруднения. В этой главе на примере простой базы данных будут проиллюстрированы полезные методы анализа данных и принятия решений о структуре создаваемой системы. Мы расскажем о первичных и вторичных ключах, моделировании связей, правилах нормализации, обо всем том, что позволит вам проектировать качественные базы данных.

Глава 3. Создание и заполнение базы данных. В этой главе структура базы данных, описанная в предыдущих главах, наполнится реальным содержимым. Мы детально расскажем о командах SQL для создания баз данных, таблиц, индексов, о командах добавления, изменения и удаления данных. Изучив синтаксис этих команд, вы сможете практически использовать язык SQL.

Глава 4. Выборка информации из базы данных. В этой главе мы начнем использовать примеры с компакт-диска и описывать основные элементы оператора SELECT. Мы расскажем, как извлекать из таблицы необходимые строки и столбцы, как выполнять вычисления, применять логические операторы и операторы сравнения.

Глава 5. Сортировка данных и другие методы выбора. Специальные предложения в операторе SELECT позволят вам сортировать данные, удалять повторяющуюся информацию в результирующих данных, использовать специальные функции для вычисления средних значений, сумм и других подсчетов.

Глава 6. Группировка данных и построение отчетов. Оператор SELECT также предоставляет вам возможности по группировке данных и генерации отчетов, используя для этого агрегирующие функции, описанные в предыдущих главах. В этой главе также ведется дискуссия о том, как в системах реляционных баз данных должны обрабатываться нулевые значения (пропуски информации).

Глава 7. Объединение таблиц и сложный анализ данных. Оператор объединения — один из китов реляционной модели. В этой главе рассказывается, как использовать этот оператор для выборки данных из одной или нескольких таблиц. Усложненный вариант простого оператора выбора, оператор объединения, требует от пользователей повышенного внимания к анализу и проверке получаемых данных.

Глава 8. Структурированные запросы и подзапросы. В этой главе мы расскажем о правильном использовании вложенных запросов, или подзапросов. На большом количестве примеров описываются коррелированные подзапросы, которые часто являются источником многих неприятностей.

Глава 9. Создание и использование виртуальных таблиц (курсоров). В этой главе описывается использование виртуальных таблиц (курсоров) для организации удобного доступа к данным. Виртуальные таблицы также позволяют обеспечить дополнительную безопасность, так как с их помощью можно запретить другим пользователям доступ к определенным частям таблицы и выполнение над ними определенных операций.

Глава 10. Безопасность, транзакции, производительность и целостность. Эта глава посвящена характерным вопросам управления реальными базами данных. В ней описываются команды для установления полномочий пользователей, кроме того, мы вернемся к теме индексирования с позиций повышения производительности системы, а также рассмотрим механизмы управления транзакциями. В этой главе также рассматриваются расширения языка SQL, обеспечивающие непротиворечивость и целостность данных. Некоторые из них характерны исключительно для реализации Sybase SQL.

Глава 11. Разрешение проблем. В этой главе на основе нашей базы *bookbiz* мы ответим на вопросы, заданные пользователями через Internet. Здесь вы столкнетесь с реальными проблемами — представлением результатов, поиском данных, много-табличными запросами и предложением GROUP BY. Эта глава напоминает кулинарную книгу, напичканную полезными советами, которыми вы можете воспользоваться в своей работе.

Глава 12. Ошибки, и как их избежать. Эта глава также наполнена примерами, взятыми из Internet и сформулированными в терминах базы *bookbiz*, но имеющими несколько другой оттенок. Это примеры наиболее характерных ошибок. Здесь вы найдете описание типичных ошибок при использовании конструкций GROUP BY, HAVING, WHERE и DISTINCT. Эта глава поможет вам избежать классических ошибок.

Приложение А. Краткое описание синтаксиса SQL.

Приложение Б. Аналогии между ключевыми словами разных диалектов SQL.

Приложение В. Словарь терминов.

Приложение Г. Описание базы данных *bookbiz*. Это приложение содержит таблицы данных, описание структуры и код для создания самой базы.

Приложение Д. Список литературы.

SQL и управление реляционными базами данных

УПРАВЛЕНИЕ РЕЛЯЦИОННЫМИ БАЗАМИ ДАННЫХ

SQL — это язык, на котором можно “разговаривать” с реляционными базами данных. Минуточку! А что такое система управления реляционными базами данных?

Все системы управления базами данных предназначены для хранения и обработки информации. Реляционный подход к управлению базами данных основан на математической модели, использующей методы реляционной алгебры и реляционного исчисления. Тем не менее большинство действительно необходимых вам определений из области управления базами данных скорее относятся к практической, чем к теоретической стороне этого вопроса.

С. Дейт дает следующее неформальное определение системе управления реляционными базами данных (СУБД).

- Вся информация в базе данных представлена в виде таблиц.
- Она поддерживает три реляционных оператора — выбора, проектирования и объединения, с помощью которых вы получаете необходимые вам данные (и можете выполнять эти операции, не требуя от системы физической записи получаемых с их помощью данных в каком-то определенном виде).

Д-р. И.Ф. Кодд, автор реляционной модели, разработал целый список критериев, которым должна удовлетворять реляционная модель. Описание этого списка, часто называемого “правилами Кодда”, требует введения сложной терминологии и теоретических выкладок, что выходит за рамки данной книги. Тем не менее в следующих главах мы опишем состоящий из 12 правил тест Кодда для реляционных систем и будем использовать его совместно с более общим определением Дейта. Чтобы считаться реляционной, система управления базами данных должна:

- представлять всю информацию в виде таблиц,
- поддерживать логическую структуру данных, независимо от их физического представления,
- использовать язык высокого уровня для структурирования, выполнения запросов и изменения информации в базах данных (теоретически это может быть любой язык баз данных, практически для этого используется язык SQL),
- поддерживать основные реляционные операции (выбор, проектирование и объединение), а также теоретико-множественные операции, такие как объединение, пересечение и дополнение,
- поддерживать виртуальные таблицы, обеспечивая пользователям альтернативный способ просмотра данных в таблицах,
- различать в таблицах неизвестные значения (**nulls**), нулевые значения и пропуски в данных,
- обеспечивать механизмы для поддержки целостности, авторизации, транзакций и восстановления данных.

В оставшейся части главы содержится обзор этих пунктов, ко многим из них мы будем обращаться и в последующих главах. Прочитав эту главу, вы будете разбираться в основных законах реляционных баз данных.

Реляционная модель: одни таблицы

Первое правило Кодда гласит, что вся информация в реляционных базах данных представляется значениями в **таблицах (table)**. В реляционных системах таблицы состоят из горизонтальных **строк (row)** и вертикальных **столбцов (column)** (рис. 1.1). Все данные представляются в табличном формате — другого способа просмотреть информацию в базе данных не существует.

Несколько замечаний по терминологии. Поскольку такие понятия как **таблица**, **строка** и **столбец** являются общепринятыми в коммерческих системах управления реляционными базами данных, мы будем стараться использовать их в этой книге. Однако иногда вы можете встретиться и с такими понятиями, как **отношение (relation)**, **кортеж (tuple)** и **атрибут (attribute)**. Это соответственно синонимы понятий **таблица**, **строка** и **столбец**, так же, как и **файл (file)**, **запись (record)** и **поле (field)**. Первые три считаются академическими терминами, последние — взяты из общего лексикона, используемого в области обработки данных.

Набор связанных таблиц образует **базу данных (database)**. Таблицы в реляционной базе разделены, но полностью равноправны. Между ними не существует никакой иерархии и, вообще говоря, они не обязательно даже физически связаны друг с другом.

Name	Address
Jane Doe	127 Elm St.
Richard Roe	10 Trenholm Place
Edgar Poe	1533 User House

Строки

Столбцы

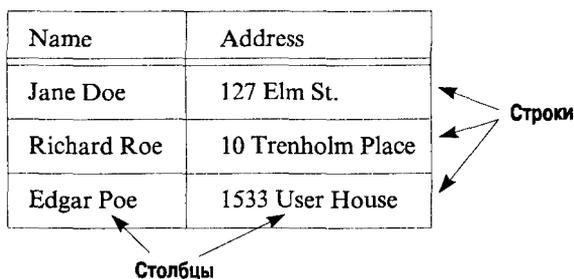


Рис. 1.1. Таблица *personnel*

Каждая таблица состоит из строк и столбцов. Каждая строка описывает отдельный **объект** или **сущность (entity)** — человека, компанию, торговую сделку или что-нибудь другое. Каждый столбец описывает одну характеристику объекта — имя человека или его адрес, телефонный номер компании или ее президента, лоты распродажи или дату.

Каждый элемент данных, или **значение (value)**, определяется пересечением строки и столбца таблицы. Чтобы найти требуемый элемент данных, необходимо знать имя содержащей его таблицы, столбец и значение его **первичного ключа (primary key)**, или уникального идентификатора. (Как вы узнаете из главы 2, каждая строка должна единственным образом идентифицироваться по одному из своих значений.)

Предположим, вы хотите узнать адрес Richard Roe. Чтобы добраться до этой информации, вы приказываете системе извлечь этот адрес из таблицы под названием *personnel* из столбца *address*. При этом имя *Richard Roe* является значением первичного ключа, идентифицирующим эту строку (рис. 1.2).

В реляционных базах данных существует два типа таблиц — **пользовательские таблицы (user table)** и **системные таблицы (system tables)**. Пользовательские таблицы содержат информацию, для поддержки которой собственно и создавались системы реляционных баз данных — данные по сделкам, заказам, персоналу и т.д. Системные таблицы, известные также под названием **системные каталоги (system catalog)**, содержат описание базы данных. Системные таблицы обычно поддерживаются самой СУБД, однако доступ к ним можно получить так же, как и к любым другим таблицам. Возможность получения доступа к системным таблицам, по аналогии с любыми другими таблицами, составляет основу другого правила Кодда для реляционных систем.

Name	Address
Jane Doe	127 Elm St.
Richard Roe	10 Trenholm Place
Edgar Poe	1533 User House

На пересечении строки и столбца находится соответствующий адрес

Рис. 1.2. Получение данных из таблицы

Независимость

При управлении базами данных, как и в нашей жизни, все стремятся к независимости. Независимость данных — критический аспект при управлении любой системой баз данных. Она позволяет изменять приложения, не изменяя для этого структуру базы данных, и изменять конструкцию базы данных, не оказывая при этом влияния на работу приложений. Система управления базами данных не должна вынуждать вас выносить окончательные решения о том, какие данные вы должны сохранять, как получать к ним доступ и что будет нужно вашим пользователям. Система не должна становиться бесполезной при изменении ваших потребностей.

Реляционная модель обеспечивает независимость данных на двух уровнях — *физическом* и *логическом*. **Физическая независимость данных (physical data independence)** означает с точки зрения пользователя, что представление данных абсолютно не зависит от способа их физического хранения. Как следствие этого, физическое перемещение данных никоим образом не может повлиять на логическую структуру базы данных и ваше восприятие данных.

Такие изменения обычно становятся просто необходимыми, особенно в больших многопользовательских системах. Например, при недостатке места для хранения информации может потребоваться установка дополнительных физических носителей. Когда устройство выходит из строя — увы, его приходится быстро заменять. Иногда может потребоваться увеличить производительность системы или упростить ее использование, изменив для этого методы доступа к физическим данным. (Эти методы связаны с созданием **стратегии доступа (access strategies)** и применением **индексов (index)**.)

Другой тип независимости, обеспечиваемый реляционными системами — **логическая независимость (logical independence)** — означает, что изменение взаимосвязей между таблицами, столбцами и строками не влияет на правильное функционирование программных приложений и текущих запросов. Вы можете разбивать таблицы по строкам или столбцам, а приложения и запросы все равно будут выполняться, как и раньше. Несмотря на изменение логической структуры базы данных, вы всегда можете воспользоваться своими старыми запросами.

Требование логической и физической независимости данных составляет основу двух других правил Кодда.

Язык высокого уровня

Определение реляционной системы, так же, как и правила Кодда, требует, чтобы весь диалог с базой данных велся на едином языке — иногда его называют **общим подязыком данных (comprehensive data sublanguage)**. В мире коммерческих систем управления базами данных такой язык получил название SQL.

SQL используется для **манипуляций с данными (data manipulation)** — выборки и модификации, **определения данных (data definition)** и **администрирования данных (data administration)**. Любая операция по выборке, модификации, определению или администрированию выполняется с помощью **оператора (statement)** или **команды (command) SQL**.

Имеется две разновидности операций по манипуляции с данными — **выборка данных (data retrieval)** и **модификация данных (data modification)**. Выборка — это поиск необходимых вам данных, а модификация означает добавление, удаление или изменение данных.

Операции по выборке (чаще называемые **запросами (queries)**) осуществляют поиск в базе данных, наиболее эффективно извлекают затребовавшую вами информацию и отображают ее. Во всех запросах SQL используется ключевое слово **SELECT**.

Дальше в этой главе вы найдете описание нескольких простых SQL-запросов. Пока особенно не задумывайтесь над их синтаксисом — он будет подробно описан в главе 3. Сейчас только посмотрите примеры запросов и результаты их выполнения, чтобы насладиться прелестями SQL и получить начальные представления.

Следующий оператор **SELECT** покажет вам все данные из таблицы *publishers*, которая является частью базы данных *bookbiz*.

SQL:

```
select*
from publishers
```

Символ звездочки (*) заменяет названия всех столбцов в таблице. В результате выполнения этого запроса вы получите:

Результат:

pub_id	pub_name	address	city	state
0736	New Age Books	1 1st St	Boston	MA
0877	Binnet & Hardley	2 2nd Ave.	Washington	DC
1389	Algodata Infosystems	3 3rd Dr.	Berkeley	CA

Операции по модификации выполняются соответственно с использованием ключевых слов **INSERT**, **DELETE** и **UPDATE**. С помощью следующей команды можно добавить строку в таблицу *publishers*.

SQL:

```
insert into publishers
values ('0010', 'Pragmatics', '4 4th Ln.', 'Chicago', 'IL')
```

Если вы снова просмотрите данные, воспользовавшись оператором **SELECT**, то увидите новую строку.

SQL:

```
select *
from publishers
```

Результат:

pub_id	pub_name	address	city	state
0736	New Age Books	1 1st St	Boston	MA
0877	Binnet & Hardley	2 2nd Ave.	Washington	DC
1389	Algodata Infosystems	3 3rd Dr.	Berkeley	CA
0010	Pragmatics	4 4th Ln.	Chicago	IL

Другие команды SQL предназначены для создания и удаления таблиц, индексов и других объектов. Следующая команда создает таблицу под названием *test* с двумя столбцами — *id* для целых чисел и *name* для хранения символьной информации (до 15 символов).

SQL:

```
create table test
(id int,
name char (15))
```

Вы можете смело пользоваться оператором SELECT, даже несмотря на то, что в таблице *test* пока нет никаких данных.

```
SQL:
select *
from test
```

Результат:

```
id          name
-----
```

Последняя категория операторов SQL — операторы администрирования, или команды управления данными (**data control**). Они позволяют вам координировать совместное использование базы данных и поддерживать ее в наиболее эффективном состоянии.

Одним из наиболее важных аспектов администрирования многопользовательских систем управления базами данных является управление доступом к данным. Хорошим примером административных команд является ключевое слово GRANT, позволяющее установить полномочия пользователей. В следующем примере пользователю с именем *karen* разрешается выбирать данные из таблицы *test*.

```
SQL:
grant select
on test
to karen
```

Имейте в виду, что мы еще не приступали к рассмотрению реляционных операций и синтаксиса SQL, так что все это — чисто вводные замечания.

Реляционные операции

В определении системы управления реляционными базами данных упоминаются три операции по выборке данных — проектирование, выбор (иногда называемый **ограничением (restriction)**) и объединение, которые позволяют строго указать системе, какие данные вы хотите увидеть. Операция проектирования выбирает столбцы, операция выбора — строки, а операция объединения собирает вместе данные из связанных таблиц.

Логическая и физическая независимость, о которой мы упоминали в этой главе, означает, что вам не нужно беспокоиться о физическом расположении данных и о том, как их искать — это проблемы исключительно систем управления базами данных. SQL можно рассматривать как **непроцедурный язык программирования (nonprocedural language)**, так как он позволяет вам выразить то, что вы хотите получить, не вдаваясь в детали самого процесса.

Все эти три операции записываются с использованием ключевого слова SELECT. Именно так! В SQL ключевое слово SELECT используется не только для выбора данных, но и для выполнения операций проектирования и объединения.

Чтобы насладиться красотой оператора SELECT, познакомьтесь с его упрощенным синтаксисом.

```
SQL:
SELECT список_выбора
FROM список_таблиц
WHERE условия_поиска
```

В следующих подразделах объясняется, как этот простой на вид оператор используется для выполнения трех реляционных операций.

Проектирование. Операция проектирования позволяет указать системе, какие столбцы таблицы вы хотите просмотреть. Например, если вы хотите просмотреть все строки таблицы, содержащей информацию об издательствах, но вам нужны

данные только об их названиях и идентификационных номерах, воспользуйтесь следующим оператором выборки:

```
SQL:
select pub_id, pub_name
from publishers
```

Результат:

pub_id	pub_name
0736	New Age Books
0877	Binnet & Hardley
1389	Algodata Infosystems
0010	Pragmatics

Опять-таки, пока не заостряйте свое внимание на синтаксисе. Взгляните на это с концептуальной точки зрения: операция проектирования определяет подмножество столбцов в таблице. Обратите внимание, что результаты выполнения проектирования (как и любой другой реляционной операции) также отображаются в форме таблицы. Результирующие таблицы иногда называют **производными таблицами (derived tables)**, чтобы отличать их от **базовых таблиц (base table)**, содержащих исходные строки данных.

Выбор. Операция выбора позволяет вам получать из таблицы подмножества ее строк. Чтобы указать, какие строки вам нужны, соответствующие условия нужно разместить в предложении WHERE. В предложении WHERE оператора SELECT определяется критерий, которому должны соответствовать выбираемые строки. Например, если вы хотите получить информацию только об издательствах, расположенных в Калифорнии, введите следующий запрос:

```
SQL:
select *
from publishers
where state = "CA"
```

В результате выполнения запроса вы получите:

Результат:

pub_id	pub_name	address	city	state
1389	Algodata Infosystems	3 3rd Dr.	Berkeley	CA

Вы можете комбинировать в запросе операции проектирования и выбора, чтобы получить требуемую информацию.

Объединение. Операция объединения может работать одновременно с одной или несколькими таблицами, соединяя данные таким образом, что вы сможете легко сопоставить или выделить определенную информацию в своей базе данных.

Операция объединения обеспечивает SQL и реляционную модель необходимой мощностью и гибкостью. Вы можете выявить любую взаимосвязь, существующую между элементами данных, а не только связи, введенные при конструировании базы.

Когда вы “объединяете” две таблицы, на период действия запроса они как бы становятся единой таблицей. Операция объединения соединяет данные, сравнивая значения в заданных столбцах и отражая результаты.

Проще всего можно разобраться в выполнении операции объединения на конкретном примере. Предположим, вы хотите узнать имена издателей всех книг в нашей базе данных.

Названия всех книг содержатся в таблице *titles*. Кроме того, в ней хранится и другая полезная информация о книгах, в том числе и идентификационный номер издателя (рис. 1.3). Однако в этой таблице вы не найдете имя издателя — оно хранится в таблице *publishers*.

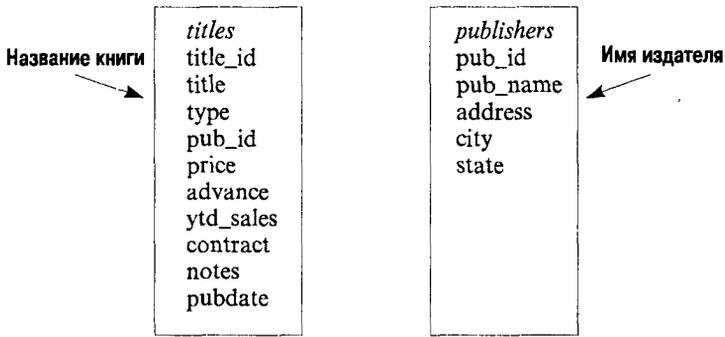


Рис. 1.3. Столбцы таблиц *titles* и *publishers*

Поставленная задача решается благодаря тому, что обе таблицы — *titles* и *publishers* — содержат идентификационные номера издателей. Чтобы получить одновременно и имена издателей, и названия книг, вы можете объединить эти таблицы (рис 1.4).

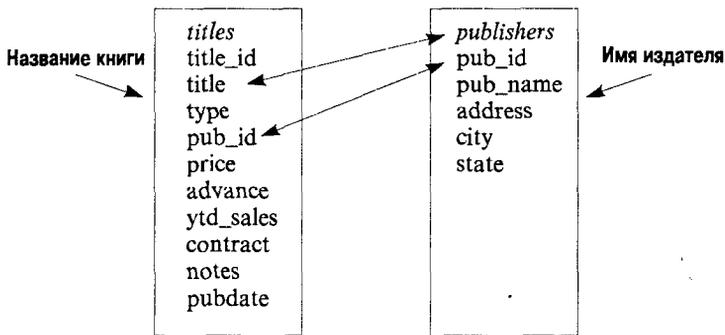


Рис. 1.4. Разделяемые столбцы в таблицах *titles* и *publishers*

Система будет искать все случаи совпадения значений в столбцах *pub_id* обеих таблиц. При каждом совпадении будет создаваться новая строка, содержащая значения из столбцов объединяемых таблиц. Ниже приведен код запроса.

SQL:

```
select title, pub_name
from titles, publishers
where publishers.pub_id = titles.pub_id
```

Столбец *title* в предложении SELECT относится к таблице *titles*, столбец *pub_name* — к таблице *publishers*. Операция проектирования позволяет в одном списке указывать столбцы из разных таблиц. В предложении FROM задаются две объединяемые таблицы. В предложении WHERE говорится, что будут объединяться строки этих таблиц, имеющие одинаковое значение идентификационного номера в столбце *pub_id*.

Ниже приводится получаемый результат.

Результат:

title	pub_name
Computer Phobic and Non-Phobic Individuals: Behavior Variations	New Age Books
Emotional Security: A New Algorithm	New Age Books
Prolonged Data Deprivation: Four Case Studies	New Age Books
Life Without Fear	New Age Books

Is Anger the Enemy?	New Age Books
You Can Combat Computer Stress!	New Age Books
The Psychology of Computer Cooking	Binnet & Hardley
Silicon Valley Gastronomic Treats	Binnet & Hardley
Sushi, Anyone?	Binnet & Hardley
Fifty Years in Buckingham Palace Kitchens	Binnet & Hardley
The Gourmet Microwave	Binnet & Hardley
Onions, Leeks, and Garlic: Cooking Secrets of the Mediterranean	Binnet & Hardley
Secrets of Silicon Valley	Algodata Infosystems
Net Etiquette	Algodata Infosystems
Cooking with Computers: Surreptitious Balance Sheets	Algodata Infosystems
Straight Talk About Computers	Algodata Infosystems
The Busy Executive's Database Guide	Algodata Infosystems
But Is It User Friendly?	Algodata Infosystems

В этом месте у вас могут возникнуть вопросы: “А не преувеличиваем ли мы значение оператора объединения? Почему нельзя поместить все эти столбцы в одну таблицу?” При этом, если таблица получится слишком широкой, для отображения нужного количества столбцов всегда можно будет использовать операцию проектирования.

Это вполне уместные вопросы. Дело в том, что количество столбцов в таблице часто ограничивается для обеспечения логической согласованности данных и просто для удобства использования таблиц. Обсуждение в следующей главе, посвященное проектированию баз данных, поможет вам решить, какие столбцы нужно включать в какую таблицу.

Альтернативный способ просмотра данных

Курсор (view) — это альтернативный способ просмотра данных из нескольких таблиц. Курсоры иногда называются **виртуальными таблицами (virtual tables)**, или производными таблицами. Таблицы, на основе которых работают курсоры, называются базовыми таблицами. Курсор можно рассматривать как перемещаемую по таблицам рамку, через которую вы можете увидеть только необходимую вам часть информации. Курсор можно получить из одной или нескольких таблиц базы данных (включая и другие курсоры), используя любые операции выбора, проектирования и объединения. Курсоры позволяют вам создавать таблицы для специальных целей. С их помощью вы можете использовать результаты выполнения операторов выбора, проектирования и объединения как основу для последующих запросов.

Виртуальные таблицы, в отличие от “настоящих”, или базовых таблиц, физически не хранятся в базе данных. Важно осознать, что курсор — это не копия некоторых данных, помещаемая в другую таблицу. Когда вы изменяете данные в виртуальной таблице, то тем самым изменяете данные в базовых таблицах. Подобно результатам операции выбора, курсоры напоминают обычные таблицы баз данных.

Курсоры создаются с помощью оператора **SQL CREATE VIEW**. Чтобы преобразовать результаты выполнения оператора **SELECT** в курсор, просто поместите перед ним команду **CREATE VIEW**. Чтобы получить виртуальную таблицу на основе предыдущего примера, воспользуйтесь следующим оператором **CREATE VIEW**:

```
SQL:
create view Book_and_Pubs
as
select title, pub_name
from titles, publishers
where publishers.pub_id = titles.pub_id
```

Если вы примените операцию выбора к виртуальной таблице, то увидите результаты выполнения запроса, на основе которого она была создана.

В идеальной реляционной системе с курсорами можно оперировать как и с любыми другими таблицами. В реальном мире различные версии SQL накладывают на курсоры определенные ограничения, в частности на обновление. Одно из правил Кодда гласит, что в истинно реляционной системе над курсорами можно выполнять все “теоретически” возможные операции. Большинство современных систем управления реляционными базами данных не удовлетворяют этому правилу полностью.

Курсоры будут подробно описаны в главе 9, там же мы поговорим о том, что означают слова “теоретически обновляемый курсор”.

Нули

В реальном мире управления информацией данные часто являются неизвестными или неполными: вы можете забыть узнать телефонный номер, ваш респондент может не захотеть назвать свой возраст, книга может быть направлена в печать, но дата ее выхода еще может быть неизвестна. Такие пропуски информации создают “дыры” в ваших таблицах.

Проблема, конечно, состоит не в простой неприглядности подобных дыр. Опасность состоит в том, что из-за них ваша база может стать противоречивой. Чтобы сохранить целостность данных в реляционной модели, так же, как и в правилах Кодда, для обработки пропущенной информации используется понятие нуля.

“Ноль” не означает пустое поле или обычный математический ноль. Он отображает тот факт, что значение неизвестно, недоступно или неприменимо. Существенно, что использование нулей инициирует переход с двухзначной логики (да/нет или что-то/ничего) на трехзначную (да/нет/может быть или что-то/ничего/не уверен).

С точки зрения другого эксперта по реляционным системам, Дейта, нули не являются полноценным решением проблемы пропусков информации. Тем не менее они являются составной частью большинства официальных стандартов SQL и de facto промышленных стандартов. Нули — это настолько важная тема, что мы будем обращаться к ней в нескольких главах. В главе 3 объясняется, как создавать таблицы, в которых в определенных столбцах могут располагаться нули. В главе 4 мы поговорим об использовании операции выбора применительно к таблицам с нулями. В главе 5 нули рассматриваются в контексте применения функций упорядочивания и агрегирующих функций. В главе 6 обобщается информация, связанная с использованием нулей в системах управления базами данных.

Безопасность

Понятие безопасности связано с необходимостью управления доступом к информации. Команды SQL GRANT и REVOKE позволяют некоторым привилегированным пользователям устанавливать права других пользователей на просмотр и модификацию информации в базе данных. В большинстве реализаций SQL правами на доступ и модификацию данных (permission) можно управлять на уровне таблиц и столбцов.

Эти права устанавливают владельцы (owner) баз данных или объектов баз данных.

Владельцем базы данных является создавший ее пользователь с помощью команды SQL CREATE. Некоторые системы разрешают передавать права владения от создателя базы другому пользователю.

В многопользовательских системах обычно имеется пользователь с правами даже более высокими, чем у владельца базы данных — системный администратор (system administrator), или администратор базы данных (database administrator). Этот пользователь обычно обладает широкими правами на наделение полномочий, а также выполняет целый ряд других задач, связанных с поддержкой и администрированием базы данных.

В качестве дополнительного механизма обеспечения безопасности могут выступать и виртуальные таблицы. Пользователи могут разрешать доступ только к определенному подмножеству своих данных, включенному в виртуальную таблицу. Виртуальные таблицы рассматриваются в главе 9, операторы GRANT и REVOKE — в главе 10.

Целостность

Целостность (integrity) — очень сложный и серьезный вопрос при управлении реляционными базами данных. Несогласованность между данными может возникнуть по целому ряду причин. Несогласованность или противоречивость данных может возникать вследствие сбоя системы — проблемы с аппаратным обеспечением, ошибки в программном обеспечении или логические ошибки в приложениях. Реляционные системы управления базами данных защищают данные от такого типа несогласованности, гарантируя, что команда SQL либо будет исполнена до конца, либо будет полностью отменена. Этот процесс обычно называют **управлением транзакциями (transaction management)**. Транзакции и связанные с ними методы SQL обсуждаются в главе 10.

Другой тип целостности, называемый **объектной целостностью (entity integrity)**, связан с корректным проектированием базы данных. Объектная целостность требует, чтобы ни один первичный ключ не имел нулевого значения.

Третий тип целостности, называемый **ссылочной целостностью (referential integrity)**, означает непротиворечивость между частями информации, повторяющимися в разных таблицах. Например, если вы изменяете неправильно введенный номер карточки социального страхования в одной таблице, другие таблицы, содержащие эту же информацию, продолжают ссылаться на старый номер, поэтому вы должны обновить и эти таблицы. *Чрезвычайно важно, чтобы при изменении информации в одном месте, она соответственно изменялась и во всех других местах.*

Правила Кодда гласят, что системы управления реляционными базами данных должны обеспечивать не только объектную и ссылочную целостность, но и позволять “вводить дополнительные ограничения на целостность, отражающие специальные требования”. Кроме того, по определению Кодда, ограничения на целостность должны:

- определяться на языке высокого уровня, используемом системой для всех других целей;
- храниться в словаре данных, а не в программных приложениях.

Первоначально только несколько реализаций SQL удовлетворяли критериям Кодда на целостность, но ситуация постепенно менялась. ANSI-стандарт SQL 1992 года (часто называемый “SQL2”) поддерживает ограничения в операторе CREATE TABLE, обеспечивающие ссылочную целостность и позволяющие задавать бизнес-правила. Эти возможности в том или ином виде реализованы в большинстве систем. Подробно они рассматриваются в главе 3.

ПРИСТУПАЯ К ПРОЕКТИРОВАНИЮ БАЗЫ ДАННЫХ

Теперь, когда вы получили основные представления о системах реляционных баз данных, можно приступать к работе. Но прежде чем извлекать или изменять информацию в базе данных, ее туда нужно поместить. А перед тем как сделать это, вы должны решить, как будет выглядеть ваша база данных. Именно проектированию баз данных мы и посвятим главу 2.

Проектирование баз данных

СТРУКТУРА БАЗЫ ДАННЫХ

Процесс, в ходе которого решается, какой вид будет у вновь создаваемой базы данных, называется **проектированием базы данных (database design)**. Работа по проектированию базы данных включает выбор

- таблиц, которые будут входить в базу данных,
- столбцов, принадлежащих каждой таблице,
- взаимосвязей между таблицами и столбцами.

Конструирование базы данных связано с построением ее логической структуры. В реляционной модели логическая структура базы абсолютно не зависит от ее физической структуры и способа хранения. Логическая структура также не определяется тем, что видит у себя на экране конечный пользователь (это могут быть виртуальные таблицы, созданные разработчиком (подробно об этом — в главе 9) или прикладными программами).

Конструирование баз данных на основе реляционной модели имеет ряд важных преимуществ перед другими моделями.

- Независимость логической структуры от физического и пользовательского представления.
- Гибкость структуры базы данных — конструктивные решения не ограничивают ваши возможности выполнять в будущем самые разнообразные запросы.

Так как реляционная модель не требует от вас описания всех возможных связей между данными, вы можете впоследствии задавать запросы о любых логических взаимосвязях, содержащихся в базе, а не только о тех, которые планировались первоначально. (В этой главе под словом “вы” мы подразумеваем разработчика базы данных.)

С другой стороны, реляционные системы не имеют никаких встроенных защитных механизмов против некорректных структурных решений и не умеют различать хорошую структуру базы данных от посредственной. К тому же не существует автоматизированных средств, которые могли бы заменить вас в процессе принятия структурных решений.

Конструирование баз данных, как и ряд других вопросов, рассматриваемых в этой книге, является очень сложным предметом. Ему посвящены сотни статей и дюжины книг — некоторые из них напичканы жаргоном и абстрактной терминологией, другие предназначены для начинающих пользователей персональных компьютеров. (Наиболее полезные из них, с нашей точки зрения, представлены в списке литературы.)

Краткий и в большей степени практический, нежели теоретический, материал этой главы призван помочь вам в разработке баз данных средней сложности. Вы познакомитесь с основной терминологией, так что в случае необходимости, сможете впоследствии разобраться и в более сложных материалах, касающихся разработки баз данных. Основные принципы остаются неизменными, независимо от того, какую базу данных вы разрабатываете — простую или сложную.

Если вы работаете в однопользовательской системе, вероятнее всего, структура вашей базы данных будет достаточно проста, так что с помощью этой главы вы сможете без труда в ней разобраться.

Если вы работаете в многопользовательской системе, вопросы структуры и создания базы данных будут находиться в компетенции специалиста. Так обязательно будет, если ваши приложения используют данные, важные для других пользователей и всей вашей организации. Этим специалистом может быть системный администратор или администратор базы данных, который хорошо разбирается в локальной архитектуре вашей системы, требованиях вашей организации и правилах построения реляционных баз данных. Чтобы создать простую в использовании базу данных, которую будет легко поддерживать и которая будет удовлетворять требованиям эффективности и непротиворечивости, ее разработчик также должен быть хорошо знаком со всеми этими вопросами.

Даже если вы полагаетесь исключительно на экспертов по проектированию баз данных, эта глава поможет вам стать квалифицированным пользователем, научив анализировать взаимосвязи между данными. Понимание основ реляционных структур поможет вам как в изучении и практическом освоении SQL, так и в поддержке, обновлении и создании запросов к базам данным, созданным другими пользователями. Кроме того, если вы будете обращаться за помощью к специалистам, такое понимание позволит вам четко формулировать свои вопросы.

Как и в других главах *Практического руководства по SQL*, в этой главе в качестве примера мы будем использовать базу данных *bookbiz*. Чтобы освоить материал этой главы и разобраться в структуре базы *bookbiz*, вам потребуется изучить команду SQL CREATE и познакомиться с процессом определения данных.

Другие вопросы, связанные с проектированием базы данных и определением данных, рассматриваются в главе 9 (создание виртуальных таблиц, удовлетворяющих специальные требования пользователей), в главе 10 (безопасность, наделение полномочиями на выполнение определенных действий с данными), в главах 3 и 10 (целостность, обеспечение непротиворечивости связанных данных).

Как подходить к проектированию базы данных

Рассуждения по поводу проектирования реляционных систем часто могут показаться просто шизофреническими. С одной стороны, мы говорим, что реляционная модель делает процесс проектирования интуитивно понятным и простым, а с другой, отмечаем, что когда ваша база перестает быть “простой”, вам может потребоваться дополнительная информация или даже помощь специалистов.

Понимая это противоречие, теоретик реляционной модели К. Дж. Дейт говорит, что “создать нужную структуру базы данных (по крайней мере в простых случаях) зачастую проще, чем строго сформулировать, какой она должна быть”.

Мы попытаемся объяснить вам, как нужно проектировать базы данных на интуитивном уровне, а также кратко обсудим такие методы проектирования, как **нормализация (normalization)** и **моделирование зависимостей (entity-relationship modeling)**.

Мы опишем некоторые ошибки, наиболее часто встречающиеся у начинающих разработчиков баз данных. Разобравшись в них, вы легко поймете, как нужно создавать правильные структуры.

Что же касается формальной стороны методологии проектирования, то большинство экспертов подчеркивают, что они в большей мере руководствуются здравым смыслом, чем какими-либо строгими правилами. Описание реальных объектов и взаимосвязей между ними во многом носит субъективный характер. Часто существует более одного правильного решения конкретной проблемы, кроме того, иногда имеет смысл просто нарушать даже самые общие правила проектирования.

Однако все это вовсе не означает бесполезность теории проектирования баз данных. Понимание основ исключительно важно и для начинающих. По мере приобретения опыта, вы, скорее всего, все чаще и чаще будете обращаться к формальным методологиям, но в то же время не позволите им господствовать над собой.

Начнем. Часто при обсуждении вопросов проектирования реляционных баз данных почти все внимание уделяется применению правил нормализации. В ходе нормализации обеспечивается защита целостности данных путем устранения дублирования данных. В результате таблица, которая первоначально казалась “имеющей смысл”, разбивается на две или более связанных таблиц, которые могут

быть “собраны вместе” с помощью операции объединения. Этот процесс называется **декомпозицией без потерь (non-loss decomposition)** и просто означает разделение таблицы на несколько меньших таблиц без потери информации.

Нормализация наиболее полезна для проверки созданной вами структуры. Вы можете проанализировать свои решения о том, какие столбцы должны быть включены в ту или иную таблицу с точки зрения правил нормализации, убедившись при этом, что не сделали каких-то фатальных ошибок. Понимание основ процесса нормализации также может помочь вам в процессе проектирования базы данных, но оно не является универсальным рецептом при построении базы с нуля.

Итак, как определить, какие столбцы должны располагаться в начале таблицы. Общего правила на этот счет не существует. Однако здесь вам может оказать существенную помощь моделирование зависимостей — анализ сущности данных (в терминах объектов или вещей) и зависимостей между ними (один-к-одному, один-ко-многим, многих-ко-многим).

На практике проектирование базы данных требует хорошего понимания моделируемой вами предметной области, а также знаний в области моделирования зависимостей и нормализации. Проектирование базы данных обычно является итеративным процессом, в ходе которого вы шаг за шагом приближаетесь к требуемому результату, а иногда и удаляете на несколько шагов, переделывая предыдущую работу с учетом появившихся новых потребностей.

Вот примерная последовательность шагов, которые мы рекомендуем вам выполнить в процессе проектирования базы данных.

1. Исследуйте информационную среду, которую вы собираетесь моделировать. Откуда поступает информация и в каком виде? Как она будет вводиться в систему и кто этим будет заниматься? Как часто она изменяется? Какие параметры системы будут наиболее критическими с точки зрения времени реакции на запрос и надежности? Изучите все бумажные материалы, а также информационные файлы и формы, которые используются в вашей организации для хранения и обработки данных. Уточните также, в каком виде информация должна извлекаться из базы данных — в форме отчетов, заказов, статистической информации — и кому она будет предназначаться. В случае коллективного использования базы данных, нужную информацию вы можете собрать, поговорив с сотрудниками вашей организации. Не забудьте посоветоваться с каждым, кто так или иначе будет связан с обработкой данных — их созданием, поддержкой, исполнением запросов, созданием на их основе отчетов и т.д.
2. Создайте список объектов (вещей, которые будут предметом вашей базы данных) вместе с их свойствами и атрибутами. Объекты, скорее всего, имеет смысл собрать в таблицы (каждая строка таблицы будет описывать один объект, например человека, компанию или книгу), свойства объектов будут представлены столбцами таблицы (например, зарплата, адрес компании, стоимость книги). Конечно, сначала можно создать список всех возможных атрибутов, а затем сгруппировать их в объекты, а не начинать с самих объектов. Какой метод выбрать — это, скорее, дело вкуса. В любом случае вы должны ответить на следующие вопросы. Действительно ли выбранные вами атрибуты подходят для описания данного объекта или лучше использовать их с другим объектом? Нужны ли еще дополнительные объекты или другие атрибуты?
3. В ходе работы обязательно записывайте все свои конструктивные решения либо на бумаге, либо с помощью текстового редактора. Разработчики баз данных обычно начинают с чистого листа, постепенно создавая макет таблиц и связей между ними, называемый **структурой данных (data structure)**, или **диаграммой зависимостей между объектами (E-R diagram)**.
4. Предварительно разобравшись с объектами и их атрибутами, убедитесь, что каждый объект имеет атрибут (или группу атрибутов), по которому однозначно можно идентифицировать любую строку в будущей таблице. Этот идентификатор обычно называется первичным ключом. Если такового нет, то для получения искусственного ключа вам следует создать дополнительный столбец.

5. Затем рассмотрите зависимости между объектами. Имеются ли у вас зависимости типа один-ко-многим (один издатель может иметь множество изданных книг, но каждая книга может быть выпущена только одним издателем) или многих-ко-многим (каждый автор может написать несколько книг, и каждая книга может иметь несколько авторов)? Есть ли у вас возможности для объединения связанных таблиц? Для этого служат **внешние ключи (foreign key)**, столбцы в связанных таблицах с совпадающими значениями первичных ключей.
6. Создав вчерне структуру базы данных, проанализируйте ее с точки зрения правил нормализации (они обсуждаются дальше в этой главе) для поиска логических ошибок. Исправьте все отклонения от нормальных форм — или обоснуйте свое решение отказавшись от выполнения ряда правил нормализации в интересах простоты освоения или производительности. Задокументируйте причины таких решений.
7. Теперь вы готовы к непосредственному созданию базы данных и помещению в нее некоторых прототипов данных, используя для этого команды SQL. Мы считаем, что вам обязательно надо будет поэкспериментировать с запросами, изучая получаемые результаты. Возможно, вы захотите выполнить ряд тестов на производительность, чтобы проверить разные технические решения.
8. Оцените свое детище с точки зрения того, удовлетворяют ли вас полученные результаты.

Оставшаяся часть этой главы посвящена рассказу о том, как мы проектировали базу данных *bookbiz*. Это пошаговое, детальное обсуждение поможет вам разобраться в процессе проектирования базы данных.

Что такое “хорошая структура”

Хорошая структура — это, в первую очередь, “прозрачная” структура. Проще говоря, хорошая структура

- максимально упрощает ваше взаимодействие с базой данных;
- гарантирует непротиворечивость данных;
- “выжимает” максимум производительности из вашей системы.

Некоторые факторы, упрощающие понимание базы данных, не имеют строгих технических определений и не являются частью процесса проектирования. Тем не менее широкие таблицы трудно читать и в них сложно разбираться. В то же время разделение данных на целый ряд небольших таблиц усложняет отслеживание взаимосвязей между ними. Выбор подходящего числа столбцов обычно является компромиссом между простотой понимания базы и правилами нормализации.

Хорошо разработанная база данных предотвращает ввод противоречивой информации и случайное удаление данных. Это достигается за счет минимизации ненужного дублирования данных в таблицах и поддержки целостности. Опасности, связанные с противоречивостью данных, детально обсуждаются дальше в этой главе.

Наконец, хорошо разработанная база должна обладать достаточной производительностью. Опять-таки здесь играет большую роль число столбцов в таблицах: выборка данных будет проводиться медленнее, если информация размещена не в одной, а в нескольких таблицах. Однако большие таблицы могут требовать от системы обработки большего количества данных, чем это на самом деле необходимо для выполнения конкретного запроса. Другими словами, количество и размер таблиц существенно влияют на производительность. (Также с точки зрения производительности критическим является выбор столбца, по которому выполняется индексирование и тип индексирования.) Индексирование в большей

мере является вопросом физического проектирования, нежели логического и об-суждается в главах 3 и 10.

Плохая структура базы данных

- приводит к непониманию результатов выполнения запросов;
- повышает риск введения в базу данных противоречивой информации;
- порождает избыточные данные;
- усложняет выполнение изменений структуры созданных ранее и уже запол-ненных данными таблиц.

Не существует идеального решения, полностью удовлетворяющего все требова-ния, предъявляемые при проектировании баз данных. Часто вы чем-то жертвуете, основываясь на требованиях и особенностях приложений, которые будут использо-вать вашу базу данных.

Описание нашей базы данных

В первую очередь, обращаем ваше внимание, что база данных *bookbiz* является исключительно учебной базой данных. Ее основное назначение — обеспечить вас небольшим набором интересных данных, манипулируя которыми вы сможете изу-чить синтаксис и семантику языка SQL.

База данных *bookbiz* содержит информацию о не существующей на самом деле компании, занимающейся издательской деятельностью и имеющей три дочерних издательства. В ней представлены данные, которые могут потребо-ваться редакторам, менеджерам и другим сотрудникам компании — информа-ция о книгах, их авторах, редакторах, о финансовом состоянии компании. На их основе можно получать самые разнообразные отчеты, например о текущих продажах, сравнивать книги разных издательств, узнать, какие редакторы ра-ботали с какими авторами и т.д.

Пользователи базы данных *bookbiz* могут задавать самые разные запросы.

- Кто из авторов проживает в Калифорнии?
- Какие книги стоят больше \$9.95?
- Кто написал самое большее количество книг?
- Чем мы обязаны автору книги *Life Without Fear (Жизнь без страха)*?
- Какова средняя стоимость книг по психологии?
- Как повлияет на авторский гонорар увеличение на 10% стоимости книг по кулинарии?
- Как продаются книги по компьютерам?

Разрабатывая базу данных, даже не пытайтесь представить себе, какой из за-просов будет наиболее важным для будущих пользователей вашего детища. Иссле-дуйте их потребности на основе имеющегося у вас набора данных и личных бесед с пользователями.

Важной областью исследований являются бизнес-правила и политика вашей организации, которые могут оказывать влияние на данные. База данных *bookbiz* учитывает следующие особенности:

- автор может написать несколько книг;
- книга может быть написана несколькими авторами;
- порядок фамилий авторов на первой странице является критической ин-формацией, так как влияет на получаемый ими гонорар;
- редактор может работать над несколькими книгами, и в каждой книге может быть несколько редакторов;
- в заказе на покупку может быть перечислено несколько книг.

ДАнные И ВЗАИМОСВЯЗИ

Мы начнем рассмотрение структуры базы данных *bookbiz* с построения простой модели взаимосвязи объектов. В самых общих чертах такое моделирование (иногда называемое объектным моделированием) подразумевает определение

- объектов, информация о которых будет содержаться в базе данных;
- свойств этих объектов;
- взаимосвязей между ними.

Объекты

Сначала рассмотрим объекты, на основе которых будет построена база данных *bookbiz*. Без учета финансовой информации список объектов будет выглядеть так:

- авторы, книги которых опубликованы компанией;
- сами книги;
- редакторы, работающие на компанию;
- издательства, которыми владеет компания.

Каждый пункт в этом списке описывает объект, существующий независимо от других объектов в мире базы *bookbiz*. Каждый такой объект представляется отдельной таблицей. (Ряд других объектов также представлен в этой базе данных отдельными таблицами, но пока не будем забегать вперед.)

Каждый из этих объектов обладает собственными свойствами, которые также записаны в базе данных. Среди них

название книги
стоимость книги
дата выхода книги
имя автора
адрес автора
номер телефона автора
имя редактора
адрес редактора
номер телефона редактора
название издательства
адрес издательства

Каждый пункт этого списка описывает отдельное свойство или атрибут рассматриваемого объекта (автор, книга, редактор или издательство) и является потенциальным столбцом в базе данных. Названия столбцов должны быть предельно ясными (назначение столбца должно быть понятно из его названия) и краткими (чтобы упростить ввод их названий и уменьшить их ширину).

Создание списка объектов и их свойств должно помочь вам решить, какие таблицы и столбцы нужно включить в базу данных. В результате вы можете получить, например, следующий макет базы:

Таблица titles

name	price	pubdate
------	-------	---------

Таблица authors

name	address	phone
------	---------	-------

Таблица editors

name	address	phone
------	---------	-------

Таблица publishers

name	address
------	---------

Другой способ представления информации используется в диаграммах взаимосвязей между объектами (рис. 2.1). Обычно каждая таблица представляется на такой диаграмме в виде прямоугольника, содержащего в себе, кроме того, названия полей.

Этот макет из четырех таблиц, каждая из которых имеет по несколько столбцов, является первым шагом на пути построения реальной базы данных. Вы можете представить себе, что каждая таблица будет содержать множество строк данных. Каждая строка в таблице описывает вхождение (*occurrence*) или экземпляр (*instance*) объекта — отдельную книгу, автора, редактора или издательство.

Одна из задач проектирования базы данных состоит в обеспечении способа идентификации различных объектов, другими словами, система должна уметь отличать друг от друга отдельные строки таблицы. Как мы уже отмечали, строки отличаются друг от друга по значению первичного ключа таблицы. Неформально, первичный ключ — это столбец или набор столбцов, однозначно определяющий строку.

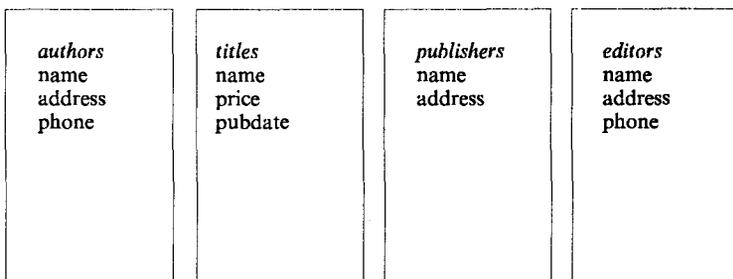


Рис. 2.1. Макет базы данных bookbiz

Первичные ключи. Что будет первичными ключами для каждой из этих таблиц? Рассмотрим таблицу *authors*. Среди ее столбцов очевидным кандидатом на первичный ключ является *name*. Авторов всегда можно различить по имени. Однако использовать этот столбец в качестве первичного ключа достаточно проблематично по нескольким причинам. Во первых, значения в столбце *name* состоят из имени и фамилии автора. Объединение в одном столбце имени и фамилии обычно является не слишком дальновидным поступком, хотя бы потому, что в некоторых системах бывает трудно (а иногда просто невозможно) выполнить алфавитную сортировку по фамилиям. Таким образом, первым необходимым изменением (рис. 2.2) будет разбиение столбца *name* на два отдельных столбца (как в таблице *authors*, так и в таблице *editors*).

Теперь, возвращаясь к вопросу о выборе первичного ключа в таблице *authors*, для этих целей можно использовать комбинацию столбцов *au_lname* и *au_fname*. Эта комбинация будет удовлетворять нас, пока таблица не разрастется до такой степени, что в ней появятся авторы с одинаковыми именами и фамилиями. Например, как только в таблице появятся две Mary Smiths, комбинация полей *au_lname* и *au_fname* перестанет однозначно идентифицировать каждого автора.

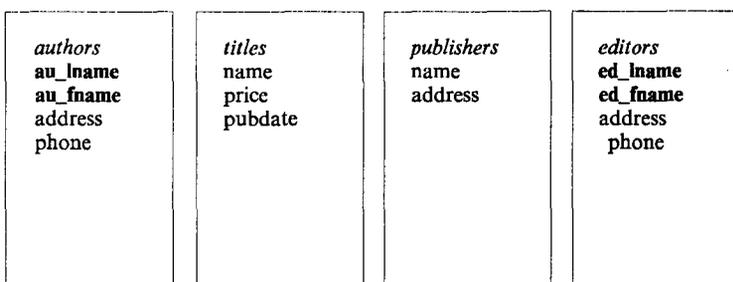


Рис. 2.2. Разделение имен авторов и редакторов на два столбца

Другая проблема, возникающая при использовании имен в качестве уникальных идентификаторов, связана с их частым неправильным вводом. Клерк, принимающий информацию по телефону и сразу заносящий ее в базу данных, может даже не задуматься над тем, как правильно пишется имя и фамилия — Anne Ringer или Ann Ringer. Та же проблема возникает и с названиями компаний. Например, название одной телефонной компании может иметь несколько вариантов написания — AT&T, A.T. and T., Ma Bell и т.д. С точки зрения компьютера, это будут разные компании.

Поэтому хорошим решением обычно является создание специального столбца, который и будет использоваться в качестве первичного ключа. Реальными примерами таких уникальных идентификаторов могут быть код предприятия, номер лицензии, заказа, студенческого билета и т.д.

В таблицах *authors* и *editors* мы использовали для этих целей номера карточек социального страхования. В таблицах *titles* и *publishers* используются произвольные идентификационные номера. Чтобы показать, какие столбцы являются первичными ключами, мы подчеркиваем их (рис. 2.3).

Выбор первичных ключей является одним из основных шагов при проектировании базы данных. Несмотря на их важность, ранние версии SQL не поддерживали определение первичных ключей. ANSI-стандарт SQL 1992 года, принятый сегодня большинством производителей, поддерживает предложение PRIMARY KEY в операторе CREATE TABLE (подробнее об этом — в главе 3). Кроме того, производители разрабатывают свои собственные методы для оперирования этим важным понятием.

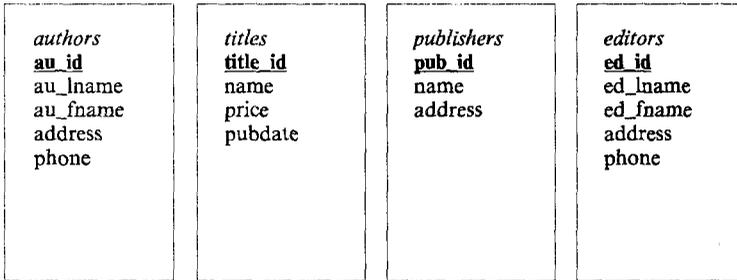


Рис. 2.3. Определение первичных ключей

Отношение один-ко-многим

К этому моменту мы имеем готовые структуры четырех таблиц базы данных *bookbiz*: *authors*, *titles*, *editors* и *publishers*. Для каждой таблицы описаны атрибуты и определены первичные ключи.

Тем не менее мы пока не определили никаких зависимостей между данными. Пока остается непонятным, как, например, связаны между собой книги и конкретные издательства.

Связь между книгами и издательствами может описываться отношением один-ко-многим: каждая книга имеет единственного издателя, в то время как издатель может выпустить несколько книг. Отношение один-ко-многим часто записывается в виде 1:N.

Как реализовать это отношение? Первым порывом может быть желание добавить столбец, как показано на рис. 2.4.

Столбец *title_id* будет являться внешним ключом в таблице *publishers*. Вы можете использовать его для ссылки на определенные строки в таблице *titles* и объединения информации о книгах и издателях. К сожалению, это решение уводит вас в неправильном направлении. Вспомним моделируемое нами отношение — один издатель, много книг — и посмотрим, что произойдет при выходе новой книги. Вы добавите новую строку в таблицу *titles* с названием книги, ценой и т.д.

<u>title_id</u>	title	price	date
BU2075	You Can Combat Computer Stress!	2.99	6/30/85

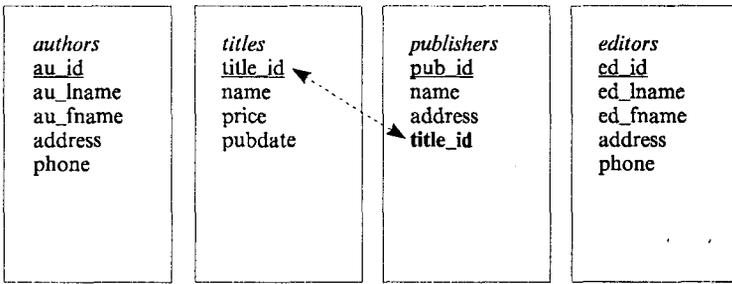


Рис. 2.4. Добавление внешнего ключа

Для каждой новой строки таблицы *titles* вы должны добавить соответствующую строку в таблицу *publishers*. При этом в строке таблицы *publishers* будет повторена уже существующая там информация и добавлен один столбец с новой информацией (*title_id*), отражающей появление новой книги в таблице *titles*.

pub_id	pub_name	address	title_id
0736	New Age Books	1 1st St Boston MA	BU2075

Вспомните, что одной из целей процесса проектирования базы данных является контроль за избыточностью данных, так как избыточность данных увеличивает вероятность появления ошибок. Лучшим решением является добавление внешнего ключа к таблице *titles* (рис. 2.5).

Теперь при выходе новой книги вы просто добавляете строку в таблицу *titles* (со столбцом *pub_id*) и ничего не изменяете в таблице *publishers*, пока компания не открывает новый филиал.

Title_id	title	price	date	pub_id
BU2075	You Can Combat Computer Stress!	2.99	6/30/85	0736

С учетом последних изменений мы получили следующую структуру:

- таблица *publishers* содержит по одной строке для каждого издателя;
- таблица *titles* имеет по одной строке на каждую книгу;
- идентификационный номер издателя повторяется в таблице *titles*, так как издатель может выпустить несколько книг, и при этом обеспечивается минимальная избыточность данных.

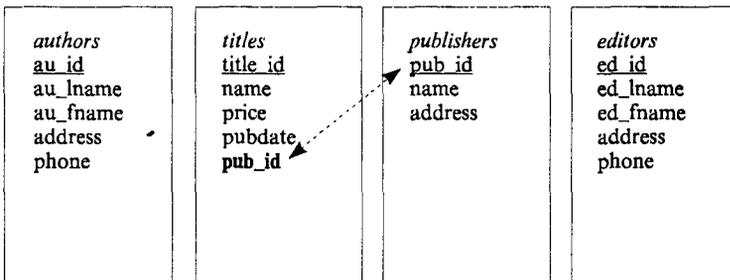


Рис. 2.5. Изменение внешнего ключа

Вы можете использовать логическую связь между столбцами *pub_id* в таблицах *titles* и *publishers* для объединения этих таблиц. Другими словами, структура базы разрабатывалась в расчете на применение пользователями оператора объединения для выборки информации из таблиц *titles* и *publishers* с помощью одного запроса.

В таблице *publishers* столбец *pub_id* является первичным ключом. В таблице *titles* этот столбец выполняет роль внешнего ключа. Таким образом, реляционная модель требует, чтобы отношение один-ко-многим реализовывалось посредством пары первичный ключ—внешний ключ.

Внешние ключи. Так же, как и первичные ключи, внешние ключи имеют важное значение в процессе проектирования базы данных. Неформально, внешний ключ — это столбец или комбинация столбцов в одной таблице, значения которого совпадают со значениями первичного ключа в другой таблице.

При рассмотрении логической взаимосвязи между информацией, содержащейся в первичном и внешнем ключе, может возникнуть несколько вопросов. Что произойдет со столбцом *pub_id* в таблице *titles*, если в таблице *publishers* будет удалена или изменена строка, описывающая издателя? Можно ли ссылаться на описание книги, используя для этого идентификационный номер издателя, не существующего больше в базе данных? Такое действие не имеет под собой логического смысла и к тому же нарушает определение внешнего ключа, в соответствии с которым значения внешнего ключа должны совпадать со значениями первичного ключа, имеющегося где-то в базе данных.

При проектировании базы данных обязательно нужно обеспечить непротиворечивость между первичными и внешними ключами (ссылочную целостность). Например,

- если в таблице изменяется или удаляется идентификационный номер издателя, система должна автоматически выполнять модификации в таблице *titles*, т.е. либо изменять соответствующие значения в столбце *pub_id* таблицы *titles*, либо удалять из нее строки с уже несуществующими в таблице значениями *pub_id*;
- если в базу данных добавляется информация по новой книге, система должна проверить корректность введенного значения в столбце *pub_id* (т.е. что оно присутствует в таблице *publishers*).

Пока просто возьмите это себе на заметку. В следующей главе мы расскажем, как можно обеспечить совместимость первичных и внешних ключей, введя ограничения в предложение REFERENCES оператора CREATE TABLE. Производители коммерческих систем баз данных также обеспечивают специальные методы для управления целостностью, например расширения SQL для процедурного кода, который может выполняться в базе данных (такой код обычно называется процедурами, или триггерами, которые мы рассмотрим в главе 10).

Отношение многих-ко-многим

Определив все отношения вида один-ко-многим в базе данных *bookbiz* и поставив им в соответствие пары первичный ключ—внешний ключ, рассмотрим другие типы отношений. Например, как связаны между собой авторы и книги?

Некоторые книги написаны несколькими авторами, к тому же некоторые авторы выпустили более одной книги. Другими словами, авторы и книги связаны отношением многих-ко-многим (которое часто записывается в виде $N:N$ и иногда называется **соединением (association)**). В соответствии с реляционной теорией, соединения должны представляться отдельными таблицами, т.е. в базе данных *bookbiz* должна быть таблица для авторов, таблица для книг и таблица, описывающая связи между ними (рис. 2.6).

Таблица *titleauthors* описывает отношение типа многих-ко-многим между авторами и книгами. Это такая же базовая таблица, как *titles* и *authors*, но вместо объектов она описывает связи. Если пользователю базы данных *bookbiz* потребуется информация о том, кто написал какую книгу, он или она создаст запрос на объединение, в котором для связи между таблицами *titles* и *authors* будет использована таблица *titleauthors*.

Таблицы *titleauthors* и *titles* объединяются по столбцам *title_id*, *titleauthors* и *authors* — по столбцам *au_id*. Другими словами, *title_id* в таблице *titleauthors* является внешним ключом, соответствующим первичному ключу *title_id* в таблице *titles*, а *au_id* в таблице *titleauthors* является внешним ключом, соответствующим первичному ключу *au_id* в таблице *authors*. По общему принципу, сформулированному К. Дж. Дейтом, “в реляционной модели участники соединения определяются внешними ключами, образующими таблицу, представляющую это соединение”.

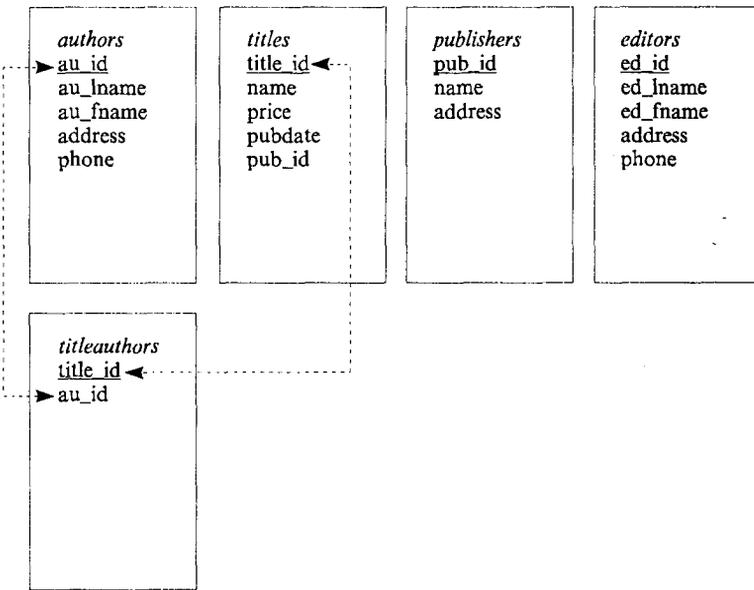


Рис. 2.6. Таблица соединения для отношения многих-ко-многим

Что является первичным ключом в таблице *titleauthors*? Идентификатор автора и идентификатор книги не идентифицируют однозначно строки в таблице *titleauthors*. В ее столбцах повторяются идентификаторы книг, имеющих больше одного автора, и идентификаторы авторов, написавших несколько книг. Тем не менее уникальной является комбинация идентификатор автора—идентификатор книги. Как говорит Дейт: “Для данного соединения, как правило, комбинация всех внешних ключей его участников имеет свойство уникальности”. Таким образом, первичным ключом в таблице *titleauthors* является комбинация *title_id* и *au_id*.

Таблицы *editors* и *titles* имеют аналогичные взаимосвязи. Редактор может работать более чем над одной книгой, а у книги может быть несколько редакторов. Это отношение многих-ко-многим также описывается связывающей таблицей.

Отношение один-к-одному

Бросим последний взгляд на наши объекты. Если вы обнаружите между таблицами отношение типа 1:1, то, скорее всего, их лучше объединить в одну таблицу. Основной причиной использования отношения 1:1 является увеличение скорости исполнения запросов. Например, если вы редко используете какую либо информацию о книге (заметки об авторских правах или список измененных страниц), ее можно поместить в отдельную таблицу, чтобы не обрабатывать ее в общих запросах. В принципе, пока вы не изучите особенности данных, в процессе проектирования базы данных лучше избегать использования отношений вида один-к-одному.

Последние замечания к объектному подходу

Объектное моделирование — значительно более сложная задача, чем можно себе представить на основе кратко описанной здесь процедуры. Тем не менее рассмотренный выше подход поможет вам создавать хорошие базы данных, которые только останется проверить с помощью рассматриваемых далее правил нормализации. Но прежде освежим в памяти основные действия, которые необходимо выполнить в процессе проектирования базы данных.

1. Представить каждый независимый объект (книга, автор, издатель, редактор, наниматель, студент, компания и т.д.) в виде строк базовых таблиц.
2. Представить свойства объектов (адреса авторов, стоимости книг и т.п.) в виде столбцов таблиц.

3. Убедиться, что в каждой таблице имеется первичный ключ. Этим ключом может быть уже существующий столбец или искусственно добавленный вами (например, порядковый номер или номер карточки социального страхования), или же комбинация двух и более столбцов.
4. Найти все отношения типа один-ко-многим между таблицами. Проверить соответствия между внешними и первичными ключами. Установить ограничения на целостность, связанные с каждым внешним ключом.
5. Представить каждое отношение типа многих-ко-многим (соединение) в виде “соединительной” таблицы, состоящей из внешних ключей, связанных с таблицами объектов. Первичным ключом для соединительной таблицы обычно является комбинация, составленная из этих внешних ключей.

После выполнения каждого шага еще раз подумайте о своих потребностях и возможностях. Учили ли вы бизнес-правила? Сможете ли вы получить необходимую информацию? Такой взгляд на базу данных *bookbiz* выявляет ряд ограничений. В ней не предусмотрена возможность записи информации о позиции автора в списке авторов книги и принципах разделения гонораров. Кроме того, в ней отсутствуют описания контрактов. Также в структуре базы не учтены вопросы, связанные с продажами книг. С учетом этого мы можем модифицировать диаграмму зависимостей между объектами, как показано на рис. 2.7.

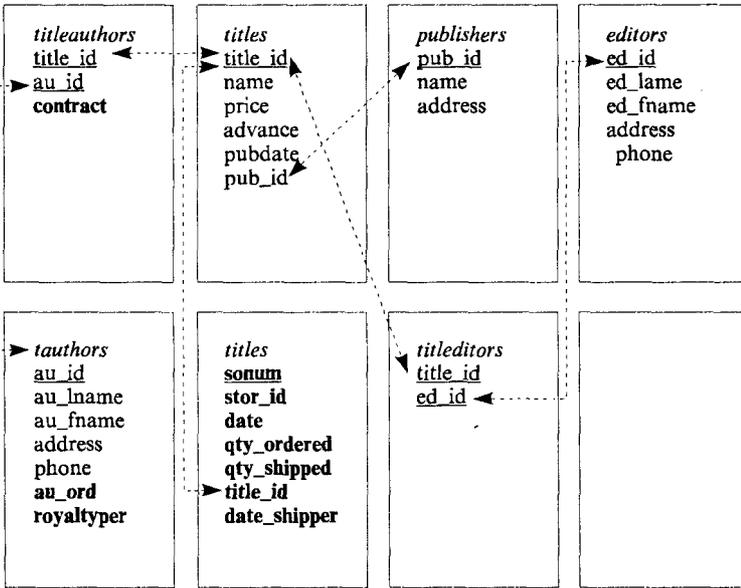


Рис. 2.7. Измененная диаграмма

РУКОВОДСТВО ПО НОРМАЛИЗАЦИИ

Вообще говоря, руководство по нормализации — это набор стандартов проектирования данных, называемых **нормальными формами (normal form)**. Общепринятыми считаются пять нормальных форм, хотя их было предложено значительно больше. Создание таблиц в соответствии с этими стандартами называется нормализацией.

Нормальные формы изменяются в порядке от первой до пятой. Каждая последующая форма удовлетворяет требованиям предыдущей. Если вы следуете первому правилу нормализации, ваши данные будут представлены в первой нормальной форме. Если ваши данные удовлетворяют третьему правилу нормализации, они будут находиться в третьей нормальной форме (а также в первой и второй формах).

Выполнение правил нормализации обычно приводит к разделению таблиц на две или больше таблиц с меньшим числом столбцов, выделению отношений пер-

вичный ключ—внешний ключ в меньшие таблицы, которые снова могут быть соединены с помощью операции объединения.

Одним из основных результатов разделения таблиц в соответствии с правилами нормализации является уменьшение избыточности данных в таблицах. При этом вас не должно смущать наличие в базе одинаковых столбцов первичных и внешних ключей. Такое преднамеренное дублирование — это не то же самое, что избыточность. На самом деле поддержка непротиворечивости между первичными и внешними ключами связана с понятием целостности данных.

Правила нормализации, подобно принципам объектного моделирования, развивались в рамках теории баз данных. Большинство разработчиков баз данных признают, что представление данных в третьей и четвертой нормальных формах полностью удовлетворяет все их потребности.

Первая нормальная форма

Первая нормальная форма требует, чтобы на любом пересечении строки и столбца находилось единственное значение, которое должно быть атомарным. Кроме того, в таблице, удовлетворяющей первой нормальной форме, не должно быть повторяющихся групп.

Рассмотрим на примере новой таблицы *sales*, как составить накладную сразу на несколько книг (рис. 2.8).

Bookbiz Sales Order From				
Oredr #14 Store 7131 Date:5/29/87				
Item #	Title #	Ordered #	Shipped	Ship date
1.	PS1372	20	20	May 29 1987
2.	PS2106	25	25	Apr 29 1987
3.	PS3333	15	10	May 29 1987
4.	PS7777	25	25	Jun 13 1987
5.				
6.				

Рис. 2.8. Накладная

sonum	stor_id	date	title_id	Непрямоугольная таблица		
13	7066	5/24/87	PC8888			
14	7131	5/29/87	PS1372	PS2106	PS3333	PS7777
15	7067	6/15/87	TC3218	TC4203	TC7777	

Рис. 2.9. Таблица должна быть прямоугольной

Вы не можете записать несколько идентификаторов книг в одном поле, так как это противоречило бы первой нормальной форме. Добавление столбцов типа *title1*, *title2* просто маскирует появление в таблице повторяющихся групп и тоже не решает проблемы. Кроме того, это не лучшее решение и с практической точки зрения, так как при включении в накладную новой книги вам придется переделывать всю таблицу, включая в нее столбец *title3*. Так как в первой нормальной форме все данные должны быть представлены в виде прямоугольных таблиц, также не допускается и использование множественных значений (рис. 2.9).

Чтобы избежать повторения столбцов, мы используем **главную таблицу (master table)** — *sales* и **вспомогательную таблицу (detail table)** — *salesdetail*, которая поддерживает информацию по отдельным позициям накладной (рис. 2.10). Обрати-

те внимание, что объектное моделирование приводит к тем же результатам, так как в этом случае мы имеем отношение один-ко-многим (одна накладная—много позиций).

Занимаясь поиском повторяющихся полей, разбейте на компоненты все составные столбцы: например столбец *address* нужно разбить на два столбца — *city* и *state*.

Вторая нормальная форма

Второе правило нормализации требует, чтобы любой неключевой столбец зависел от всего первичного ключа. Следовательно, таблица не должна содержать неключевых столбцов, зависящих только от части составного первичного ключа. Представление таблицы во второй нормальной форме требует, чтобы все столбцы, не являющиеся первичными ключами (столбцы, описывающие объект, но однозначно не идентифицирующие его), зависели от всего первичного ключа, а не от его отдельных компонентов.

В качестве примера рассмотрим столбец *contract* в таблице *titleauthors*. Зависит ли он исключительно от комбинации автор-название? Если каждый автор заключает отдельный контракт, тогда — да, но ситуация изменится, если компания подпишет контракт только при достижении согласия со всеми авторами.

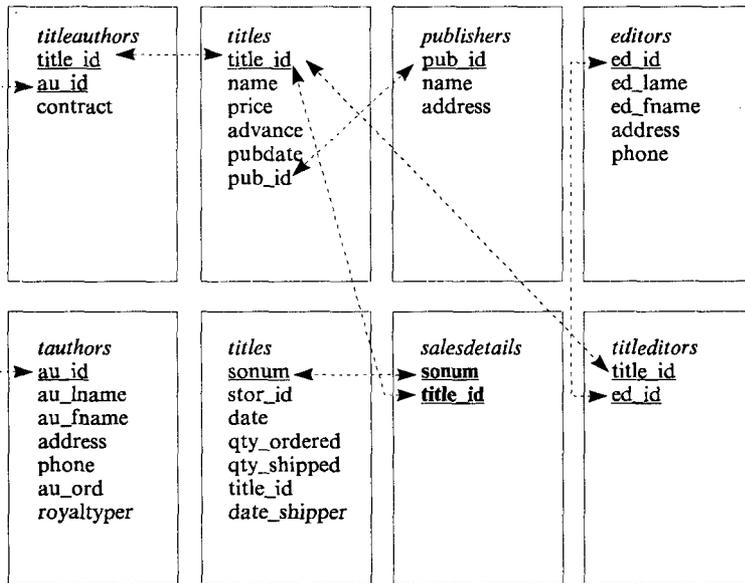


Рис. 2.10. Разбиение таблицы sales

В этом случае контракт определяется книгой, а не отдельными авторами, и вы должны переместить столбец *contract* в таблицу *titles* (рис. 2.11). Этот пример иллюстрирует сложность процесса проектирования базы данных: ваши решения часто могут зависеть от конкретных принципов организации работы в вашей организации.

Суммируя вышесказанное, вторая нормальная форма требует, чтобы ни один неключевой столбец не зависел только от части первичного ключа. Это правило относится к случаю, когда первичный ключ образован из нескольких столбцов, и неприменимо, когда первичный ключ образован только из одного столбца.

Третья нормальная форма

Третья нормальная форма повышает требования второй нормальной формы: она не ограничивается составными первичными ключами. Третья нормальная форма требует, чтобы ни один неключевой столбец не зависел от другого неключевого столбца. Любой неключевой столбец должен зависеть только от столбца первичного ключа.

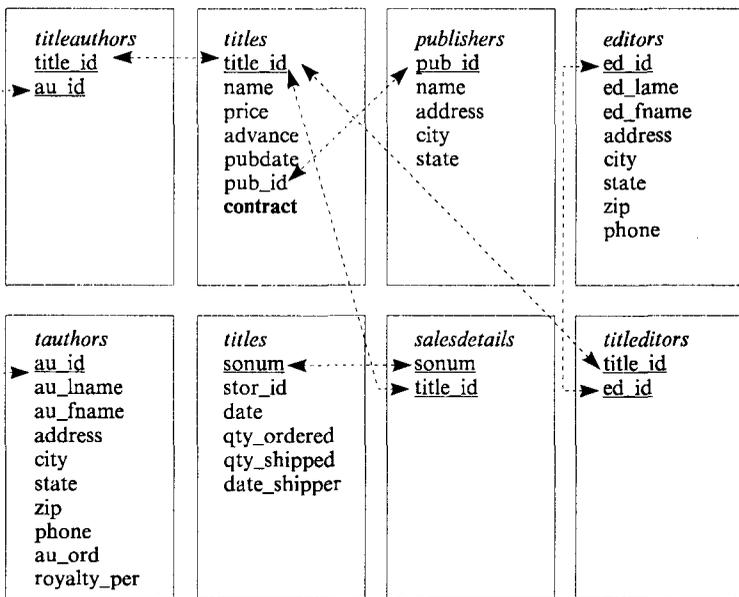


Рис. 2.11. Диаграмма зависимостей между объектами базы данных bookbiz после преобразования данных во вторую нормальную форму

В таблице *authors* первичным ключом является *au_id*. Проверив каждый столбец, вы обнаружите, что столбец *au_ord* (позиция автора в списке авторов книги) не зависит от столбца авторов (*au_id*), так как у каждого автора может быть несколько книг, и в каждой из них он может занимать разные позиции (первую, вторую или третью). На самом деле позиция определяется отношением автор—книга. Так же обстоят дела и со столбцом *royaltyper*. Оба этих столбца переносятся в таблицу *titleauthors*.

Столбцы *qty_ordered* и *qty_shipped* из таблицы *sales* также иллюстрируют этот принцип. Они относятся к отдельным пунктам, а не ко всей накладной, и, следовательно, должны быть перенесены в таблицу *salesdetails*.

Столбец *data_shipped* более загадочен.

- Если заказ выполняется только при наличии всех книг, столбец *data_shipped* относится ко всей накладной и поэтому должен быть перемещен в таблицу *sales*.
- Если же заказ выполняется по мере выполнения отдельных пунктов, этот столбец должен принадлежать таблице *salesdetails*.

Так как книги могут быть доступны только после выхода из печати, мы придерживаемся второй модели. Измененная диаграмма представлена на рис. 2.12.

Рассматривая структуру этих таблиц, вы увидите, что они удовлетворяют как второй, так и третьей нормальной форме. Они удовлетворяют второй нормальной форме, так как все неключевые столбцы зависят от всего первичного ключа, и третьей нормальной форме, так как все неключевые столбцы не зависят друг от друга. Другими словами, любой неключевой столбец зависит от ключа, всего ключа и ничего, кроме ключа.

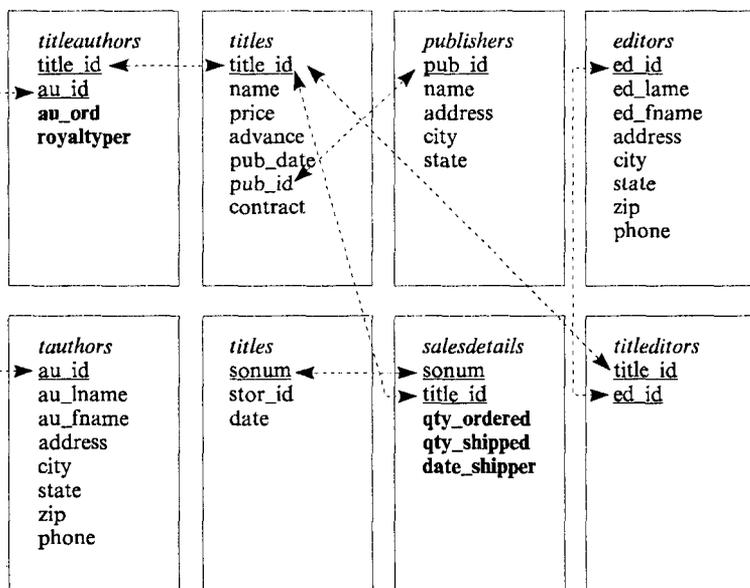


Рис. 2.12. Диаграмма данных в третьей нормальной форме

Четвертая и пятая нормальные формы

Четвертая нормальная форма запрещает независимые отношения типа один-ко-многим между ключевыми и неключевыми столбцами. В качестве примера рассмотрим несколько надуманный пример: каждый автор может иметь несколько машин и несколько домашних животных, но между машинами и животными нет абсолютно никакой связи, хотя они естественным образом связаны с автором.

<u>Au_lname</u>	car	pet
Ringer	1987 Chevy Nova	Rover
Ringer	1994 Volvo Station Wagon	
Bennet	1990 VW Rabbit	Spot
Green		Valiant
Green	1989 Toyota Corolla	Fluffy
Green		Sam

Помещение этой разнородной информации в одну таблицу может привести к появлению в ней пустых мест, так как домашних животных может быть больше, чем машин (как в случае с Green) или наоборот (как в случае с Ringer). Удаление данных о машинах или животных также может привести к появлению пустых мест.

Проблема здесь состоит в кажущемся существовании зависимости между машинами и домашними животными, так как эти данные размещаются рядом в одной строке. Лучше было бы поместить их в разные таблицы и связать с авторами посредством внешнего ключа.

car	au_id
1987 Chevy Nova	998-72-3567
1994 Volvo Station Wagon	998-72-3567
1990 VW Rabbit	409-56-7008
1989 Toyota Corolla	213-46-8915

pet	au_id
Rover	998-72-3567
Spot	409-56-7008
Valiant	213-46-8915
Fluffy	213-46-8915
Sam	213-46-8915

Пятая нормальная форма доводит весь процесс нормализации до логического конца, *разбивая таблицы на минимально возможные части для устранения в них всей избыточности данных*. Нормализованные таким образом таблицы обычно содержат минимальное количество информации, помимо первичного ключа. Например:

```

titles table
title_id          title

authors table
au_id            au_lname      au_fname

authors&titles table
au_id            title_id

prices table
title_id          price

advances table
title_id          advance

pets table
pet              au_id

```

Преимуществом преобразования базы данных в пятую нормальную форму является возможность управления целостностью. Поскольку при этом любой фрагмент неключевых данных (данных, не являющихся первичным или внешним ключом) встречается в базе данных только один раз, не возникает никаких проблем при их обновлении. Если, например, изменяется стоимость книги, соответствующие поправки нужно внести только в таблицу *prices* и не надо просматривать остальные таблицы на предмет поиска и изменения в них значения соответствующего поля *price*.

Однако, поскольку каждая таблица в пятой нормальной форме имеет минимальное число столбцов, вы должны дублировать в них одни и те же ключи, обеспечивая возможности для объединения таблиц и получения полезной информации.

Изменение значения единственного ключа (например, *title_id*) уже является очень серьезной проблемой. Вы должны найти все вхождения этого значения в вашей базе данных и внести соответствующие изменения. К счастью, столбцы первичных ключей обычно изменяются значительно реже, чем неключевые.

Мораль сей басни такова: нужно добиваться равновесия между избыточностью данных и избыточностью ключей.

ОБЗОР БАЗЫ ДАННЫХ

Давайте вспомним процесс создания базы данных *bookbiz*. Мы начали с четырех таблиц: *authors*, *titles*, *publishers* и *editors*. Их создание было интуитивно ясно, к тому же, в соответствии с объектным подходом, независимые объекты должны представляться отдельными базовыми таблицами. Кроме того, в результате моделирования зависимостей между объектами мы построили еще две базовые таблицы —

одну для связи таблиц *titles* и *authors* (*titleauthors*), другую — для связи таблиц *titles* и *editors* (*titleditors*). Позднее мы добавили таблицы *sales* и *salesdetails*.

К этому моменту мы не рассмотрели единственную таблицу — *roysched*, которая определяет авторские гонорары в виде зависимости от количества проданных книг. Таблица *roysched* является **поисковой таблицей** (**lookup table**), используемой исключительно для справочных целей. Ее данные модифицируются только в случае изменений условий контракта или выхода новой книги.

Таким образом, мы создали девять таблиц.

Перед тем как углубиться в детали базы данных *bookbiz*, давайте рассмотрим диаграмму с конечной структурой данных (рис. 2.13). Каждый прямоугольник на диаграмме представляет таблицу и содержит в себе название таблицы и названия полей. Линии между прямоугольниками представляют связи между таблицами, указывая на возможные объединения (возможны и другие объединения, на диаграмме показаны только запланированные нами объединения).

Символы *I* и *N* на концах линии между таблицами *titles* и *publishers* описывают тип отношения между этими таблицами — один-ко-многим. Издатель может выпустить несколько книг, но каждая книга может иметь только одного издателя.

Последние замечания о базе данных *bookbiz*

Чтобы лучше усвоить материал, на котором будет построено большинство следующих примеров *Практического руководства по SQL*, давайте более внимательно рассмотрим девять таблиц базы данных *bookbiz*.

База данных *bookbiz* содержит информацию о деятельности трех филиалов издательской компании, и так как они не являются независимыми, то используют одну и ту же базу данных.

Таблица *publishers* содержит информацию о трех издательствах: их идентификационные номера, названия и адреса.

Информация о каждом авторе, имеющем контракт с издателем, содержится в таблице *authors*: номер карточки социального страхования, имя, фамилия и адресные данные.

Аналогичную информацию о каждом редакторе содержит и таблица *editors*. Кроме того, в ней имеется дополнительный столбец, описывающий вид выполняемой редактором работы (подбор информации или управление всем проектом).

По всем вышедшим и готовящимся к печати книгам таблица *titles* содержит следующую информацию: идентификационный номер, название, тема, идентификационный номер издателя, стоимость, расходы, количество проданных экземпляров, состояние контракта, дополнительные заметки и дата выхода. Числа в столбце *ytd_sales* должны изменяться по мере увеличения количества проданных книг одним из следующих способов.

- С помощью команд модификации данных.
- Из приложений, которые будут автоматически изменять значения в столбце *ytd_sales*, как только будут вводиться новые данные в таблицу *salesdetails*.
- Используя SQL для определения триггеров, выполняющих автоматическое обновление. (Триггеры, не поддерживаемые стандартом ISO-ANSI 1992 года, входят как расширения во многие системы управления реляционными базами данных. Подробнее об этом вы узнаете в главе 10.)

Книги и авторы представлены в разных таблицах, но могут быть связаны с помощью третьей таблицы — *titleauthors*. Для каждой книги таблица *titleauthors* содержит строку с описанием идентификатора книги, идентификатора автора, позиции автора в списке авторов книги и информацию по разделению гонорара. Таблица *titleditors* аналогично связывает книги с их редакторами. Кроме того, она описывает порядок редактирования, т.е. можно узнать, кто был первым или последним редактором.

Таблица *roysched* описывает зависимость между количеством проданных книг и размером авторского гонорара. Гонорар составляет определенную часть суммы, полученной за проданные книги.

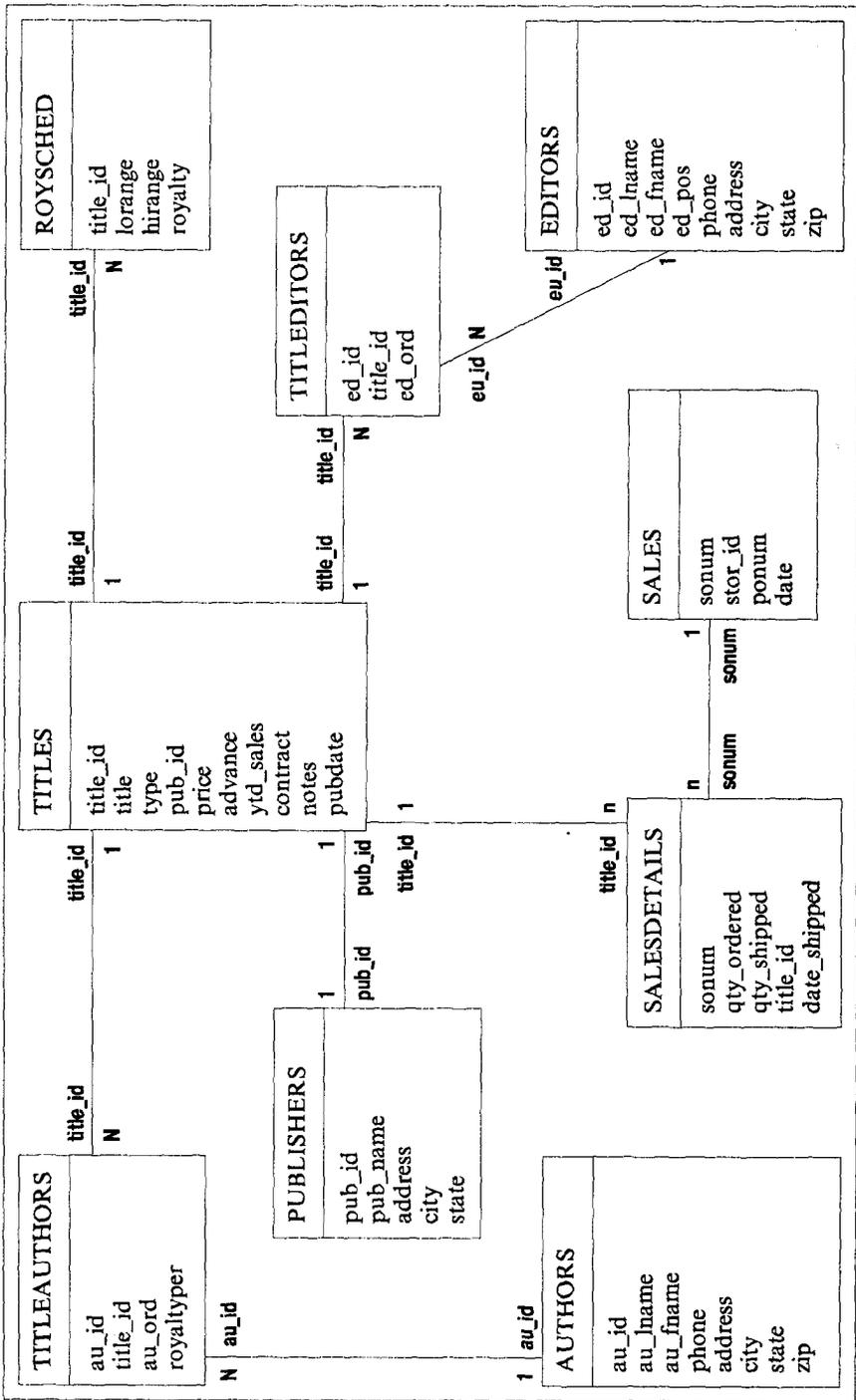


Рис. 2.13. База данных bookbiz

Таблица *sales* содержит общую информацию о заказах, полученных от книжных магазинов: номер квитанции на продажу (получаемый от издателя), идентификатор магазина, номер заказа на покупку (получаемый от книжного магазина) и дату выполнения заказа.

Таблица *salesdetails* содержит информацию о каждом пункте заказа на покупку (предполагая, что сразу могут быть заказаны несколько книг): название, количество заказанных книг, количество отправленных книг и дата отправления.

Конечно, реальная база данных может быть значительно сложнее и содержать таблицы для описания магазинов, дистрибьюторов, производственных расходов, посредников и т.д. (В качестве тренировки вы можете сами добавить некоторые из этих таблиц и решить, как они будут связаны с уже существующими таблицами в базе данных *bookbiz*. Тем не менее этих девяти таблиц вполне достаточно для изучения языка SQL и они будут использоваться в примерах этой книги.)

Проверка структуры базы данных

Разработав структуру базы данных, вы должны создать таблицы и заполнить их некоторыми данными. (Эта процедура подробно описывается в следующей главе.) Затем вы должны проверить разработанную структуру, запуская на выполнение запросы и изменяя данные. Эти тесты могут выявить недочеты в исходной структуре данных.

При проектировании базы данных нельзя полагаться исключительно на теорию. Перед тем как переходить к серьезным действиям, особенно если вы планируете использовать для обработки данных сложные приложения, поэкспериментируйте с прототипом базы данных. Это сэкономит вам сотни часов бесполезного программирования.

Рассмотрение других понятий из области баз данных

Перед практическим использованием созданной базы данных вам потребуется рассмотреть еще несколько важных вопросов: индексирование, безопасность и целостность.

Выбор столбца для индексирования и типа индексирования является предметом обсуждения главы 3.

Вопросы безопасности описаны в главе 10.

Обеспечение целостности — это одна из главных задач в процессе проектирования базы данных. Вы должны быть уверены, что изменения в одной части данных распространяются на все копии этих данных по всей базе. На примере использования базы данных *bookbiz* можно увидеть, к чему приводит неправильный ввод идентификатора автора. Если вы измените его только в таблице *authors*, то больше не сможете получить список книг, написанных данных автором, так как идентификатор автора в таблице *titleauthors* больше не совпадает с идентификатором этого же автора в таблице *authors*. Чтобы получить доступ к информации таблицы *titleauthors*, вы должны обеспечить одновременное изменение идентификатора автора в таблицах *authors* и *titleauthors*.

Мы надеемся, что вы учтете эти замечания при разработке собственных баз данных. В следующей главе вы узнаете, как обеспечивается целостность на SQL 92. Другие методы, не поддерживаемые всеми производителями, обсуждаются в главе 10.

Реализация структуры

Итак, вы изучили логическую структуру базы данных *bookbiz*. Теперь вы увидите, как нормализованная диаграмма зависимостей между объектами переводится в команды SQL.

Создание и заполнение базы данных

Создав структуру базы данных на бумаге, как это описывалось в предыдущей главе, вы готовы воплотить ее в жизнь. С помощью команды SQL CREATE вы можете определить в своей системе управления реляционными базами данных имя, структуру и другие характеристики реальной базы данных и ее объектов. А поместить информацию в базу данных вы можете с помощью команды INSERT.

Если вы используете прилагаемый к этой книге компакт-диск с Sybase SQL Anywhere, вам не нужно вводить многочисленные команды CREATE и INSERT: база данных *bookbiz* для вас уже создана. Просто запустите соответствующую программу на своем ПК. Sybase SQL Anywhere предоставляет вам все возможности для выполнения запросов. Единственное, вы не сможете создавать новые объекты и изменять существующие данные (эта версия SQL Anywhere предназначена исключительно для просмотра данных).

На компакт-диске также содержится программный код (набор команд SQL) для создания и заполнения базы данных *bookbiz*. Вы можете использовать этот код, если захотите воспроизвести базу данных *bookbiz* на своей СУБД или на полноценной версии SQL Anywhere. Возможно, вам придется несколько подправить этот код в соответствии с правилами конкретного диалекта SQL.

Полная структура базы данных *bookbiz* и программный код для ее создания описаны в Приложении Г.

СИНТАКСИС SQL

Это первая глава, читая которую, вы будете сидеть за терминалом и вводить конкретные команды. Поэтому сейчас пришло время познакомиться с соглашениями, используемыми в этой книге для представления команд SQL и их синтаксиса.

Пример реальной команды SQL представлен на рис. 3.1.

```
select pub_id, pub_name, address, city, state
from publishers
where pub_id = '0736'
```

← Все слова набраны
строчными буквами

Рис. 3.1. Пример команды

Синтаксис оператора сродни шаблону, который четко определяет, что требуется (допустимо) для данного оператора (рис. 3.2).

Так как в записи операторов может заключаться большое количество информации, мы будем сопровождать их конкретными примерами на SQL.

При описании синтаксиса SQL используются следующие соглашения.

- Хотя SQL является языком с **необусловленными формами (free-form language)**, не ограничивая тем самым количество слов в одной строке и места разрывов строк, в примерах этой книги каждый оператор обычно начинается с новой строки. Длинные и сложные предложения разбиваются на несколько строк, выделяемых отступом.
- Слова и фразы, которые вы должны подставлять в операторы, всегда записываются строчными буквами. На рис. 3.2 под словами *список_выбора*, *список_таблиц* и *условия* подразумеваются конкретные значения (константы, выражения и идентификаторы (**identifier**)) — имена баз данных, таблиц или

других объектов баз данных), которые используются в командах SQL. Уточните в документации по своей СУБД правила записи идентификаторов: их допустимую минимальную и максимальную длину, недопустимые символы (обычно пробелы и точки) и различия между регистрами (обычно таблица *AUTHORS* — это совсем не то же самое, что таблица *authors*).



Рис. 3.2. Пример синтаксиса оператора

- Ключевые слова и операторы SQL всегда записываются прописными буквами, хотя в большинстве версий SQL вы можете вводить их в любом регистре. Так, в примере на рис 3.2 ключевыми словами являются SELECT, FROM и WHERE, которые можно вводить в любом регистре. Это означает, что слова SELECT, select и SeLeCt описывают один и тот же оператор SQL.
- Фигурные скобки ({ }) вокруг слов или фраз означают, что вы *должны* выбрать по крайней мере одну из заключенных в них опций. Если опции разделены вертикальной чертой (|), вы можете использовать только одну из них, если опции разделены запятой (,) — одну или несколько.
- Квадратные скобки ([]) означают, что заключенные в них опции необязательны. Если, кроме того, опции разделены вертикальной чертой (|), вы можете либо вообще их не использовать, либо использовать только одну из них. Если же опции разделены запятой (,), вы можете не использовать ни одну из них, использовать одну или несколько. На рис. 3.3 показано, как работают вместе фигурные и квадратные скобки, вертикальные линии и запятые.

Фигурные и квадратные скобки не являются частью оператора, так что их не нужно набирать при вводе команды SQL. При выборе нескольких опций разделяйте их запятыми. Многоточие (...) означает, что вы можете повторить последнюю конструкцию нужное вам количество раз.

Ниже приведен (не-SQL) пример того, как эти элементы могут использоваться в записи команд.

```
BUY thing_name = price AS {cash | heck | credit}
  [, thing_name = price AS {cash | check | credit}]...
```

В этом выдуманном операторе BUY и AS являются ключевыми словами. Опции, заключенные в фигурные скобки и разделенные вертикальной чертой, означают, что вы должны выбрать один (и только один) способ оплаты. Также вы можете приобрести и ряд других вещей. Для каждой покупаемой вещи вы должны указать ее название, стоимость и способ оплаты.

{early_lunch no_lunch}	Вы должны выбрать один элемент
{soup , salad , sandwich}	Вы должны выбрать один или несколько элементов
[dessert]	Нет выбора
[coffee soda wine]	Вы можете выбрать один элемент или не выбирать вовсе
[tomato , pickle , onion]	Вы можете выбрать любое количество элементов

Рис. 3.3. Значения фигурных и квадратных скобок, вертикальных линий и запяток в операторах

Вот как будет выглядеть реальный пример:

```
buy lunch = 4.95 as cash, book = 34.95 as check
```

Если в записи оператора встречаются круглые скобки (`()`), они действительно являются его частью (в отличие от фигурных и квадратных скобок). Не забывайте об этом!

Операторы SQL обычно требуют наличия **терминаторов (terminator)**, которые иницируют исполнение оператора в системе управления базами данных. Различные SQL используют разные терминаторы, наиболее распространенными являются точка с запятой (`;`) и слово `go`. В системах баз данных с графическим интерфейсом (GUI) для выполнения команды можно просто щелкнуть на соответствующей кнопке или выбрать опцию из меню. Так как здесь могут быть самые разные варианты и возможности, терминаторы не включаются в примеры команд из этой книги.

Обработка ошибок

Ошибки в операторах SQL могут возникать по целому ряду причин. Наиболее распространенными являются ошибки при наборе, синтаксические ошибки, неправильное использование кавычек и имен объектов баз данных. Подобные ошибки представлены на рис. 3.4.

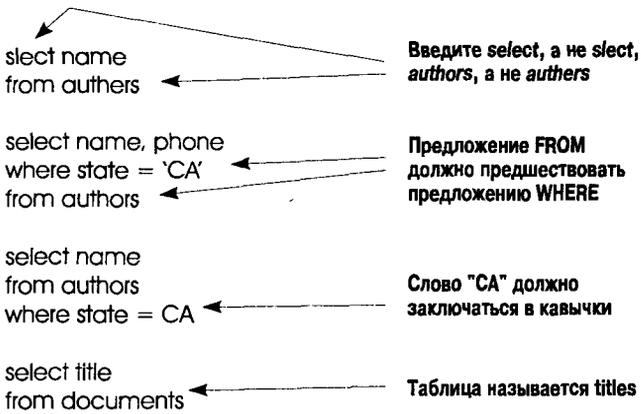


Рис. 3.4. Типичные ошибки в операторах SQL

Когда вы вводите оператор, который не поддерживается данной версией SQL, команда, естественно, не выполняется. Вместо этого на экране отображается сообщение об ошибке, полезность которого в разных системах может существенно отличаться. В хорошем сообщении будет представлена максимально возможная информация об ошибке, включая ее тип и номер строки с ошибкой.

Вот несколько примеров из SQL Anywhere:

SQL:

```
select *  
from publihsers
```

Результат:

```
table publihsers not found
```

SQL:

```
slect *  
from publishers
```

Результат:

```
Syntax error near ' * '
```

Получив такое сообщение, нужно исправить ошибку и снова запустить оператор на выполнение. (Сообщения об ошибках в разных версиях SQL могут кардинально отличаться друг от друга.)

СОЗДАНИЕ БАЗ ДАННЫХ

База данных — это именованная область на носителе данных, содержащая таблицы, курсоры, индексы и другие объекты. Сначала вы создаете саму базу данных, а затем входящие в нее объекты.

Так как под каждую новую базу данных выделяется достаточно большое пространство (даже если в ней никогда не будет содержаться много информации), права на создание баз данных в многопользовательских системах предоставляются далеко не всем. Если это касается и вас, вы можете не изучать команды SQL, предназначенные для создания баз данных, и сразу перейти к главам, в которых описываются правила создания объектов баз данных. В этом случае вы можете попросить системного администратора создать для вас базу данных, объяснив, что она не будет занимать много места на диске и предназначена исключительно для учебных целей.

В однопользовательских системах вся ответственность за управление ресурсами ложится на вас.

В большинстве многопользовательских версий SQL каждая база данных управляется или находится во владении конкретного пользователя, на которого возложена определенная ответственность и который наделен определенными правами. Эти права и степень ответственности различаются в разных системах, но обычно включают следующее.

- Установка прав других пользователей на использование базы данных.
- Регулярное создание резервных копий и запуск процедур восстановления в случаях сбоев системы.
- Выделение в случае необходимости дополнительного пространства на диске под базу данных.
- Владение большинством производных объектов базы данных.
- Понимание типов данных базы данных и умение их использовать.

Обычно база данных управляется создавшим ее пользователем. Но так как обычно монополией на создание баз данных владеет системный администратор, некоторые системы управления реляционными базами данных позволяют администратору передавать свои права другим пользователям.

В зависимости от используемого программного и аппаратного обеспечения базы данных могут физически храниться на дисках, в разделах диска или в файлах операционной системы. Для всех этих случаев мы будем использовать общий термин — **устройство базы данных (database device)**.

В большинстве систем все заботы о физическом хранении данных возлагаются на системного администратора. Поэтому перед тем как создавать базы данных с помощью операторов SQL, пользователи получают от него соответствующие указания.

В ANSI-стандарт SQL 1992 года (часто называемый SQL-92) не входит оператор CREATE DATABASE. Вместо этого он предусматривает команду CREATE SCHEMA для определения части базы данных, которой будет владеть конкретный пользователь. Обычно база данных состоит больше чем из одной **схемы (schema)**.

Однако в коммерческие версии SQL обычно включена команда CREATE DATABASE. Ее упрощенная форма (т.е. без необязательных предложений и расширений) имеет вид

```
CREATE DATABASE имя_базы_данных
```

Чтобы создать базу данных *bookbiz*, введите следующую команду:

```
SQL:  
create database bookbiz
```

(SQL Anywhere для создания базы данных использует не команду CREATE DATABASE, а специальную утилиту инициализации.)

В зависимости от реализации в запись этого оператора могут входить разные предложения, позволяющие вам (или системному администратору) управлять расположением базы данных (устройством базы данных, на котором физически расположена база данных), ее размером (пространством, которое может использоваться под базу данных) и другими параметрами. Например, общий синтаксис команды CREATE DATABASE в Sybase System 11 SQL Server имеет вид

```
CREATE DATABASE имя_базы_данных
[ON {DEFAULT | имя_устройства} [= SIZE]
 [ , имя_устройства [= SIZE]] ]
[LOG ON устройство_базы_данных [= SIZE]
 [ , устройство_базы_данных [= SIZE]] ]
[WITH OVERRIDE]
[FOR LOAD]
```

Чтобы получить представление о синтаксисе этого оператора в разных коммерческих системах, просмотрите Приложение Б. В нем представлен полный синтаксис большинства команд, описываемых в этой книге.

Выбор базы данных

Создание базы данных не означает автоматическую возможность ее использования. В зависимости от используемой вами версии SQL, сразу после создания базы данных, возможно, вам придется ввести приблизительно следующее:

```
USE имя_базы_данных
```

или

```
DATABASE имя_базы_данных
```

Возможно также, что для выбора нужной базы данных вам потребуется использовать команду CONNECT или возможности графического интерфейса пользователя. Уточните это в документации к своей системе.

СОЗДАНИЕ ТАБЛИЦ

Создав базу данных и подключившись к ней, можно начать создание таблиц.

Таблицы являются основными строительными блоками базы данных. В них содержатся строки и столбцы ваших данных. С помощью команд определения данных SQL можно создавать, удалять и манипулировать таблицами (добавлять, удалять, переставлять столбцы и менять их параметры).

В большинстве реализаций SQL таблицей владеет создавший ее пользователь, выдавая разрешения на ее использование другим пользователям. Чтобы создать таблицу, по меньшей мере, надо сделать следующее.

- Задать имя таблицы.
- Задать имена составляющих ее столбцов.
- Определить тип данных для каждого столбца.
- Определить (или использовать заданный по умолчанию) нулевой статус для каждого столбца — допускается или запрещается использование в столбце нулевых значений.

Для максимальной гибкости в базе данных *bookbiz* при определении таблиц используются только эти основные элементы. Тем не менее большинство производителей предусматривают дополнительные возможности для оператора CREATE TABLE, поддерживаемые стандартом SQL-92.

- Значения по умолчанию (автоматически вводимые значения, например текущая дата в соответствующем столбце заказа).
- Ограничения на возможные значения в столбцах (например, в столбце *pub_id* можно вводить только значения 0736, 0877 или 1389).
- Ограничения на определение первичных ключей.
- Проверка соответствия между первичными и внешними ключами.

Мы начнем с упрощенного синтаксиса оператора CREATE TABLE (имя столбца, тип данных и статус нуля), а ограничения рассмотрим далее в этой главе.

Команда SQL для определения таблицы записывается в виде CREATE TABLE. (Перед использованием оператора CREATE TABLE убедитесь, что вы используете подходящую базу данных.)

Упрощенный синтаксис оператора CREATE TABLE имеет следующий вид:

```
CREATE TABLE имя_таблицы
(имя_столбца тип_данных [NULL | NOT NULL]
[ , имя_столбца тип_данных [NULL | NOT NULL]] )
```

В качестве примера рассмотрим оператор CREATE TABLE, с помощью которого создается таблица *authors*:

SQL:

```
create table authors
(au_id char(11) not null,
au_lname varchar(40) not null,
au_fname varchar(20) not null,
phone char(12) null,
address varchar(40) null,
city varchar(20) null,
state char(2) null,
zip char(5) null )
```

Давайте изучим синтаксис этого оператора, так как до сих пор в этой книге вы не сталкивались со столь сложными конструкциями.

- Первое предложение — CREATE TABLE — не вызывает особых затруднений, если, конечно, выбранное имя таблицы соответствует правилам записи идентификаторов в вашей системе.
- Вторая строка начинается с открывающейся круглой скобки, ввод которой обязателен.
- Затем задайте имя первого столбца и через пробел укажите его тип данных. (В некоторых базах данных вы должны определить длину, размер или точность данных — обычно в виде целого числа в скобках, следующих сразу за описанием типа данных.)
- Ключевые слова NULL и NOT NULL, заключенные в квадратные скобки, необязательны. Вертикальная черта между ними означает, что можно использовать только одну из этих опций, а не две сразу. Если вы ничего не выберете, система сделает это за вас. По умолчанию SQL-92 разрешает нулевые значения, но по этому пути следуют не все коммерческие версии. Возьмите за привычку всегда явно указывать одну из этих опций, даже если вас устраивают значения, выбираемые по умолчанию. В будущем это может существенно упростить вашу работу.
- Третья строка, заключенная в квадратные скобки и с многоточием в конце, повторяет определение столбца. Квадратные скобки указывают, что определять второй столбец необязательно, а многоточие означает, что таким образом можно определить любое необходимое число столбцов (в рамках ограничений системы — обратитесь к справочному руководству по своей СУБД).

Обратите внимание, что каждое определение столбца отделяется от предыдущего определением запятой, а также не забудьте о закрывающейся круглой скобке после списка столбцов.

Кроме системных ограничений на число столбцов, которое можно включить в одну таблицу, узнайте об ограничениях на длину строки (суммарное количество байтов, которое могут использовать все столбцы). Обычно это достаточно большая величина, и вам вообще не придется об этом беспокоиться. Однако если вы определили таблицу с большим количеством столбцов или слишком широкими столбцами, вам следует аккуратно вычислить длину каждой строки, просуммировав длины максимально возможных значений для каждого столбца или исходя из описания типов данных столбцов.

Выбор типа данных

Тип данных столбца определяет, какого рода данные будут содержаться в этом столбце (символы, числа, даты и т.д.). Благодаря этому система узнает о физическом представлении данных и методах их обработки. Например, символьный тип данных поддерживает буквы, цифры и специальные символы, в то время как целый тип допускает только цифры. Над данными в столбцах целого типа можно выполнять арифметические операции, в отличие от столбцов символьного типа.

Реляционные системы поддерживают самые разнообразные типы данных. Однако надо быть исключительно внимательным — даже если две системы используют одно и то же название типа данных, значения типов могут отличаться. Обратитесь к справочному руководству по своей системе, чтобы выяснить имеющиеся в ней типы данных и правила работы с ними.

Выбор правильных типов данных является не менее сложным, чем проектирование структуры базы данных, так как во многих версиях SQL сложно корректно изменить тип данных столбца. Тем не менее в большинстве SQL реализованы специальные функции преобразования типов. Например, даже если в символьном столбце содержатся только числа, изменить его тип будет сложно или вообще невозможно. Однако с помощью специальных функций над этими числами можно будет выполнять арифметические операции, недопустимые для символьных столбцов.

Ниже описаны некоторые возможные типы данных и приведены советы по их использованию.

- **Символьные типы данных (*character datatypes*)** содержат буквы, цифры и специальные символы. Два основных типами являются символы с фиксированной длиной (*character* или *char*) и символы с переменной длиной (*variable character* или *varchar*). В некоторых наборах символов также поддерживаются национальные символы фиксированной и переменной длины (*national character*, *nchar*, *national character varying*, *nvarchar*). Ряд систем допускают использование специальных символьных типов данных для записи больших текстовых фрагментов. Обычно они называются типами *long* или *text*. Большинство символьных типов данных используется с параметром, определяющим максимальный размер столбца (число в скобках после описания типа данных). Символьные типы удобно использовать и в ряде неочевидных случаев — например при записи почтовых кодов и номеров телефонов. Почтовые коды лучше хранить в символьных столбцах, так как довольно часто возникает потребность в их сортировке, а коды, начинающиеся с нуля и представленные в числовом формате, могут сортироваться некорректно. Правда, некоторые системы просто отбрасывают все начальные нули во всех числах. (Порядок сортировки также различается от системы к системе.) Кроме того, в символьных столбцах можно использовать специальные символы, например дефисы и скобки в номерах телефонов.
- **Целые типы данных (*whole-number datatypes*)** поддерживают только целые числа (никаких дробных частей или десятичных точек). Они обычно известны под именами *number*, *integer*, *int*, *smallint* и *tinyint*. Во всех коммерческих версиях SQL над целыми типами разрешается выполнять арифметические

операции и применять к ним агрегирующие функции для поиска максимальных, минимальных, средних и суммарных значений столбца, а также подсчета количества значений в столбце. В некоторых реализациях SQL поддерживаются и другие возможности, например статистические операции.

- **Десятичные типы данных (decimal datatypes)** описывают числа с дробной частью. Для десятичных чисел с плавающей точкой используются имена *decimal* или *numeric*. Обычно разрешается определять их точность (общее количество цифр и количество цифр после запятой). Вещественные числа в стандартном виде описываются ключевыми словами *real*, *double*, *double precision*, *float* и *smallfloat*. Их точность и количество значащих цифр меняется от одной системы управления базами данных к другой, а также может зависеть от используемого вами аппаратного обеспечения.
- **Денежные типы данных (money datatypes)** используются для описания денежных величин. Если в вашей системе такой тип отсутствует, используйте для этих целей десятичные числа с плавающей точкой.
- **Дата и время (data and time datatypes)** используются для записи дат, времени и их комбинаций. В некоторых системах имеются функции, позволяющие определять интервал между двумя датами или добавлять, или вычитать определенное количество времени к конкретной дате.
- **Двоичные типы данных (binary datatypes)** используются для хранения двоичных и шестнадцатеричных кодов. Аналогично символьным данным, они могут иметь фиксированную или переменную длину.
- **Последовательные типы данных (serial datatypes)** используются для представления последовательно возрастающих числовых последовательностей. В одних системах — это отдельный тип, в других такие последовательности строятся с помощью специальных функций, оперирующих с данными основных типов.

Познакомившись с характеристиками типов данных в своей системе, вы сможете эффективнее использовать их.

Выбор длины типа данных. Прежде всего уточните, как физически хранятся в вашей системе данные разных типов. Узнайте, какие из них имеют фиксированную длину, другими словами, для каких типов данных дисковое пространство, выделяемое в операторе `CREATE TABLE`, не зависит от введенного в столбец значения. Например, под все значения столбца, определенного как *char(10)*, всегда будет отводиться по десять байт дискового пространства, независимо от того, какое значение будет введено. Более длинные значения будут подрезаться, а к более коротким справа будут добавляться пробелы:

```
Greenjeans  
Kangaroo__  
Ho_____  
Rumpelstil
```

Типы данных, не предусматривающие спецификацию длины, обычно имеют фиксированную длину. Если в вашей системе под целое значение отводится восемь байт, то все они будут использоваться, независимо от того, какое значение вводится — 3 или 300000.

Данные переменной длины занимают в памяти только минимально необходимое им место. Пусть, например, ваша система поддерживает символьный тип переменной длины, и вы определили столбец как *varchar(10)* вместо *char(10)*. При этом значение 10 будет просто определять максимально допустимую длину данных этого типа, но к более коротким значениям не будут дописываться пробелы. Для первого значения из примера выше (Greenjeans) потребуются все десять байт, для второго (Kangaroo) — восемь, а для третьего (Ho) — только два.

Если многие значения имеют заведомо меньшую длину, чем используемая при определении типа, применение типов данных с фиксированной длиной может привести к излишнему расходу дискового пространства.

Выбор точности. Десятичные типы с плавающей точкой позволяют устанавливать требуемую точность. Например, если известно, что десятичное значение состоит строго из шести цифр, причем три из них располагаются после запятой, его можно описать как *decimal(6,3)*.

Назначение нулевого статуса

В большинстве систем при создании таблицы можно указать статус каждого столбца (нулевой или ненулевой). Некоторые системы вообще не поддерживают нулевые значения — а просто по умолчанию вставляют некоторое значение (пробел или обычный ноль), если в столбец не были введены никакие данные.

При назначении нулевого статуса столбцу, пока пользователь не введет конкретные данные, все его значения будут нулевыми. Еще раз напоминаем, что под нулем подразумевается неизвестное, недоступное или неприменимое значение, а не обычный ноль или пробел.

Назначение ненулевого статуса столбцу означает, что все его значения должны быть определены. Если вы не задали какое-то значение столбца, система выдаст соответствующее сообщение об ошибке. (В системах, использующих значения по умолчанию, эти значения будут вводиться автоматически во все пустые места столбцов с ненулевым статусом.) Во многих системах по умолчанию используется NULL. Однако не нужно особенно полагаться на значения по умолчанию, лучше четко сформулировать то, что вы хотите получить.

Как же поступать с нулевым статусом столбцов при проектировании таблицы? Нельзя допускать появление нулей в столбцах, значения которых являются критическими для всей базы данных. Например, нули не могут появляться в столбцах первичных ключей, так как в этом случае они не будут однозначно идентифицировать строки. Так, без столбца *au_id* нельзя было бы определить авторов книг из базы данных *bookbiz*. Эта база данных построена так, что таблицы *authors* и *titles* связываются через объединение по столбцам *au_id* и *title_id*.

Кроме того, целесообразно запретить нулевые значения в столбцах *au_lname* и *au_fname*, так как достаточно странно вводить адрес и телефон автора, не зная его имени и фамилии.

Однако если известно, что определенные данные в столбце могут быть неизвестны, тогда можно разрешить появление в нем нулевых значений. Например, могут быть неизвестны адреса и номера телефонов. В столбце имен также могут быть нулевые значения, так как некоторые авторы используют однословные псевдонимы (например, *Colette*).

В таблице *titles* появление нулей не допускается только в столбцах *title_id* и *title*, так как первый является первичным ключом, а каждая книга идентифицируется уникальным номером. Однако в столбце *advance* могут находиться нулевые значения. Это может означать, что автор и издатель все еще не договорились о гонораре или его просто забыл ввести какой-то клерк.

Тогда как в случае с гонораром ноль обозначает “пока еще неизвестное значение”, в других ситуациях он может иметь совсем другие значения. Ноль может обозначать значение, которое вообще никогда не станет известно. Например, если в таблицу *authors* добавляется поле *phone*, а телефоны некоторых авторов неизвестны, они, возможно, навсегда заменяются нулями. Кроме того, ноль может означать, что некоторое значение в принципе неприменимо, например, если у автора вообще нет телефона.

В отличие от первичных ключей, внешние ключи иногда могут иметь нулевые значения. В случае с книгами и издателями в столбце *titles.pub_id* допускаются нулевые значения. Это, конечно, связано с бизнес-моделью, используемой в базе данных *bookbiz*: книги иногда могут приобретаться головной фирмой и только через некоторое время попадать в один из ее филиалов.

Отметим, что решение о том, допускаются ли в принципе нулевые значения во внешних ключах, принимается не на основе теоретического рассмотрения, а на основе логического анализа взаимосвязей между информацией первичных и внешних ключей. Все зависит от реальной ситуации — политики компании, организационных правил и т.д.

Процесс создания таблицы

Создание таблицы, подобно созданию базы данных, представляет собой пошаговый процесс, начинающийся с проектирования и заканчивающийся вводом команд SQL. Ниже описываются основные действия, которые надо предпринять для создания таблицы. При этом предполагается, что база данных уже спроектирована и нормализована, и, следовательно, известно, какие столбцы должны быть в каких таблицах.

- Определить типы данных (длину, точность, если необходимо) для каждого столбца.
- Определить, какие столбцы допускают нулевые значения, а какие — нет.
- Решить, какие столбцы должны содержать только уникальные значения. (В системах, не использующих ограничения, уникальность может обеспечиваться за счет индексирования, обсуждаемого дальше в этой главе и в главе 10. Предложения PRIMARY KEY и UNIQUE также описываются дальше в этой главе.)
- Разобраться с парами первичный ключ—внешний ключ. Простой синтаксис оператора CREATE TABLE, используемый в базе данных *bookbiz*, не позволяет выполнять над этими парами какие-либо действия. Можно только убедиться в совместимости их типов данных, размеров и нулевых статусов. Используя описываемый далее более сложный синтаксис, можно определять ограничения для обеспечения ссылочной целостности.
- В случае многопользовательского окружения убедиться в наличии прав на создание таблиц и индексов. При их отсутствии нужно поговорить об этом с системным администратором или владельцем используемой вами базы данных.
- Создать таблицу (и в случае необходимости все нужные индексы) с помощью операторов CREATE TABLE и CREATE INDEX.

На рис. 3.5 представлена структура таблицы *titles*: имена столбцов, типы данных, нулевые статусы, столбцы с уникальными значениями, первичные и вторичные ключи.

Столбец	Тип данных	Ноль?	Уникальный?	Ключи
title_id	varchar(11)	not null	да—titleidind	первичный
title	varchar(80)	not null	да—tilteind	
type	char(12)	null		
pub_id	char(4)	null		внешний (таблица <i>publishers</i>)
price	money	null		
advance	money	null		
ytd_sale	int	null		
contract	bit	not null		
notes	varchar(200)	null		
pubdate	datetime	null		

Рис. 3.5. Структура таблицы *titles*

Определение таблиц в базе данных *bookbiz*. Теперь вы готовы к созданию реальных таблиц. Ниже приведена команда CREATE TABLE для таблицы *titles*.

SQL:

```
create table titles
(title_id char(6) not null,
title varchar(80) not null,
type char(12) null,
pub_id char(4) null,
price money null,
advance money null,
ytd_sales int null,
contract bit not null,
notes varchar(200) null,
pubdate datetime null)
```

Если битовый тип данных в вашей системе не поддерживается, то для описания столбца *contract* используйте символьный тип. В этом столбце допускаются и нулевые значения.

Операторы CREATE TABLE для восьми оставшихся таблиц базы данных *bookbiz* приведены в Приложении Г, а также содержатся на прилагаемом к книге компакт-диске.

СОЗДАНИЕ ИНДЕКСОВ

В системах управления реляционными базами данных индексация является механизмом для повышения производительности. Подобно тому, как предметный указатель помогает находить требуемые страницы в книге, индексирование ускоряет выборку информации из базы данных. Когда вы ищете определенную тему в книге, то не просматриваете для этого подряд все страницы. Аналогично, при поиске нужных данных индексы служат в качестве логических указателей на их физическое местоположение.

Однако между предметным указателем в книге и индексом в базе данных существует несколько важных отличий. Читатель книги сам решает, обращаться ему к предметному указателю или нет. Пользователь реляционной базы данных только решает, создавать ему индекс или нет, а способы его использования при выполнении запросов определяет исключительно система. После своего создания индексы работают и используются независимо от пользователей базы данных (за исключением ряда коммерческих расширений SQL, позволяющих опытным пользователям или администраторам баз данных управлять использованием индексов, что выходит за рамки этой книги).

Тогда как каждая книга вместе с предметным указателем печатается единственный раз, данные в реляционных системах и их индексах могут постоянно меняться. В любой момент данные в таблице могут измениться, при этом должны соответствующим образом поменяться один или несколько индексов. При этом все заботы о поддержке индексов берет на себя система.

Еще одно отличие между предметными указателями книг и индексами баз данных состоит в том, что очень часто таблица имеет несколько индексов (правда, иногда нечто подобное встречается и в книгах — тематический указатель и указатель по авторам). Объединяет предметные указатели и индексы то, что и книга, и таблица могут их вообще не иметь.

Дальше в этой главе приводится синтаксис операторов CREATE INDEX и DROP INDEX, описываются способы индексирования в реляционных системах и даются рекомендации по выбору подходящих для индексирования столбцов.

Оператор CREATE INDEX

В большинстве систем имеется команда следующего вида:

```
CREATE [UNIQUE] INDEX имя_индекса
ON имя_таблицы (имя_столбца)
```

Имена таблицы и столбца определяют столбец, который вы собираетесь индексировать, и таблицу, которой он принадлежит.

Чтобы проиндексировать таблицу *authors* по столбцу *au_id*, нужно выполнить следующую команду:

```
SQL:
create index auidind
on authors(au_id)
```

Лучше всего выполнять индексирование при создании таблицы, однако SQL также позволяет создавать индексы и после заполнения таблиц данными.

Во многих системах поддерживаются **составные индексы (composite indexes)** — индексы, использующие несколько столбцов, и **уникальные индексы (unique indexes)** — предотвращающие дублирование данных. В некоторых системах реализованы **групповые индексы (clustered index)** с возможностью не только логической, но и физической сортировки. Уточните в справочном руководстве по своей системе, какими типами индексов вы располагаете.

Составные индексы. Составные индексы используются, когда одновременно лучше осуществлять поиск по двум или нескольким столбцам, что объясняется их логической взаимосвязью. Например, таблица *authors* имеет составной индекс на основе столбцов *au_fname* и *au_lname*.

При создании составного индекса нужно указать все соответствующие столбцы. Команда, создающая составной индекс по таблице *authors*, может иметь следующий вид:

```
SQL:
create index aunameind
on authors (au_lname, au_fname)
```

В большинстве диалектов SQL порядок столбцов при создании составного индекса не обязательно должен повторять их порядок в операторе CREATE TABLE. Например, столбцы *au_lname* и *au_fname* могут быть перечислены в операторе создания индекса в обратном порядке. Однако, с точки зрения производительности, лучше первым указывать столбец, по которому чаще всего выполняется поиск.

Уникальные индексы. Уникальность означает, что никакие две строки не могут иметь индексы с одинаковыми значениями. В этом случае при создании индекса и при каждом добавлении информации система будет следить за тем, чтобы значения индексов не повторялись. Уникальные индексы обычно создаются на ключевых столбцах, чтобы усилить их возможности как уникальных идентификаторов строк.

Создавать уникальные индексы имеет смысл только тогда, когда это диктуется самими данными. Например, не нужно создавать уникальный индекс по столбцу *last_name*, так как даже в таблице всего из нескольких сотен строк наверняка найдется несколько человек с фамилиями *Smith* или *Wong*. С другой стороны, целесообразно создать уникальный индекс по столбцу с номерами карточек социального страхования. В этом случае уникальность является характеристикой данных: эти номера обязательно различаются у разных людей. Кроме того, уникальные индексы служат в качестве гаранта целостности. Одинаковые номера карточек социального страхования свидетельствуют об ошибке ввода данных или просчете государственной службы.

Реализации SQL, поддерживающие уникальные индексы, должны иметь для этого специальные механизмы. Обычно система гарантирует уникальность, отбрасывая команды, которые пытаются выполнить следующее.

- Создать уникальный индекс на существующих данных, которые содержат одинаковые значения.
- Изменить данные, на основе которых построен уникальный индекс, таким образом, чтобы в них появились одинаковые значения.

Для создания составных и простых (одностолбцовых) индексов применяется ключевое слово **UNIQUE**.

Групповые индексы. Некоторые системы управления реляционными базами данных предоставляют пользователям выбор между групповыми и негрупповыми индексами. При создании группового индекса строки таблиц сортируются таким образом, что их физический порядок на устройстве базы данных совпадает с их логическим (индексным) порядком.

Поскольку групповой индекс управляет физическим положением данных, таблица может иметь только один такой индекс. Групповые индексы обычно создаются на первичных ключах, но целесообразнее использовать для их построения столбцы с наиболее часто запрашиваемой информацией.

При использовании негруппового индекса физический порядок строк не совпадает с их индексным порядком. Все негрупповые индексы таблицы могут обеспечивать доступ к данным в разном порядке.

Поиск данных с использованием групповых индексов почти всегда выполняется быстрее поиска с негрупповыми индексами. Это преимущество особенно ощутимо, когда из базы данных извлекается целый набор строк с последовательными **ключевыми значениями (key values)**. Как только будет найдена строка с первым ключевым значением, дальнейший поиск прекращается, так как строки с последовательными индексными значениями физически располагаются друг за другом. Однако наличие группового индекса может замедлять выполнение операторов модификации данных, так как системе требуется время на перестройку индекса при изменении ключевых значений.

Как, что и зачем нужно индексировать

Индексирование ускоряет процесс выборки данных. После создания индекса по столбцу, запрос, который до этого требовал для своего выполнения достаточно много времени, может отработать почти мгновенно.

Так почему же не проиндексировать все столбцы? Одна из основных причин не делать этого состоит в том, что построение и поддержка индексов требует времени и места на устройстве базы данных.

Другая причина связана с тем, что вставка, удаление или изменение данных в индексированных столбцах требует несколько большего времени, чем в неиндексированных, так как при изменении ключевых значений системе необходимо дополнительное время на перестройку индекса. Однако это обычно компенсируется за счет увеличения производительности при выборке данных.

В общем случае имеет смысл индексировать только часто используемые в запросах столбцы, в первую очередь ключевые столбцы и столбцы, используемые для объединения и сортировки. Вот несколько более точных рекомендаций. Обычно должны индексироваться:

- столбцы с первичными ключами, особенно если они часто используются в операциях объединения с другими таблицами. Уникальные индексы на первичных ключах предотвращают появление в них одинаковых значений и гарантируют, что каждое значение в столбце первичного ключа будет однозначно идентифицировать строку;
- столбцы, по которым часто выполняется сортировка;
- столбцы, постоянно используемые в операциях объединения, так как в этом случае объединение будет выполняться быстрее;
- столбцы, по группам значений которых часто выполняется поиск, особенно если система поддерживает групповые индексы. Как только будет найдена строка с первым искомым значением, дальнейший поиск прекращается, так как строки с последовательными значениями физически располагаются друг за другом. При поиске единственного значения групповые индексы такими преимуществами не обладают.

Существует ряд случаев, в которых применение индексов нецелесообразно.

- Индексирование столбцов, которые редко используются в запросах, не приводит к повышению производительности, так как их значения очень редко или вообще не используются при поиске и выборке строк из базы данных. Однако и в этом случае индексирование можно применять для обеспечения уникальности значений столбца.
- Не имеет смысла индексировать столбцы только с двумя-тремя значениями (например, с описанием пола).
- Индексирование не приводит к осязаемому выигрышу на небольших таблицах с несколькими строками. В этом случае для поиска информации обычно используется простое **сканирование таблицы (table scan)** — поочередный просмотр строк, а не индекс. Время такого сканирования пропорционально количеству строк в таблице.

Индексирование является довольно сложной темой. Чтобы правильно его использовать, необходимо понимать, как работает системный оптимизатор запросов и какие требования с точки зрения производительности выдвигаются к вашему приложению. Вообще говоря, нужно начинать с построения очевидно необходимых индексов, а затем следить за получаемой производительностью на этапах разработки прототипа и тестирования. Когда вы поймете, что вам необходимо, сможете должным образом настроить и индексы.

СОЗДАНИЕ ТАБЛИЦ С ПОМОЩЬЮ ОГРАНИЧЕНИЙ SQL-92

Большинство коммерческих систем поддерживают предложения (ограничения) PRIMARY KEY, UNIQUE, DEFAULT, CHECK, REFERENCES и FOREIGN KEY в команде CREATE TABLE в соответствии со стандартом SQL-92. Эти элементы обеспечивают защиту целостности данных.

- PRIMARY KEY помечает столбец (в котором не могут присутствовать нулевые значения) в качестве первичного ключа таблицы. Каждое вводимое в этот столбец значение должно быть уникальным, ввод одинаковых значений не допускается. Внутренняя реализация этого ограничения различается от системы к системе, но часто она эквивалентна индексу. Если первичный ключ включает несколько столбцов, PRIMARY KEY определяется на уровне таблицы.
- UNIQUE также гарантирует отличие между всеми значениями в столбце, но допускает в нем одно нулевое значение.
- DEFAULT определяет значение, которое автоматически вставляется системой, если пользователь не введет требуемые данные. Например, можно определить значение по умолчанию для столбца *type* в таблице *titles*. Если книга еще не классифицирована, то система могла бы автоматически вставлять в этот столбец любое назначенное вами значение, например слово *unclassified* (не классифицирована). Если значение по умолчанию не определено, но столбец *type* допускает нулевые значения, система автоматически вставит вместо неизвестных данных значение NULL.
- CHECK определяет, какие данные могут вводиться в определенный столбец, т.е. позволяет определить область значений столбца. Например, вы можете захотеть, чтобы в столбец *title_id* всегда вводилось значение, состоящее из двух букв и четырех цифр, а в столбец *type* — один из шести допустимых терминов. Если пользователь попытается нарушить это ограничение, соответствующая команда модификации данных выполняться не будет. Эти ограничения иногда называются **правилами (rules)**, поскольку позволяют проверить принадлежность вводимых данных области значений данного столбца. Правила, использующие несколько столбцов, определяются на уровне таблицы.

- REFERENCES и FOREIGN KEY связывают вместе первичные и внешние ключи. Если вы вводите значение в столбец внешнего ключа, определенного с помощью предложения REFERENCES, этот столбец и столбец, на который он ссылается, должны существовать в таблице, иначе команда модификации данных будет отвергнута.

Перед тем как определять ограничения и значения по умолчанию, четко сформулируйте свои требования. На рис. 3.6 представлены значения по умолчанию и ограничения, используемые в таблице *titles*.

Столбец	Тип данных	Нуль?	Ключ	Значение по умолчанию	Ограничения	Ссылки
title_id	char(6)	not null	первичный, уникальный		2 буквы и 4 цифры	
title	varchar(80)	not null	уникальный			
type	char(12)			не классифицирована	business, mod_cook, trad_cook, psychology, popular_comp, unclassified	
pub_id	char(4)	null				publishers, pub_id
price	money	null				
advance	money	null				
ytd_sale	int	null				
contract	bit	not null				
notes	varchar(200)	null				
pubdate	datetime	null		текущая дата		

Рис. 3.6. Использование ограничений при создании таблицы *titles*

Синтаксис этих предложений может быть различным в разных системах, так что уточните его из руководства к своей системе. Обычно они следуют после определения столбца.

Кроме того, эти предложения можно узнать по ключевым словам DEFAULT, CHECK, PRIMARY KEY, UNIQUE, FOREIGN KEY и REFERENCES. Помимо этого, может присутствовать слово CONSTRAINT и имя ограничения. (Задание имен ограничениям упрощает процесс их поиска и изменения. Хотя в большинстве систем для ограничений генерируются специальные метки, использовать их обычно неудобно.) Ограничения на таблицу лучше всего помещать после списка всех ее элементов (хотя это и необязательно). Столбцы могут иметь более одного ограничения, причем ограничения не разделяются запятыми:

```
CREATE TABLE имя_таблицы
(имя_столбца тип [NULL | NOT NULL] [DEFAULT значение_по_умолчанию]
 [ограничение_на_столбец]...
 [ , имя_столбца тип[NULL | NOT NULL] [DEFAULT значение_по_умолчанию]
 [ограничение_на_столбец]... ]...
 [ограничение_на_таблицу]...)
```

Ниже представлена SQL-версия таблицы *titles* (см. рис. 3.6) для SQL Anywhere (под названием *titlescnstr*), использующая значения по умолчанию и ограничения.

(Предложения LIKE и IN рассматриваются в следующей главе. Для столбца *pubdate* SQL Anywhere использует значение по умолчанию CURRENT DATE, соответствующее текущей дате. Аналогичные функции имеются и в других системах.)

```
create table titlescnstr
(title_id char(6) not null
 primary key
 check (title_id like '[A-Z] [A-Z] [0-9] [0-9] [0-9] [0-9] ' ),
title varchar(80) not null
 unique,
type char(12)
 default 'unclassified' null
 check (type in ( 'business', 'mod_cook', 'trad_cook',
 'psychology', 'popular_comp', 'unclassified' )),
pub_id char(4) null
 references publishers (pub_id),
price money null,
advance money null,
ytd_sales int null,
contract bit not null,
notes varchar(200) null,
pubdate datetime null
 default current date)
```

Код для этой же таблицы в Sybase SQL Server имеет ряд отличий:

SQL:

```
create table titlescnstr
(title_id char(6) not null
 constraint tididx primary key
 constraint tidcheck check
(title_id like '[A-Z] [A-Z] [0-9] [0-9] [0-9] [0-9]' ),
title varchar(80) not null
 constraint titleidx unique,
type char(12)
 default 'unclassified' null
 constraint typecheck check
 (type in ( 'business', 'mod_cook', 'trad_cook',
 'psychology', 'popular_comp', 'unclassified' )),
pub_id char(4) null
 references publishers (pub_id),
price money null,
advance money null,
ytd_sales int null,
contract bit not null,
notes varchar(200) null,
pubdate datetime
 default getdate ( ) null )
```

Уточните соответствующие детали в своем руководстве по системе. Ограничения на таблицу используют более одного столбца. На рис. 3.7 показаны ограничения для таблицы *titleauthors*.

Ниже показано, как в SQL Anywhere реализуются ссылки на примере таблицы *titleauthors* (во избежание недоразумений она называется *titleauthorscnstr*). Ограничение на первичный ключ является ограничением на всю таблицу, так как использует более одного столбца.

Столбец	Тип данных	Ноль?	Ключ	Значение по умолчанию	Ограничения	Ссылки
au_id	char(11)	not null	первичный ключ			authors.au_id
title_id	char(6)	not null	первичный ключ			titles.title_id
au_ord	tinyint	null				
royaltyshare	float	null				

Рис. 3.7. Использование ограничений при создании таблицы titleauthors

SQL:

```
create table titleauthorscnstr
(au_id char(11) not null references authors (au_id),
title_id char(6) not null references titles (title_id),
au_ord tinyint null,
royaltyshare float null,
primary key (au_id, title_id))
```

Ограничения с проверкой также могут применяться ко всей таблице, если они используют более одного столбца (столбцы должны принадлежать одной таблице). Например, в таблице *salesdetails* есть столбцы для описания количества заказанных и отправленных книг. В этом случае можно использовать табличное ограничение, проверяющее, что количество отправленных не превышает количество заказанных книг. Все ограничения для таблицы *salesdetails* представлены на рис. 3.8.

SQL:

```
create table salesdetailscnstr
(sonum int not null references sales (sonum) ,
qty_ordered smallint not null,
qty_shipped smallint null,
title_id char(6) not null references titles
(title_id),
date_shipped datetime null,
check (qty_shipped <= qty_ordered),
primary key (sonum, title_id))
```

Столбец	Тип данных	Ноль?	Ключ	Значение по умолчанию	Ограничения	Ссылки
sonum	int	not null	первичный ключ			sales.sonum
title_id	char(6)	not null	первичный ключ			titles.title_id
qty_shipped	smallint	null			qty_shipped not > than qty_ordered	
qty_ordered	smallint	not null			qty_shipped not > than qty_ordered	
date_shipped	datetime	null				

Рис. 3.8. Использование ограничений при создании таблицы salesdetails

Чтобы предложение REFERENCES работало надлежащим образом, оператор SQL Anywhere CREATE TABLE требует, чтобы столбец *sonum* в таблице *sales* являлся либо первичным ключом, либо имел уникальный индекс. Не забудьте сделать это перед выполнением описанной выше команды.

В ряде систем имеются специальные команды (не связанные с оператором CREATE TABLE) для обработки значений по умолчанию, правил и ограничений на целостность. Они обсуждаются в главе 10. Первые СУБД иногда перекладывали вопросы, связанные со значениями по умолчанию и ограничениями на целостность, на приложения и не поддерживали соответствующие команды SQL.

ИЗМЕНЕНИЕ И УДАЛЕНИЕ БАЗ ДАННЫХ И ИХ ОБЪЕКТОВ

Итак, вы поняли, что базы данных и их объекты создаются с помощью той или иной модификации команды CREATE. Для удаления самой базы данных и объектов базы данных в большинстве систем предусмотрена команда DROP. (Как ни странно, до 1988 г. эта команда не входила в стандарт ANSI.) Для изменения объектов обычно применяется команда ALTER.

Изменение баз данных

В некоторые версии SQL включена команда, позволяющая изменять размер базы данных (обычно в большую сторону). Возможность выделения дополнительного пространства на устройстве базы данных особенно важна для растущих со временем приложений. Не менее важна (но реже реализуема) возможность просмотра базы данных на предмет наличия неиспользуемого пространства на устройстве базы данных.

Изменение определений таблицы

Спроектировав и создав базу данных, через некоторое время вы можете обнаружить, что она недостаточно хороша, или что просто изменились некоторые требования. В некоторых системах с помощью команды ALTER TABLE можно изменять структуру таблицы (даже если она заполнена данными). В ней могут использоваться ключевые слова для добавления и удаления столбцов, изменения их имени, типа, длины, нулевого статуса и ограничений.

Например, чтобы добавить столбец к таблице *authors*, можно попробовать ввести следующую команду:

SQL:

```
alter table authors  
add birth_date datetime null
```

Столбцы, добавленные к таблице с помощью оператора ALTER TABLE, как правило, должны допускать нулевые значения. Это связано с тем, что если новый столбец добавляется к уже заполненной данными таблице, то в нем должны содержаться некоторые значения, и самым естественным выбором при этом являются “нулевые” (неизвестные) значения.

Во многих реляционных системах добавление столбца является единственной командой, позволяющей выполнять структурные изменения в таблицах. В них отсутствуют команды для удаления и переименования столбцов, изменения их типа и нулевого статуса. Однако обычно можно избежать этих ограничений. Например, если не удастся физически удалить столбец, можно создать не включающую его виртуальную таблицу и с ее помощью выполнять все операции по выборке и модификации данных. Кроме того, с помощью виртуальной таблицы можно создать иллюзию изменений

- в названии столбца, просто определив новый заголовок в предложении SELECT оператора CREATE VIEW;
- в типе данных столбца, используя различные функции преобразования типов данных.

Для реструктуризации таблицы имеется и другая возможность. Можно создать новую таблицу с требуемой вам структурой, сохранить данные из старой таблицы в файле операционной системы, а затем загрузить их в новую таблицу. В некоторых системах имеется специальная команда (в Sybase SQL Server — SELECT INTO), позволяющая перемещать данные между таблицами разных структур. Подробности вы узнаете в главе 9.

Удаление базы данных

При удалении базы данных или объекта базы данных канут в лету все связанные с ними структуры и данные. Поэтому в большинстве систем выполнять команду DROP разрешается только либо владельцу соответствующего объекта, либо лицу, наделенному специальными полномочиями.

Синтаксис команды DROP DATABASE обычно имеет следующий вид:

```
DROP DATABASE имя_базы_данных
```

Это очень опасная команда, так как при ее выполнении уничтожается все содержимое базы данных.

Удаление таблиц

Для удаления таблицы из базы данных используется команда DROP TABLE. В большинстве диалектов SQL она имеет следующий вид:

```
DROP TABLE имя_таблицы
```

При выполнении этой команды из базы данных удаляется заданная таблица со всем ее содержимым.

Если же требуется сохранить структуру таблицы, но удалить из нее все данные, можно воспользоваться командой DELETE (читайте о ней дальше в этой главе).

Удаление индекса

Удаление индекса может потребоваться в двух ситуациях:

- вы, или кто-то другой создали индекс по столбцу, но в большинстве запросов он не используется;
- необходимо серьезным образом модифицировать ключевые значения. Поскольку при каждом таком изменении система будет перестраивать индекс, имеет смысл сначала удалить его, а после выполнения модификаций создать индекс заново.

Хотя повторное создание индекса потребует некоторого времени, это удобнее, чем наблюдать, как система “задумывается” при выполнении каждого оператора модификации данных.

В большинстве систем команда удаления индекса имеет следующий вид:

```
DROP INDEX имя_таблицы.имя_индекса
```

При выполнении этой команды система удаляет из базы данных заданный индекс. Команда для удаления индекса *aidind* в таблице *authors* имеет следующий вид:

SQL:

```
drop index authors.aidind
```

ДОБАВЛЕНИЕ, ИЗМЕНЕНИЕ И УДАЛЕНИЕ ДАННЫХ

После проектирования и создания базы данных (таблиц и, возможно, индексов) для начала полноценной работы в нее нужно поместить данные, которые впоследствии можно будет в случае необходимости добавлять, изменять и удалять. У вас есть структура. Теперь требуется наполнить ее содержанием.

В SQL для изменения данных используются три основные команды (их часто называют операторами модификации данных).

- Оператор INSERT добавляет новые строки в базу данных.
- Оператор UPDATE изменяет существующие в базе данных строки.
- Оператор DELETE удаляет строки из базы данных.

В этом разделе описываются команды SQL по модификации данных. В следующем разделе приводится пример использования оператора INSERT.

Другой метод добавления данных в таблицу подразумевает их загрузку с помощью специальной команды вставки из файла операционной системы. Этот метод особенно подходит для переноса данных из одной системы управления базами данных в другую.

Реляционные системы с графическим интерфейсом пользователя предоставляют для ввода данных специальные **формы (form)** — они напоминают обычный бумажный бланк, в строки которого вводятся необходимые данные. Формы обычно удобнее использовать, чем операторы модификации данных, так как они позволяют автоматизировать и упростить работу. Однако все действия по модификации в системах реляционных баз данных выполняются на основе команд SQL, так что изучать их надо даже независимо от того, планируете ли вы их использовать в будущем.

Выполнять операторы модификации данных обычно позволяет не всем. Владелец базы данных или владельцы отдельных объектов базы данных с помощью операторов GRANT и REVOKE могут разрешить отдельным пользователям выполнять определенные команды модификации.

С помощью каждого оператора модификации (INSERT, UPDATE, DELETE) за один раз можно изменять данные только в одной таблице. Однако в ряде систем эти изменения могут касаться данных из других таблиц и даже других баз данных. Используя оператор SELECT в команде модификации данных, можно переместить значения из одной таблицы в другую. Подобные команды описываются дальше в этой главе.

С некоторыми ограничениями команды модификации данных работают и на виртуальных таблицах. За подробностями обращайтесь к главе 9.

Добавление новой строки

Оператор INSERT позволяет добавлять строки в базу данных одним из двух способов: с помощью ключевого слова VALUES или с помощью оператора SELECT. Опишем сначала правила использования ключевого слова VALUES.

Ключевое слово VALUES определяет значения некоторых или всех данных в столбцах новой строки. Ниже представлена общая форма оператора INSERT, использующего ключевое слово VALUES:

```
INSERT INTO имя_таблицы [(столбец1 [ , столбец2]...)]  
VALUES (константа1 [ , константа2]...)
```

Следующий оператор INSERT добавляет новую строку в таблицу *publishers* и задает значения для каждого столбца:

```
SQL:  
  
insert into publishers  
values ('1622', 'Jardin, Inc.', '5 5th Ave.', 'Camden', 'NJ')
```

Обратите внимание, что значения нужно вводить в том порядке, в котором определялись столбцы в соответствующем операторе CREATE TABLE (другими словами, сначала идентификационный номер, затем имя, адрес, город и, наконец,

штат). Данные после ключевого слова VALUES заключаются в круглые скобки. В большинстве систем значения вводятся в двойных или одинарных кавычках и разделяются запятыми.

Для каждой добавляемой строки используется отдельный оператор INSERT.

Вставка данных в несколько столбцов. Если данные добавляются не во все столбцы таблицы, их нужно дополнительно определить. Во избежание сбоев, для столбцов, значения которых не вводятся, должны быть определены значения по умолчанию либо они должны допускать нулевые значения. Например, для добавления данных только в два столбца (скажем, в *pub_id* и *pub_name*) используется следующая команда:

SQL:

```
insert into publishers (pub_id, pub_name)
values ('1756', 'HealthText')
```

Порядок перечисления столбцов в операторе INSERT может быть любым, но при этом он должен соответствовать порядку перечисления значений. Например, тот же результат достигается при выполнении следующего оператора (в нем перечислены столбцы *pub_name* и *pub_id*):

SQL:

```
insert into publishers (pub_name, pub_id)
values ('HealthText', '1756')
```

Оба этих оператора помещают значение "1756" в столбец идентификационных номеров и "HealthText" в столбец имен издателей. А что будет со столбцами *address*, *city* и *state*? Добавленную в таблицу *publishers* строку можно просмотреть с помощью следующего оператора SELECT:

SQL:

```
select pub_id, pub_name, address, city, state
from publishers
where pub_name = 'HealthText'
```

Результат:

pub_id	pub_name	address	city	state
1756	HealthText	NULL	NULL	NULL

Так как в операторе INSERT не определены значения для столбцов *address*, *city* и *state*, а таблица *publishers* допускает использование для них нулевых значений, они будут обнулены. Если в вашей системе не разрешается использование нулевых значений, вместо пропущенных данных вы, скорее всего, увидите пробелы или обычные ноли.

Если в операторе CREATE TABLE для столбцов *city* и *state* определен ненулевой статус, предыдущий оператор INSERT не сработает, так как для таких столбцов должны вводиться все значения. Попытка выполнить в Transact-SQL оператор INSERT, в котором не определяется значение для столбца *pub_id* (получившего при создании таблицы *publishers* ненулевой статус), приведет к следующим результатам:

SQL:

```
insert into publishers (pub_name, address, city, state)
values ('Tweedledum Books', '1 23rd St.', 'New York', 'NY')
```

Результат:

```
column 'pub_id' in table 'publishers' cannot be NULL (столбец 'pub_id' в
таблице 'publishers' не может быть нулевым)
```

Использование оператора SELECT в команде INSERT

Для получения данных из одной или нескольких таблиц в команде INSERT можно использовать оператор SELECT. Упрощенный синтаксис команды INSERT, использующей оператор SELECT, имеет следующий вид:

```
INSERT INTO имя_таблицы [(вставляемый_список_столбцов)]
SELECT список_столбцов
FROM список_таблиц
WHERE условия
```

Оператор SELECT в команде INSERT позволяет взять данные из нескольких или всех столбцов одной таблицы и вставить их в другую таблицу. Если вы вставляете значения только для части столбцов, определить значения для других столбцов можно будет позднее с помощью оператора UPDATE.

Если вы вставляете строки из одной таблицы в другую, эти таблицы должны иметь совместимую структуру, т.е. соответствующие столбцы должны иметь одинаковый тип, или же система должна уметь автоматически выполнять нужное преобразование.

Если столбцы в обеих таблицах совместимы по типам и определены в одинаковом порядке в соответствующих операторах CREATE TABLE, перечислять их в команде INSERT необязательно. Предположим, что в таблице *newauthors* содержатся строки с информацией об авторах в том же формате, что и в таблице *authors*. Для добавления всех строк из таблицы *newauthors* в таблицу *authors* можно воспользоваться одной из следующих команд:

SQL:

```
insert into authors
select au_id, au_lname, au_fname, phone, address, city, state, zip
from newauthors
```

```
insert into authors
select *
from newauthors
```

Если столбцы в двух таблицах (таблица, в которую вы вставляете данные, и таблица, из которой вы берете данные) определены в разном порядке в соответствующих операторах CREATE TABLE, для установления соответствия между ними можно воспользоваться предложениями INSERT или SELECT.

Например, предположим, что в операторе CREATE TABLE для таблицы *authors* столбцы определены в следующем порядке — *au_id*, *au_fname*, *au_lname* и *address*, а для таблицы *newauthors* — *au_id*, *address*, *au_fname* и *au_lname*. Тогда установить соответствие между ними можно с помощью оператора INSERT. Для этого столбцы таблицы *authors* нужно перечислить в предложении INSERT:

SQL:

```
insert into authors (au_id, address, au_lname, au_fname)
select * from newauthors
```

Такой же результат можно получить, перечислив в нужном порядке столбцы таблицы *newauthors* в предложении SELECT:

SQL:

```
insert into authors
select au_id, au_fname, au_lname, address
from newauthors
```

Если последовательности столбцов в двух таблицах не согласованы, операция INSERT либо не будет выполняться вообще, либо выполнится не полностью. При этом данные могут быть размещены в неверных столбцах. Например, вы вряд ли захотите, чтобы информация об адресе попала в столбец *au_lname*.

Выражения. Одним из преимуществ использования оператора SELECT в команде INSERT является возможность включения в него различных **выражений (expression)** — строк символов, математических формул и функций, позволяющих манипулировать вставляемыми данными. (Подробная информация об этом содержится в главах 4–8.)

Ниже приводится пример предложения SELECT, в котором над столбцом выполняются математические действия. Предположим, что один из филиалов описываемой нами компании перекупил серию книг у другой издательской компании. Причем, по счастливой случайности, для описания книг эта компания использовала таблицу с той же структурой, что и таблица *titles*. Эти книги описаны в таблице *Books* и их нужно поместить в таблицу *titles*. Однако при покупке стоимость этих книг была увеличена на 50%. Оператор, увеличивающий значения стоимости книг и вставляющий строки из таблицы *Books* в таблицу *titles*, имеет следующий вид:

```
SQL:
insert into titles
select title_id, title, type, pub_id, price * 1.5,
       advance, royalty, ytd_sales, contract, notes, pubdate
from Books
```

Вставка данных в несколько столбцов. С помощью оператора SELECT можно добавлять данные как во все сразу, так и в отдельные столбцы, по аналогии с предложением VALUE. Для этого нужно просто задать имена столбцов, в которые вы хотите добавить данные в предложении INSERT.

Например, если в таблице *titles* имеются книги, на которые еще не заключены контракты, и которые, следовательно, не представлены соответствующими строками в таблице *titleauthors*, тогда для выборки идентификационных номеров из таблицы *titles* и вставки их в таблицу *titleauthors* (пока просто для резервирования места) можно воспользоваться следующим оператором:

```
SQL:
insert into titleauthors (title_id)
select title_id
       from titles
       where contract = 0
```

Однако в этот оператор закралась ошибка, так как необходимо ввести значение в столбец *au_id* таблицы *titleauthors* (согласно определению этой таблицы, в столбце *au_id* не допускаются нулевые значения и для него не определены значения по умолчанию). Поэтому в столбец *au_id* в виде константы надо поместить **фиктивное значение (dummy value)**, например *xxxxxx*:

```
SQL:
insert into titleauthors (title_id, au_id)
select title_id, 'xxxxxx'
from titles
where contract = 0
```

В результате выполнения этой команды таблица *titleauthors* будет содержать две новые строки с реальными значениями в столбце *title_id*, фиктивными значениями в столбце *au_id* и нулевыми значениями в двух других столбцах. Однако это не сработает, если по столбцу построен уникальный индекс или на него наложены ограничения UNIQUE или PRIMARY KEY.

В большинстве версий SQL не допускается использование одной и той же таблицы для выборки и вставки данных:

```
SQL:
insert into test
select *
from test
```

Sybase SQL Server — одна из немногих систем, допускающих подобный синтаксис.

ИЗМЕНЕНИЕ СУЩЕСТВУЮЩИХ ДАННЫХ

В то время как оператор INSERT добавляет в таблицу новые строки, оператор UPDATE предназначен для изменения существующих в таблице данных.

В операторе UPDATE нужно указать изменяемые строки и их новые значения. Новые данные могут быть константами или выражениями, или могут быть получены из других таблиц.

Ниже приведен упрощенный синтаксис команды UPDATE, изменяющей выбранные строки:

```
UPDATE имя_таблицы
SET имя_столбца = выражение
[WHERE условие]
```

Оператор UPDATE

Ключевое слово UPDATE предшествует названию таблицы или курсора (виртуальной таблицы). Как и в случае с другими операторами модификации данных, в каждом операторе UPDATE можно изменять данные только одной таблицы.

Если при выполнении оператора UPDATE нарушаются ограничения на целостность (например, добавляемое значение имеет неверный тип), система запрещает обновление данных и обычно выдает сообщение об ошибке. Ограничения на обновление виртуальных таблиц описываются в главе 8.

Предложение SET

В предложении SET определяются столбцы и указываются их новые значения. В предложении WHERE можно указать изменяемые строки. Если предложение WHERE не используется, то при выполнении команды UPDATE будут изменены значения во всех строках столбцов указанных в предложении SET.

Например, таблица *publishers* имеет следующий вид:

SQL:

```
select *
from publishers
```

Результат:

<u>pub_id</u>	<u>pub_name</u>	<u>address</u>	<u>city</u>	<u>state</u>
0736	New Age Books	1 1st St	Boston	MA
0877	Binnet & Hardley	2 2nd Ave.	Washington	DC
1389	Algodata Infosystems	3 3rd Dr.	Berkeley	CA
0010	Pragmatics	4 4th Ln.	Chicago	IL
1756	HealthText	NULL	NULL	NU

Если, например, все филиалы разместят свои центральные офисы в Атланте, шт. Джорджия, в таблицу потребуется внести изменения:

```
update publishers
set city = 'Atlanta', state = 'GA'
```

Теперь таблица примет следующий вид:

SQL:

```
select *
from publishers
```

Результат:

pub_id	pub_name	address	city	state
0736	New Age Books	1 1st St	Atlanta	GA
0877	Binnet & Hardley	2 2nd Ave.	Atlanta	GA
1389	Algodata Infosystems	3 3rd Dr.	Atlanta	GA
0010	Pragmatics	4 4th Ln.	Atlanta	GA
1756	HealthText	NULL	Atlanta	GA

(Кроме того, вы, вероятно, захотите изменить также и адреса.) Аналогично можно изменить имена всех издателей на "ZIPP!":

SQL:

```
update publishers
set pub_name = 'ZIPP!'
```

Теперь таблица примет следующий вид:

SQL:

```
select *
from publishers
```

Результат:

pub_id	pub_name	address	city	state
0736	ZIPP!	1 1st St	Atlanta	GA
0877	ZIPP!	2 2nd Ave.	Atlanta	GA
1389	ZIPP!	3 3rd Dr.	Atlanta	GA
0010	ZIPP!	4 4th Ln.	Atlanta	GA
1756	ZIPP!	NULL	Atlanta	GA

Кроме того, при обновлении данных со значениями столбцов можно выполнять математические преобразования. Чтобы увеличить в два раза все цены в таблице *titles*, можно воспользоваться оператором:

SQL:

```
update titles
set price = price * 2
```

Поскольку в этом операторе не используется предложение WHERE, будут изменены цены во всех строках таблицы.

Предложение WHERE

Предложение WHERE в операторе UPDATE определяет изменяемые строки. (Оно аналогично предложению WHERE в операторе SELECT, который будет подробно рассматриваться в главах 4–8.) Например, если вдруг северная Калифорния станет отдельным штатом с названием Pacifica (сокращенно PC), а жители Окленда решат переименовать свой город в Big Bad Bay City, то в таблицу *authors* потребуются внести следующие изменения:

SQL:

```
update authors
set state = 'PG', city = 'Big Bad Bay City'
where state = 'CA' and city = 'Oakland'
```

После этого таблица *authors* примет следующий вид:

SQL:

```
select au_fname, au_lname, city, state
from authors
```

Результат:

au_fname	au_lname	city	state
Abraham	Bennet	Berkeley	CA
Marjorie	Green	Big Bad Bay City	PC
Cheryl	Carson	Berkeley	CA
Albert	Ringer	Salt Lake City	UT
Anne	Ringer	Salt Lake City	UT
Michel	DeFrance	Gary	IN
Sylvia	Panteley	Rockville	MD
Heather	McBadden	Vacaville	CA
Dirk	Stringer	Big Bad Bay City	PC
Dick	Straight	Big Bad Bay City	PC
Livia	Karsen	Big Bad Bay City	PC
Stearns	MacFeather	Big Bad Bay City	PC
Ann	Dull	Palo Alto	CA
Akiko	Yokomoto	Walnut Creek	CA
Michael	O'Leary	San Jose	CA
Burt	Gringlesby	Covelo	CA
Morningstar	Greene	Nashville	TN
Johnson	White	Menlo Park	CA
Innes	del Castillo	Ann Arbor	MI
Sheryl	Hunter	Palo Alto	CA
Chastity	Locksley	San Francisco	CA
Reginald	Blotchet-Halls	Corvallis	OR
Meander	Smith	Lawrence	KS

Кроме того, потребуется выполнить дополнительные операторы UPDATE для авторов из других городов северной Калифорнии.

Предложение WHERE в операторе UPDATE может включать в себя подзапросы к одной или нескольким другим таблицам. Подзапросы описываются в главе 8.

УДАЛЕНИЕ ДАННЫХ: КОМАНДА DELETE

Не менее важной, чем добавление и изменение строк, является возможность их удаления. Подобно INSERT и UPDATE, команда DELETE позволяет манипулировать с одной или несколькими строками. Так же, как и в случае с другими операторами модификации данных, при удалении строк можно пользоваться информацией из других таблиц.

Оператор DELETE имеет следующий синтаксис:

```
DELETE FROM имя_таблицы
WHERE условие
```

В предложении WHERE определяются подлежащие удалению строки. Для удаления строки из таблицы *publishers* (издатель с идентификационным номером 1622) можно использовать следующую команду:

```
SQL:
delete from publishers
where pub_id = '1622'
```

После удаления строки, описывающей этого издателя, с помощью объединения таблиц *publishers* и *titles* по столбцу с идентификационными номерами уже не удастся найти названия изданных им книг.

В случае отсутствия предложения WHERE в операторе DELETE будут удалены все строки таблицы.

ПРИСТУПАЯ К ВЫБОРКЕ ДАННЫХ

Теперь вы полностью готовы к использованию нашей базы данных. На прилагаемом компакт-диске содержится все необходимое.

- Если вы будете пользоваться SQL Anywhere, то перед установкой этой системы и базы данных *bookbiz* прочитайте файл *readme*.
- Для воссоздания базы данных в своей СУБД, воспользуйтесь файлом *bookbiz.sql*. В нем содержатся все необходимые операторы CREATE и INSERT. Возможно, вам придется внести в него ряд изменений, отражающих синтаксические правила вашей системы. Например, названия типов данных меняются от системы к системе. Также вам может потребоваться изменить командный терминатор (в файле — GO).

Выборка информации из базы данных

ПЕРЕД ВЫБОРОМ

Оператор SELECT без преувеличения является сердцем SQL. Он позволяет выполнять поиск и представлять данные самыми разными способами. С его помощью можно ответить на вопросы о типе, количестве и расположении данных. Познакомившись с иногда достаточно запутанным синтаксисом команды SELECT, вы поймете, что она позволяет делать.

Поскольку оператор SELECT имеет такое большое значение, ему посвящаются пять следующих глав. В этой главе рассматриваются предложения SELECT, FROM и WHERE, условия и выражения. В главе 5 описываются ключевые слова ORDER BY и DISTINCT, а также агрегирующие функции. В главе 6 рассматриваются предложения GROUP BY и HAVING и описываются методы создания отчетов. В этой же главе суммируется информация об обработке нулевых значений в системах управления базами данных. Глава 7 посвящена многотабличным запросам и вопросам объединения таблиц. В главе 8 рассматриваются вложенные запросы (*nested queries*), также известные как подзапросы.

Все запросы в этой главе используют по одной таблице, так что вы сможете уделить все свое внимание их синтаксису.

Синтаксис оператора SELECT

Все дальнейшие усложнения начинаются со следующей конструкции оператора SELECT:

```
SELECT список_столбцов
FROM таблица[ы]
[WHERE условия]
```

SQL:

```
select address
from personnel
where name = 'Richard Roe'
```

<i>столбец1</i>	<i>столбец2</i>		
Name	Address		
Jane Doe	127 Elm St.	<i>строка1</i>	На пересечении строки и столбца приведен соответствующий адрес
Richard Roe	10 Trenholm Place	<i>строка2</i>	
Edgar Poe	1533 User House	<i>строка3</i>	

Рис. 4.1. Выборка данных из таблицы personnel

В списке столбцов указывается, из каких столбцов нужно выбрать данные. В списке таблиц определяется, в каких таблицах содержатся эти столбцы. В предложении WHERE указываются интересующие вас строки. Предложения SELECT

и WHERE могут содержать в себе как константы, так и выражения. С помощью комбинаций предложений SELECT, FROM и WHERE вы сможете получить ответы на интересующие вас вопросы и не утонуть в море информации.

Предложения SELECT и WHERE можно рассматривать в качестве горизонтальных и вертикальных осей матрицы, как показано на рис. 4.1.

Данные, выбираемые с помощью оператора SELECT, находятся на пересечении предложений SELECT (столбец) и WHERE (строка). В нашем случае — на пересечении строки 2 и столбца 2. Давайте рассмотрим использование оператора SELECT на примере таблицы *authors*.

В таблице *authors* хранится информация об авторах: их идентификационные номера, имена, адреса и номера телефонов. Чтобы узнать только имена авторов, живущих в Калифорнии (без адресов и номеров телефонов), нужно указать список соответствующих столбцов и использовать предложение WHERE для ограничения объема данных, возвращаемых оператором SELECT.

Ниже приводится запрос, в котором указываются конкретные столбцы для выборки данных. При этом извлекаются имена и фамилии авторов и игнорируются их идентификационные номера, адреса и телефоны.

SQL:

```
select au_lname, au_fname
from authors
```

Результат:

au_lname	au_fname
Bennet	Abraham
Green	Marjorie
Carson	Cheryl
Ringer	Albert
Ringer	Anne
DeFrance	Michel
Panteley	Sylvia
McBadden	Heather
Stringer	Dirk
Straight	Dick
Karsen	Livia
MacFeather	Stearns
Dull	Ann
Yokomoto	Akiko
O'Leary	Michael
Gringlesby	Burt
Greene	Morningstar
White	Johnson
del Castillo	Innes
Hunter	Sheryl
Locksley	Chastity
Blotch-Halls	Reginald
Smith	Meander

В результате получилось не совсем то, что мы хотели, так как в список попали все авторы, независимо от их места проживания. Для получения требуемого результата в оператор выборки данных необходимо ввести предложение WHERE.

SQL:

```
select au_lname, au_fname
from authors
where state = 'CA'
```

Результат:

au_lname	au_fname
Bennet	Abraham
Carson	Cheryl
McBadden	Heather
Dull	Ann
Yokomoto	Akiko
O'Leary	Michael
Gringlesby	Burt
White	Johnson
Hunter	Sheryl
Locksley	Chastity

Теперь в списке будут перечислены только авторы, проживающие в Калифорнии.

На практике синтаксис оператора SELECT может быть простым и сложным. Он упрощается, если в оператор включаются только предложения SELECT и FROM. Предложение WHERE (и все другие предложения) в принципе необязательны. С другой стороны, полный синтаксис оператора SELECT имеет следующий вид:

```
SELECT [ALL | DISTINCT] список_выбора
FROM {имя_таблицы | имя_курсора}
[ , {имя_таблицы | имя_курсора}]...
[WHERE условия]
[GROUP BY имя_столбца [ , имя_столбца]...]
[HAVING условия]
[ORDER BY {имя_столбца | список_выбора} [ASC | DESC]
[ , {column_name | список_выбора} [ASC | DESC] ]...]
```

Хотя SQL и является языком с безусловленной формой, порядок предложений в операторе SELECT должен строго соблюдаться (например, предложение GROUP BY должно предшествовать предложению ORDER BY). В противном случае это приведет к появлению ошибки синтаксиса.

Помимо этого, нужно однозначно описывать все объекты базы данных (в соответствии с конкретным диалектом SQL). Например, если в базе данных содержится несколько столбцов с именем *notes*, нужно точно указать, какой из них вы имеете в виду. Для этого в соответствующий запрос нужно включить имя базы данных, таблицы или курсора и имя владельца, например:

```
база_данных.владелец.имя_таблицы.notes
база_данных.владелец.имя_курсора.notes
```

Все примеры запросов в этой главе используют по одной таблице, поэтому такие уточнения вам не потребуются. О них также не упоминается в большинстве книг, статей и руководств по SQL, так как более краткая форма упрощает понимание операторов SELECT. Тем не менее не забывайте включать их в запросы в случае необходимости.

ВЫБОР СТОЛБЦОВ: СПИСОК ВЫБОРА

Любой оператор SELECT начинается с ключевого слова SELECT. Ключевые слова ALL и DISTINCT, которые определяют, будут ли включаться в результат повторяющиеся строки, являются необязательными и описываются в следующей главе.

В списке выбора (select list) определяется столбец или столбцы, включаемые в результат. Он может состоять из имен одного или нескольких столбцов, или включать единственную звездочку (*), определяющую все столбцы. Можно также использовать выражения — константы, имена столбцов, функций и их комбинации с арифметическими операторами и скобками. Вот несколько примеров выражений:

```
ytd_sales * price
price * 1.2
(12000 - 500)/ 13
avg(advance)
```

Каждый элемент в списке выбора отделяется от другого запятой.

Выбор всех столбцов: SELECT *

Знак звездочки (*) играет в списке выбора особую роль. Он означает выбор *всех имен столбцов во всех таблицах* из списка таблиц. Столбцы отображаются в соответствии с порядком, в котором они были определены в операторе (операторах) CREATE TABLE. Этот знак используется, если необходимо увидеть все столбцы таблицы.

Для выбора всех столбцов таблиц используется следующий оператор:

```
SELECT *
FROM список_таблиц
```

Так как оператор SELECT * находит все текущие столбцы таблицы, при изменении структуры таблицы (добавление, удаление или переименование столбцов) соответственно изменяются и его результаты. Просмотр столбцов по отдельности, конечно, позволяет более тщательно контролировать получаемые результаты, однако вводить оператор SELECT * значительно проще (к тому же в нем сложно сделать ошибку). Оператор SELECT * наиболее подходит для таблиц с несколькими столбцами, так как одновременное отображение данных из большого количества столбцов обычно неудобно. Также этот оператор будет полезен, если вы хотите получить общие представления о структуре таблицы (о столбцах и их порядке).

Следующий оператор извлекает все столбцы из таблицы *publishers* и отображает их в соответствии с порядком, в котором они были определены при создании таблицы. Поскольку в этом запросе не используется предложение WHERE, из таблицы будут выбраны все строки.

SQL:

```
select *
from publishers
```

Результат:

pub_id	pub_name	address	city	state
0736	New Age Books	1 1st St	Boston	MA
0877	Binnet & Hardley	2 2nd Ave.	Washington	DC
1389	Algodata Infosystems	3 3rd Dr.	Berkeley	CA

Тот же результат получается и при перечислении всех имен столбцов этой таблицы после ключевого слова SELECT:

SQL:

```
select pub_id, pub_name, address, city, state
from publishers
```

Звездочка в списке выбора многотабличного запроса заставляет SQL отображать все столбцы из всех таблиц в списке таблиц. В некоторых системах в списке выбора могут одновременно присутствовать и звездочка, и имена отдельных столбцов. Это наиболее удобно в многотабличных запросах, когда вместе со звездочкой используется имя таблицы. Следующий оператор извлекает информацию из таблицы *publishers* для каждого значения *title_id* из таблицы *titles*. При этом будут извлекаться все столбцы из таблицы *publishers* (так как в списке выбора присутствует *publishers.**) и только один столбец из таблицы *titles* (так как в списке выбора присутствует *title_id*).

SQL:

```
SELECT title_id, publishers.*
from titles, publishers
where titles.pub_id = publishers.pub_id
```

Результат:

title_id	pub_id	pub_name	address	city	state
PS1372	0736	New Age Books	1 1st St	Boston	MA
PS7777	0736	New Age Books	1 1st St	Boston	MA
PS3333	0736	New Age Books	1 1st St	Boston	MA
PS2106	0736	New Age Books	1 1st St	Boston	MA
PS2091	0736	New Age Books	1 1st St	Boston	MA
BU2075	0736	New Age Books	1 1st St	Boston	MA
MC3026	0877	Binnet & Hardley	2 2nd Ave.	Washington	DC
MC2222	0877	Binnet & Hardley	2 2nd Ave.	Washington	DC
TC7777	0877	Binnet & Hardley	2 2nd Ave.	Washington	DC
TC4203	0877	Binnet & Hardley	2 2nd Ave.	Washington	DC
MC3021	0877	Binnet & Hardley	2 2nd Ave.	Washington	DC
TC3218	0877	Binnet & Hardley	2 2nd Ave.	Washington	DC
PC8888	1389	Algodata Infosystems	3 3rd Dr.	Berkeley	CA
PC9999	1389	Algodata Infosystems	3 3rd Dr.	Berkeley	CA
BU1111	1389	Algodata Infosystems	3 3rd Dr.	Berkeley	CA
BU7832	1389	Algodata Infosystems	3 3rd Dr.	Berkeley	CA
BU1032	1389	Algodata Infosystems	3 3rd Dr.	Berkeley	CA
PC1035	1389	Algodata Infosystems	3 3rd Dr.	Berkeley	CA

Без использования звездочки этот запрос имел бы следующий вид:

SQL:

```
select title_id, publishers.pub_id, publishers.pub_name, publish-
ers.address, publishers.city, publishers.state
from titles, publishers
where titles.pub_id = publishers.pub_id
```

(Вопросы, связанные с операцией объединения, будут рассматриваться в главе 7. Поэтому сейчас не обращайтесь особого внимания на вид этого запроса.)

Выбор отдельных столбцов

Для выбора подмножества столбцов таблицы нужно просто перечислить их в списке выбора, как это делалось в предыдущем примере:

```
SELECT имя_столбца [ , имя_столбца]...  
FROM имя_таблицы
```

Столбцы в списке выбора разделяются запятыми.

Оформление результата. Порядок вывода столбцов полностью зависит от вас: в списке выбора их можно указать в любом нужном вам порядке.

Ниже приводятся два примера. В результате выполнения обоих запросов выводятся имена и идентификационные номера издателей из всех трех строк таблицы *publishers*. В первом запросе сначала выводится столбец *pub_id*, а затем *pub_name*. Во втором запросе эти столбцы выводятся в обратном порядке.

SQL:

```
select pub_id, pub_name  
from publishers
```

Результат:

pub_id	pub_name
0736	New Age Books
0877	Binnet & Hardley
1389	Algodata Infosystems

SQL:

```
select pub_name, pub_id  
from publishers
```

Результат:

pub_name	pub_id
New Age Books	0736
Binnet & Hardley	0877
Algodata Infosystems	1389

Выражения: больше, чем просто имена столбцов

До сих пор с помощью операторов `SELECT` мы отображали только содержащиеся в таблицах данные. Это полезно, но не всегда достаточно. `SQL` позволяет манипулировать результатами, делая их вполне понятными. В списке выбора можно использовать символьные строки, математические действия и функции, реализованные в вашей системе.

Переименование столбцов и задание имен выражениям. При выводе результатов запроса каждый столбец по умолчанию получает заголовок, совпадающий с его именем в базе данных. Столбцы в базах данных обычно имеют сокращенные названия (чтобы их легче было набирать) и могут быть непонятны пользователям, незнакомым с используемым в базе данных жаргоном.

Для упрощения чтения и понимания результатов запроса можно переопределить заголовки столбцов. Чтобы получить необходимые имена заголовков, просто введите `имя_столбца имя_заголовка` или `имя_столбца as имя_заголовка` в списке выбора вместо обычных имен столбцов. (В некоторых системах используется другая форма записи в виде `заголовок_столбца = имя_столбца`).

Например, для изменения заголовка *pub_name* на *Publisher* попробуйте выполнить один из следующих операторов:

```
SQL:
select pub_name Publisher, pub_id
from publishers
```

```
SQL:
select pub_name as Publisher, pub_id
from publishers
```

В результате выполнения этого запроса изменится заголовок столбца:

Результат:

Publisher	pub_id
New Age Books	0736
Binnet & Hardley	0877
Algodata Infosystems	1389

Размер заголовка не ограничивается размером данных соответствующего столбца. Например, столбец *pub_id* может иметь заголовок более чем из четырех символов. Вот что получится, если изменить заголовок этого столбца на *Identification#*.

```
SQL:
select pub_name as Publisher, pub_id as Identification#
from publishers
```

Результат:

Publisher	Identification#
New Age Books	0736
Binnet & Hardley	0877
Algodata Infosystems	1389

В большинстве систем ширина отображаемых столбцов устанавливается по максимальной длине заголовка, однако, если вы используете короткие заголовки, столбцы все равно не будут уже находящихся в них данных.

Большинство диалектов SQL, позволяющих определять заголовки столбцов, имеют ряд ограничений (за деталями обратитесь к руководству по своей системе). Как правило, в заголовках нельзя использовать кавычки и пробелы.

Таким же образом можно создавать заголовки и для столбцов, использующих различные вычисления и выражения, например *New_price*, *Double_Advance* и т.д.:

```
SQL:
select title, advance*2 as Double_Advance
from titles
```

Символьные строки в результатах запроса. Иногда, чтобы сделать более понятными результаты запроса, имеет смысл добавить к ним небольшие пояснения. Здесь на помощь приходят символьные строки.

Например, перед именами издателей можно добавить текст типа “The publisher’s name is” (Имя издателя). Для этого данную строку нужно вставить в список выбора, взяв в двойные или одинарные кавычки, чтобы система не посчитала ее названием столбца, и отделить от других элементов списка выбора запятыми.

Если в строке имеется апостроф, нужно строго следовать правилам конкретной системы. Например, чтобы апостроф не рассматривался системой как закрывающаяся кавычка, в нашем случае используется вторая одинарная кавычка.

SQL:

```
select 'The publisher' 's name is', pub_name as Publisher
from publishers
```

Результат:

	Publisher
The publisher's name is	New Age Books
The publisher's name is	Binnet & Hardley
The publisher's name is	Algodata Infosystems

В результате выполнения этого запроса создается новый столбец, однако, то, что вы видите на экране, никак не влияет на физическую структуру базы данных.

Кроме того, каждое слово символьной строки можно превратить в отдельное поле:

SQL:

```
select 'The', 'publisher', 'name', 'is', pub_name
from publishers
```

Результат:

				pub_name
The	publisher	name	is	New Age Books
The	publisher	name	is	Binnet & Hardley
The	publisher	name	is	Algodata Infosystems

С помощью такого подхода можно комбинировать столбцы и текст, например:

SQL:

```
select 'The name for publisher #', pub_id, 'is', pub_name
from publishers
```

Результат:

	pub_id		pub_name
The name for publisher #	0736	is	New Age Books
The name for publisher #	0877	is	Binnet & Hardley
The name for publisher #	1389	is	Algodata Infosystems

Вычисления с использованием констант. В списке выбора с числовыми данными и константами можно выполнять арифметические действия.

Ниже приводится список допустимых арифметических операторов:

Символ	Операция
+	сложение
-	вычитание
/	деление
*	умножение

Арифметические операторы — сложение, вычитание, деление и умножение — могут применяться к любым числовым столбцам. (В некоторых системах добавляется еще одна операция — остаток от деления (*modulo*), представляемая символом %. Результатом выполнения этой операции является остаток от деления двух целых чисел. Например, $21\%9 = 3$, так как при делении 21 на 9 в остатке получается 3.)

Если в вашей системе реализованы функции обработки данных, то некоторые арифметические операции могут выполняться и на нецифровых столбцах.

В списке выбора эти операции могут использоваться в любых комбинациях с именами столбцов и числовыми константами. Например, чтобы отобразить в таблице *titles* повышение стоимости всех книг на 100%, можно выполнить следующий запрос:

```
SQL:
select title_id, ytd_sales, ytd_sales * 2
from titles
```

В результате получится следующая таблица:

Результат:

title_id	ytd_sales	
PC8888	4095	8190
BU1032	4095	8190
PS7777	3336	6672
PS3333	4072	8144
BU1111	3876	7752
MC2222	2032	4064
TC7777	4095	8190
TC4203	15096	30192
PC1035	8780	17560
MC3026	NULL	NULL
BU2075	18722	37444
PS2091	2045	4090
PS2106	111	222
MC3021	22246	44492
TC3218	375	750
BU7832	4095	8190
PS1372	375	750
PC9999	NULL	NULL

Обратите внимание на нулевые значения в *ytd_sales* и производном столбце. Любые арифметические операции над нулевыми значениями в результате дают NULL.

Производному столбцу можно задать заголовок (например, *Projected_Sales*):

```
SQL:
select title_id, ytd_sales, ytd_sales * 2 as Projected_Sales
from titles
```

Иногда, как это было в предыдущем примере, в результат включаются данные как исходного, так и производного столбца. Однако включать в список выбора столбец, на основе которого выполнялись вычисления, необязательно. Чтобы увидеть только вычисленные значения, можно выполнить следующую команду:

```
SQL:
select title_id, ytd_sales * 2
from titles
```

Результат:

title_id	
PC8888	8190
BU1032	8190
PS7777	6672

PS3333	8144
BU1111	7752
MC2222	4064
TC7777	8190
TC4203	30192
PC1035	17560
MC3026	NULL
BU2075	37444
PS2091	4090
PS2106	222
MC3021	44492
TC3218	750
BU7832	8190
PS1372	750
PC9999	NULL

Вычисления с использованием имен столбцов. Арифметические операции также можно выполнять над данными двух и более столбцов. Например:

SQL:

```
select title_id, ytd_sales * price
from titles
```

Результат:

title_id	
PC8888	81900.0000
BU1032	81859.0500
P57777	26654.6400
P53333	81399.2800
BU1111	46318.2000
MC2222	40619.6800
TC7777	61384.0500
TC4203	180397.2000
PC1035	201501.0000
MC3026	(NULL)
BU2075	55978.7800
P52091	22392.7500
P52106	777.0000
MC3021	66515.5400
TC3218	7856.2500
BU7832	81859.0500
P51372	8096.2500
PC9999	(NULL)

Наконец, можно получать требуемые значения на основе столбцов из разных таблиц. (Многотабличные запросы рассматриваются в главе, посвященной операциям объединения таблиц и подзапросам.)

Порядок выполнения арифметических операторов. Если в выражении имеется несколько арифметических операторов, они будут выполняться в порядке, установленном в системе (рис. 4.2). В соответствии с общепринятыми правилами сначала вы-

полняется умножение и деление, затем вычитание и сложение. Если в выражении присутствует несколько операторов с одинаковым приоритетом, они выполняются слева направо. Максимальный приоритет имеют выражения, взятые в скобки.

Рассмотрим пример. В следующем операторе SELECT сначала вычисляется произведение *ytd_sales* и *price*, так как операция умножения имеет более высокий приоритет, чем операция вычитания.

SQL:

```
select title_id, ytd_sales * price - advance
from titles
```

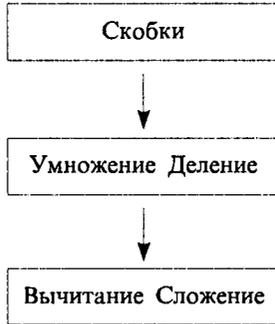


Рис. 4.2. Иерархия арифметических операторов

Во избежание недоразумений используйте скобки. При выполнении следующего запроса получаются те же результаты, но его запись может показаться более понятной:

SQL:

```
select title_id, (ytd_sales * price) - advance
from titles
```

Результат:

title_id	
PC8888	73900.0000
BU1032	76859.0500
P57777	22654.6400
P53333	79399.2800
BU1111	41318.2000
MC2222	40619.6800
TC7777	53384.0500
TC4203	176397.2000
PC1035	194501.0000
MC3026	(NULL)
BU2075	45853.7800
P52091	20117.7500
P52106	-5223.0000
MC3021	51515.5400
TC3218	856.2500
BU7832	76859.0500
P51372	1096.2500
PC9999	(NULL)

Скобки также используются для изменения порядка выполнения операций, поскольку заключенные в них операторы выполняются в первую очередь. В случае вложенных скобок (одна пара скобок внутри другой пары) первыми выполняются операторы, имеющие максимальную глубину вложения. Например, результаты предыдущего запроса изменятся, если переставить скобки таким образом, чтобы сначала выполнялась операция вычитания:

```
SQL:
select title_id, ytd_sales * (price - advance)
from titles
```

Результат:

<u>title_id</u>	
PC8888	-32678100.000
BU1032	-20393140.950
P57777	-13317345.360
P53333	-8062600.720
BU1111	-19333681.800
MC2222	40619.680
TC7777	-32698615.950
TC4203	-60203602.800
PC1035	-61258499.000
MC3026	(NULL)
BU2075	-189504271.220
P52091	-4629982.250
P52106	-665223.000
MC3021	-333623484.460
TC3218	-2617143.750
BU7832	-20393140.950
P51372	-2616903.750
PC9999	(NULL)

УКАЗАНИЕ ТАБЛИЦ: СПИСОК ТАБЛИЦ

В списке таблиц (*table list*) указываются имена таблиц и курсоров, содержащих столбцы из списка выбора и предложения WHERE. Имена таблиц в списке разделяются запятыми. Ниже приведен общий вид предложения FROM:

```
SELECT список_выбора
FROM [квалификатор] {имя_таблицы | имя_курсора}
[ , [квалификатор] { имя_таблицы | имя_курсора} ]...
```

В списке таблиц также можно указать соответствующие таблицам базы данных и их владельцев. Необходимость в этом может возникнуть, если в запросе используются одноименные таблицы из разных баз данных.

В большинстве диалектов SQL для упрощения набора таблицам допускается задавать псевдонимы (*aliases*). Псевдоним указывается после имени таблицы в списке таблиц:

```
SQL:
select p.pub_id, p.pub_name
from publishers p
```

Буква *p* перед именами столбцов заменяет полное имя таблицы (*publishers*). Этот запрос эквивалентен следующему:

SQL:

```
select publishers.pub_id, publishers.pub_name
from publishers
```

Так как в каждом запросе используется только одна таблица, и при использовании столбца *pub_id* не возникает никакой неопределенности, имя таблицы можно опустить. Псевдонимы действительно полезны в многотабличных запросах, когда нужно различать одноименные столбцы из разных таблиц. Примеры их использования содержатся в главах 7 и 8.

ВЫБОР СТРОК: ПРЕДЛОЖЕНИЕ WHERE

Предложение WHERE является частью оператора SELECT и позволяет определить условия для выборки строк. Общий формат оператора имеет следующий вид:

```
SELECT список_выбора
FROM список_таблиц
WHERE условия
```

При выполнении оператора SELECT с предложением WHERE выбираются все строки, удовлетворяющие наложенному условию.

В SQL имеется целый ряд операторов и ключевых слов для задания условий.

- Операторы сравнения (=, <, > и т.д.)
where advance * 2 > ytd_sales * price
- Комбинации условий и логическое отрицание (AND, OR, NOT)
where advance < 5000 or ytd_sales > 2000
- Диапазоны (BETWEEN и NOT BETWEEN)
where ytd_sales between 4095 and 12000
- Списки (IN, NOT IN)
where state in ('CA', 'IN', 'MD')
- Неизвестные значения (IS NULL и IS NOT NULL)
where advance is null
- Соответствия символов (LIKE и NOT LIKE)
where phone not like '415%'

Все эти ключевые слова и операторы будут описаны и проиллюстрированы дальше в этой главе.

Кроме того, в предложение WHERE могут включаться условия на объединение (глава 7) и подзапросы (глава 8).

Операторы сравнения

Часто в приложениях возникает необходимость провести сравнительный анализ некоторых значений из базы данных: выяснить, какое из них больше или меньше (в числовом или алфавитном порядке). Для этих целей в SQL предусмотрен целый ряд операторов сравнения. В большинстве диалектов имеются следующие операторы сравнения:

Оператор	Значение
=	Равно
>	Больше
<	Меньше
>=	Больше или равно

Оператор	Значение
<=	Меньше или равно
!=	Не равно
<>	Не равно

Эти операторы используются следующим образом:

WHERE выражение оператор_сравнения выражение

В качестве выражений могут использоваться константы, имена столбцов, функции, подзапросы или их комбинации, связанные арифметическими операторами.

Обычно операторы сравнения применяются к числовым значениям. В SQL они также могут применяться к данным с типами *char* и *varchar* (< означает раньше в алфавитном порядке, > означает позже) и к датам (< означает раньше в хронологическом порядке, > означает позже). При использовании символьных значений и дат в SQL их нужно заключать в кавычки.

Порядок следования строчных, прописных и специальных символов определяется используемой вами кодовой таблицей символов и может отличаться в разных системах. В большинстве систем при сравнении игнорируются все концевые пробелы. Например, значения "Dirk" и "Dirk " считаются одинаковыми. На примере следующих операторов SELECT можно увидеть, как работают операторы сравнения. Первый запрос используется для поиска книг стоимостью выше \$15:

SQL:

```
select title, price
from titles
where price > $15.00
```

Результат:

title	price
Secrets of Silicon Valley	20
The Busy Executive's Database Guide	19.99
Prolonged Data Deprivation: Four Case Studies	19.99
Silicon Valley Gastronomic Treats	19.99
But Is It User Friendly?	22.95
Onions, Leeks, and Garlic: Cooking Secrets of the Mediterranean	20.95
Straight Talk About Computers	19.99
Computer Phobic and Non-Phobic Individuals: Behavior Variations	21.59

В результате выполнения второго запроса находятся авторы с фамилиями, идущими в алфавитном порядке после фамилии McBadden:

SQL:

```
select au_lname, au_fname
from authors
where au_lname > 'McBadden'
```

Результат:

au_lname	au_fname
Ringer	Albert
Ringer	Anne
Panteley	Sylvia
Stringer	Dirk
Straight	Dick

Yokomoto	Akiko
O'Leary	Michael
White	Johnson
Smith	Meander

(Полученные вами результаты могут отличаться от приведенных выше, в зависимости от метода сортировки, используемого в вашей системе. За подробностями обращайтесь к главе 5.) Следующий запрос используется для вывода гипотетической информации — удвоенной стоимости книг, затраты на которые привысили \$10000 вместе с их идентификационными номерами:

SQL:

```
select title_id, price * 2
from titles
where advance > $10000
```

Результат:

title_id	
BU2075	5.98
MC3021	5.98

Вот пример запроса, использующего оператор сравнения “не равен” для поиска телефонных номеров авторов, живущих за пределами Калифорнии (в разных версиях SQL для записи этого оператора могут применяться знаки != или <>):

SQL:

```
select au_id, phone
from authors
where state != 'CA'
```

Результат:

au_id	phone
998-72-3567	801 826-0752
899-46-2035	801 826-0752
722-51-5454	219 547-9982
807-91-6654	301 946-8853
527-72-3246	615 297-2723
712-45-1867	615 996-8275
648-92-1872	503 745-6402
341-22-1782	913 843-0462

Совместное использование условных и логических операторов

Если в предложение WHERE нужно поместить несколько условий, то для их соединения можно использовать логические операторы (logical operators) AND, OR и NOT, называемые также булевыми операторами.

Оператор AND объединяет два и более условий и возвращает истинное значение только при выполнении всех условий. Например, следующий запрос находит только авторов с фамилией Ringer и именем Anne. Albert Ringer в этот список не попадет.

SQL:

```
select *
from authors
```

```
where au_lname = 'Ringer'
and au_fname = 'Anne'
```

В следующем примере находятся все книги по бизнесу стоимостью выше \$10 и с затратами ниже \$20000:

```
SQL:
select title, type, price, advance
from titles
where type = 'business'
and price > $10.00
and advance < $20000
```

Результат:

title	type	price	advance
The Busy Executive's Database Guide	business	19.99	5000
Cooking with Computers: Surreptitious Balance Sheets	business	11.95	5000
Straight Talk About Computers	business	19.99	5000

Оператор OR также связывает два или больше условий, но возвращает истинный результат при выполнении хотя бы одного условия. Следующий запрос предназначен для поиска строк, содержащих в столбце *au_fname* значения Anne или Ann:

```
SQL:
select au_id, au_lname, au_fname
from authors
where au_fname = 'Anne'
or au_fname = 'Ann'
```

Результат:

au_id	au_lname	au_fname
899-46-2035	Ringer	Anne
427-17-2319	Dull	Ann

В результате выполнения следующего запроса находятся все книги стоимостью выше \$20 и затратами меньше \$5000.

```
SQL:
select title, type, price, advance
from titles
where price > $20.00
or advance < $5000
```

Результат:

title	type	price	advance
Emotional Security: A New Algorithm	psychology	7.99	4000
Prolonged Data Deprivation: Four Case Studies	psychology	19.99	2000
Silicon Valley Gastronomic Treats	mod_cook	19.99	0
Fifty Years in Buckingham Palace Kitchens	trad_cook	11.95	4000
But Is It User Friendly?	popular_comp	22.95	7000
Is Anger the Enemy?	Psychology	10.95	2275
Onions, Leeks, and Garlic: Cooking Secrets of the Mediterranean	trad_cook	20.95	7000
Computer Phobic and Non-Phobic Individuals: Behavior Variations	psychology	21.59	7000

Следующий пример отражает потенциальные проблемы, которые могут возникнуть при использовании оператора OR. Предположим, что нужно найти все книги по бизнесу, а также все книги стоимостью выше \$10 и все книги с затратами меньше \$20000. В обычном языке для описания этого набора используется связка AND (И), тогда как в SQL нужно использовать оператор OR, потому что нужно найти книги из всех трех категорий, а не только книги, удовлетворяющие всем этим условиям одновременно. Ниже приведен соответствующий оператор SQL:

SQL:

```
select title, type, price advance
from titles
where type = 'business'
      or price > $10.00
      or advance < $20000
```

Результат:

title	type	price	advance
Secrets of Silicon Valley	popular_comp	20	8000
The Busy Executive's Database Guide	business	19.99	5000
Emotional Security: A New Algorithm	psychology	7.99	4000
Prolonged Data Deprivation: Four Case Studies	psychology	19.99	2000
Cooking with Computers: Surreptitious Balance Sheets	business	11.5	5000
Silicon Valley Gastronomic Treats	mod_cook	19.99	0
Sushi, Anyone?	Trad_cook	14.99	8000
Fifty Years in Buckingham Palace Kitchens	trad_cook	11.95	4000
But Is It User Friendly?	Popular_comp	22.95	7000
You Can Combat Computer Stress!	Business	2.99	10125
Is Anger the Enemy?	Psychology	10.95	2275
Life Without Fear	psychology	7	6000
The Gourmet Microwave	mod_cook	2.99	15000
Onions, Leeks, and Garlic: Cooking Secrets of the Mediterranean	trad_cook	20.95	7000
Straight Talk About Computers	business	19.99	5000
Computer Phobic and Non-Phobic Individuals: Behavior Variations	psychology	21.59	7000

Логический оператор NOT используется для построения отрицаний и помещается перед самим выражением. Следующие два запроса эквивалентны:

SQL:

```
select au_lname, au_fname
from authors
where state != 'CA'
```

SQL:

```
select au_lname, au_fname, state
from authors
where not state = 'CA'
```

В результате получится следующий список:

Результат:

au_lname	au_fname	state
Ringer	Albert	UT
Ringer	Anne	UT
DeFrance	Michel	IN
Panteley	Sylvia	MD
Greene	Morningstar	TN
del Castillo	Innes	MI
Blotchet-Halls	Reginald	OR
Smith	Meander	KS

Порядок выполнения логических операторов. Аналогично операторам сравнения, логические операторы также выполняются в строго определенной последовательности. Если в выражении присутствуют оба типа операторов, первыми выполняются арифметические операторы. Если в выражении используется несколько логических операторов, они выполняются в следующей последовательности: сначала NOT, затем AND и наконец OR. Эта иерархия представлена на рис. 4.3.

Рассмотрим несколько примеров. Следующий запрос используется для поиска всех книг по бизнесу в таблице *titles*, независимо от величины затрат на них, а также книги по психологии с затратами больше \$5500. Ограничение на затраты применяется только к книгам по психологии, так как оператор AND выполняется перед оператором OR.

SQL:

```
select title_id, type, advance
from titles
where type = 'business'
or type = 'psychology'
and advance > $5500
```

Результат:

titles_id	type	advance
BU1032	business	5000
BU1111	business	5000
BU2075	business	10125
PS2106	psychology	6000
BU7832	business	5000
PS1372	psychology	7000

Поскольку логические операторы выполняются в определенной последовательности, в результирующий список будут включены три книги по бизнесу с расходами меньше \$5500. Другими словами, сначала будут найдены все книги по психологии с расходами больше \$5500, а затем все без ограничений книги по бизнесу.

Чтобы первым выполнялся оператор OR, в запросе нужно использовать скобки. В результате выполнения следующего запроса будут найдены все книги по бизнесу и психологии с затратами выше \$5500:

SQL:

```
select title_id, type, advance
```

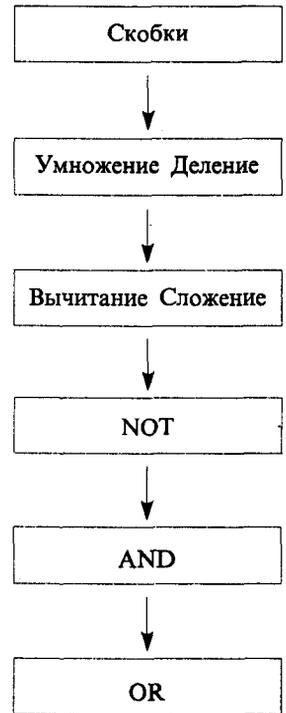


Рис. 4.3. Порядок выполнения логических операторов

```
from titles
where (type = 'business'
or type = 'psychology')
and advance > $5500
```

Результат:

title_id	type	advance
BU2075	business	10125
PS2106	psychology	6000
PS1372	psychology	7000

Благодаря скобкам сначала осуществляется поиск всех книг по бизнесу и психологии, а затем из них выбираются книги с затратами больше \$5500.

Ниже приводится пример запроса, в котором одновременно используются арифметические, логические и операторы сравнения. Этот запрос используется для поиска книг, затраты на которые не окупились. Другими словами, находятся все книги, доход от продажи которых (т.е. произведение *ytd_sales* на *price*) меньше удвоенной суммы, выплаченной авторам. Кроме того, в нашем случае за точку отсчета берется 15 октября 1985 года, чтобы зафиксировать достаточный срок, в течение которого должен быть разойтись тираж. Это условие присоединяется к запросу с помощью оператора AND и в соответствии с правилами вычисляется после арифметических операций.

SQL:

```
select title_id, type, price, advance, ytd_sales
from titles
where price * ytd_sales < 2 * advance
and pubdate < '10/15/85'
```

Результат:

title_id	type	price	advance	ytd_sales
PS2106	psychology	7	6000	111

Диапазоны (BETWEEN и NOT BETWEEN)

Другим средством реализации условий являются диапазоны. Определить диапазон можно двумя способами:

- с помощью операторов сравнения > и <;
- с помощью ключевого слова BETWEEN.

Ключевое слово BETWEEN можно использовать для задания **включающего диапазона (inclusive range)**, если в искомые значения должны включаться и границы диапазона. Например, для поиска всех книг с количеством проданных экземпляров между 4095 и 12000 (включительно) можно использовать следующий запрос:

SQL:

```
select title_id, ytd_sales
from titles
where ytd_sales between 4095 and 12000
```

Результат:

title_id	ytd_sales
PC8888	4095
BU1032	4095

TC7777	4095
PC1035	8780
BU7832	4095

Обратите внимание, что в результат включаются книги с 4095 проданными экземплярами. Также туда были бы включены и книги с 12000 проданными экземплярами, если бы в таблице имелись соответствующие данные. В этом смысле диапазон BETWEEN отличается от диапазона больше-меньше (><). При выполнении предыдущего запроса с использованием диапазона больше-меньше получаются другие результаты, так как при поиске не учитываются границы диапазона:

SQL:

```
select title_id, ytd_sales
from titles
where ytd_sales > 4095 and ytd_sales < 12000
```

Результат:

title_id	ytd_sales
PC1035	8780

При использовании конструкции NOT BETWEEN находятся все строки, не входящие по своим значениям в указанный диапазон. Так, для поиска всех книг с количеством проданных экземпляров вне диапазона от 4095 до 12000 можно использовать следующий запрос:

SQL:

```
select title_id, ytd_sales
from titles
where ytd_sales not between 4095 and 12000
```

Результат:

title_id	ytd_sales
PS7777	3336
PS3333	4072
BU1111	3876
MC2222	2032
TC4203	15096
BU2075	18722
PS2091	2045
PS2106	111
MC3021	22246
TC3218	375
PS1372	375

Тот же результат можно получить с помощью операторов сравнения, однако обратите внимание, что в следующем запросе вместо оператора AND используется оператор OR:

SQL:

```
select title_id, ytd_sales
from titles
where ytd_sales < 4095 or ytd_sales > 12000
```

Результат:

title_id	ytd_sales
PS7777	3336
PS3333	4072
BU1111	3876
MC2222	2032
TC4203	15096
BU2075	18722
PS2091	2045
PS2106	111
MC3021	22246
TC3218	375
PS1372	375

Это еще один пример, когда могут возникнуть определенные затруднения. На обычном языке суть запроса формулируется следующим образом: нужно найти все книги с количеством проданных экземпляров до 4095 и все книги с количеством проданных экземпляров больше 12000. Хотя мы говорим “и”, в запросе нужно использовать оператор OR (или). Если вместо него применить оператор AND, то в результате не будет найдена ни одна книга, так как не может быть продано экземпляров одной книги меньше 4095 и одновременно больше 12000.

Списки (IN и NOT IN)

Ключевое слово IN позволяет выбрать значения, совпадающие со значениями из заданного списка. Например, без использования IN для получения списка авторов, проживающих в Калифорнии, Индиане и Мериленде, нужно было бы выполнить следующий запрос:

SQL:

```
select au_lname, state
from authors
where state = 'CA' or state = 'IN' or state = 'MD'
```

Этот же результат можно получить с использованием ключевого слова IN. Значения, идущие после IN, должны браться в кавычки, разделяться запятыми и заключаться в скобки.

SQL:

```
select au_lname, state
from authors
where state in ('CA', 'IN', 'MD')
```

Оба предыдущих запроса дают одинаковый результат:

Результат:

au_lname	state
Bennet	CA
Green	CA
Carson	CA
DeFrance	IN
Panteley	MD
McBadden	CA
Stringer	CA

Straight	CA
Karsen	CA
MacFeather	CA
Dull	CA
Yokomoto	CA
O'Leary	CA
Gringlesby	CA
White	CA
Hunter	CA
Locksley	CA

Чем больше элементов в списке, тем значительнее становится экономия времени при использовании ключевого слова IN, так как отпадает необходимость во вводе многочисленных операторов сравнения.

Возможно, наиболее полно преимущества ключевого слова IN проявляются во вложенных запросах, также называемых подзапросами. Подзапросы подробно обсуждаются в главе 8. Тем не менее, рассмотрев следующий пример, вы поймете, как в подзапросах можно использовать ключевое слово IN.

Предположим, нужно найти имена авторов, которые получают меньше 50 процентов от суммарного гонорара за книги, написанные ими в соавторстве. Имена авторов содержатся в таблице *authors*, информация о распределении гонораров хранится в таблице *titleauthors*. Ключевое слово IN позволяет как бы объединить обе таблицы (без получения общей таблицы) и извлечь при этом нужную информацию.

SQL:

```
select au_lname, au_fname
from authors
where au_id in
(select au_id
 from titleauthors
 where royaltyshare < .50)
```

Результат:

au_lname	au_fname
Green	Marjorie
Ringer	Anne
MacFeather	Stearns
Yokomoto	Akiko
O'Leary	Michael
Gringlesby	Burt

Таким образом, шесть авторов хотя бы за одну книгу получали меньше 50 процентов от общего гонорара.

Конструкция NOT IN позволяет найти авторов, не удовлетворяющих условиям, перечисленным в списке. С помощью следующего запроса можно найти всех авторов, которые хотя бы за одну книгу получили больше половины от общего гонорара.

SQL:

```
select au_lname, au_fname
from authors
where au_id not in
(select au_id
 from titleauthors
 where royaltyshare < .50)
```

Результат:

au_lname	au_fname
Bennet	Abraham
Carson	Cheryl
Ringer	Albert
DeFrance	Michel
Panteley	Sylvia
McBadden	Heather
Stringer	Dirk
Straight	Dick
Karsen	Livia
Dull	Ann
Greene	Morningstar
White	Johnson
del Castillo	Innes
Hunter	Sheryl
Locksley	Chastity
Blotchet-Halls	Reginald
Smith	Meander

Выборка нулевых значений

Как вы помните из предыдущих глав, значение NULL используется для представления неизвестной информации. И это не то же самое, что обычный ноль или пустая позиция.

Чтобы прояснить эти различия, рассмотрим следующий листинг, в котором представлены названия и расходы по книгам, принадлежащим одному издателю.

SQL:

```
select title, advance
from titles
where pub_id = '0877'
```

Результат:

title	advance
Silicon Valley Gastronomic Treats	0
Sushi, Anyone?	8000
Fifty Years in Buckingham Palace Kitchens	4000
The Psychology of Computer Cooking	NULL
The Gourmet Microwave	15000
Onions, Leeks, and Garlic: Cooking Secrets of the Mediterranean	7000

Бросив взгляд на результаты запроса, можно отметить, что расходы (авансовые выплаты) на книгу *Silicon Valley Gastronomic Treats* составили ровно \$0.00, что, вероятно, связано с малоизвестностью автора. Свой гонорар он будет получать по мере продажи экземпляров этой книги. Расходы на другую книгу *The Psychology of Computer Cooking* выражаются нулевым значением (NULL), возможно, автор и издатель еще не полностью договорились о характере своих отношений либо эти данные просто забыли ввести в базу данных. Со временем это значение будет уточнено и помещено в таблицу. Возможно, оно будет нулем (как в случае с пер-

вой книгой), миллионом или несколькими сотнями долларов. И пока сумма расходов не определена, она будет выражаться значением NULL.

Что произойдет, если операция сравнения будет применена к нулевому значению? Так как нуль (NULL) представляет неизвестное значение, его нельзя сравнить с любым другим значением, даже с другим нулевым значением. Например, в результаты запроса, который ищет все идентификаторы книг с расходами ниже \$5000, не будет включена книга MC3026, *The Psychology of Computer Cooking*.

SQL:

```
select title_id, advance
from titles
where advance < $5000
```

Результат:

title_id	advance
PS7777	4000
PS3333	2000
MC2222	0
TC4203	4000
PS2091	2275

Эта книга не будет найдена и при поиске книг с расходами выше \$5000:

SQL:

```
select title_id, advance
from titles
where advance > $5000
```

Результат:

title_id	advance
PC8888	8000
TC7777	8000
PC1035	7000
BU2075	10125
PS2106	6000
MC3021	15000
TC3218	7000
PS1372	7000

Значение NULL не меньше, не больше и не равно \$5000, потому что является неизвестным. Однако не расстраивайтесь! С помощью специального условия из таблицы всегда можно извлечь строки с нулевыми значениями:

WHERE имя_столбца IS [NOT] NULL

Чтобы найти все книги с нулевыми затратами, можно использовать следующий запрос:

SQL:

```
select title_id, advance
from titles
where advance is null
```

Результат:

<u>title_id</u>	<u>advance</u>
MC3026	NULL
PC9999	NULL

Это условие можно использовать совместно с другими операторами. Например, в результате выполнения следующего запроса находятся все книги, расходы на которые были либо нулевыми (NULL), либо составили меньше \$5000:

SQL:

```
select title_id, advance
from titles
where advance < $5000
or advance is null
```

Результат:

<u>title_id</u>	<u>advance</u>
PS7777	4000
PS3333	2000
MC2222	0
TC4203	4000
MC3026	NULL
PS2091	2275
PC9999	NULL

Поиск по подстрокам: предложение LIKE

Некоторые задачи нельзя решить только с помощью операторов сравнения. Вот несколько примеров.

- “Его фамилия начинается с Mc или Mac, а дальше я не помню”.
- “Мне нужен список телефонов, начинающихся с 415”.
- “Я забыл название книги, но помню, что в ее описании упоминалось слово *exercise*”.
- “Я не помню, как точно пишется его фамилия — Carson или Karsen”.
- “Его имя Dirk или Dick. Четыре буквы, начинается с D и заканчивается на k”.

В каждом из этих случаев известен некоторый образец, где-то встречающийся в столбце, и с его помощью нужно извлечь всю или часть соответствующей ему строки. Для решения этой задачи и предназначено ключевое слово LIKE. Его можно применять к символьным полям, а в некоторых системах и к полям даты. Однако ключевое слово LIKE не работает с числовыми полями (целыми, денежными, десятичными и плавающими). Его синтаксис имеет следующий вид:

```
WHERE имя_столбца [NOT] LIKE 'образец' [ESCAPE ключевой_символ]
```

Образец должен заключаться в кавычки и может содержать один или несколько шаблонов (*wildcards*) — символов, замещающих в образце пропущенные буквы или строки. Ключевое слово ESCAPE используется в том случае, если в образце содержится один из шаблонов, но его нужно рассматривать как простой литерал.

ANSI SQL предусматривает использование двух символов шаблона совместно с ключевым словом LIKE — знак процента (%) и подчеркивание (_).

Шаблон	Значение
%	Любая строка с любым количеством символов
_	Любой одиночный символ

В разных системах могут использоваться разные символы шаблонов, так что обратитесь к документации по своей системе.

Теперь вернемся к сформулированным выше задачам и приведем соответствующие запросы. Сначала осуществим поиск по шотландским фамилиям:

SQL:

```
select au_lname, city
from authors
where au_lname like 'Mc%' or au_lname like 'Mac%'
```

Результат:

au_lname	city
MacFeather	Oakland
McBadden	Vacaville

В соответствии с образцами, приведенными после ключевого слова LIKE, система найдет все фамилии, начинающиеся с Mc и Mac. Обратите внимание на то, что шаблон находится внутри кавычек.

С помощью следующего запроса можно получить список всех телефонов, начинающихся с 415:

SQL:

```
select au_lname, phone
from authors
where phone like '415%'
```

Результат:

au_lname	phone
Bennet	415 658-9932
Green	415 986-7020
Carson	415 548-7723
Stringer	415 843-2991
Straight	415 834-2919
Karsen	415 534-9219
MacFeather	415 354-7128
Dull	415 836-7128
Yokomoto	415 935-4228
Hunter	415 836-7128
Locksley	415 585-4620

Как и в предыдущем случае, нам известно несколько начальных символов, за которыми следует неизвестная строка.

Найти книгу по слову в ее описании несколько сложнее. Неизвестно, где в описании находится слово *exercise* — в начале или в конце, кроме того, неизвестно, с какой буквы оно начинается — строчной или прописной. Все эти возможности можно предусмотреть, используя шаблоны в начале и в конце образца.

SQL:

```
select title_id, notes
from titles
where notes like '%xercise%'
```

Результат:

title_id	notes
PS2106	New exercise, meditation, and nutritional techniques that can reduce the shock of daily interactions. Popular audience. Sample menus included, exercise video available separately.

Если известно точное количество пропущенных символов, можно использовать односимвольный шаблон (`_`). В нашем случае с фамилиями, первая буква либо *K*, либо *C*, а предпоследняя — *e* или *o*.

SQL:

```
select au_lname, city
from authors
where au_lname like '_ars_n'
```

Результат:

au_lname	city
Carson	Berkeley
Karsen	Oakland

Следующий пример аналогичен предыдущему. С его помощью находятся четырехбуквенные имена, начинающиеся с *D* и заканчивающиеся на *k*.

SQL:

```
select au_lname, au_fname, city
from authors
where au_fname like 'D__k'
```

Результат:

au_lname	au_fname	city
Straight	Dick	Oakland
Stringer	Dirk	Oakland

Вместе с шаблонами можно использовать и конструкцию `NOT LIKE`. Чтобы найти в таблице *authors* все номера телефонов, которые не начинаются с 415, можно воспользоваться одним из следующих запросов (они эквивалентны):

SQL:

```
select phone
from authors
where phone not like '415%'
```

```
select phone
from authors
where not phone like '415%'
```

Символы шаблонов обычно используются вместе с ключевым словом `LIKE`. Без него они интерпретируются как обычные символы. В результате выполнения приведенного ниже запроса будут найдены телефоны, состоящие только из четырех символов 415%.

SQL:

```
select phone
from authors
where phone = '415%'
```

А что, если нужно найти значение, которое само содержит один из символов шаблона? Например, в таблице *titles* в столбце *notes* одной из строк используется знак процента. Для поиска по знаку процента нужно использовать ключевое слово ESCAPE, позволяющее трактовать его как обычный символ. Шаблон, следующий непосредственно после ключевого символа, рассматривается в качестве простого литерала. Все последующие символы шаблонов имеют свое обычное значение. С помощью следующего выражения LIKE в столбце *notes* находятся строки, содержащие знак процента. Так как он, скорее всего, не является первым или последним символом строки, в запросе используются шаблоны в начале и в конце образца, а также шаблон в середине, непосредственно после ключевого символа.

SQL:

```
select title_id, notes
from titles
where notes like '%%%' escape '@'
```

Результат:

title_id	notes
TC7777	Detailed instructions on improving your position in life by learning how to make authentic Japanese sushi in your spare time. 5-10% increase in number of friends per recipe reported from beta test.

Вот несколько примеров использования ключевого слова LIKE.

Символ	Значение
LIKE '27%'	27, за которым следует любая строка символов
LIKE '27@%'	27%
LIKE '_n'	an, in, on и так далее
LIKE '@_n'	_n

ЧТО ДАЛЬШЕ

В следующей главе рассматриваются дополнительные вопросы, связанные с операцией выборки данных — упорядочение результатов с помощью ключевого слова ORDER BY, устранение в получаемых результатах повторяющейся информации с помощью ключевого слова DISTINCT и агрегирующие функции.

Сортировка данных и другие методы выбора

ДОПОЛНИТЕЛЬНЫЕ ВОЗМОЖНОСТИ ОПЕРАТОРА SELECT

Теперь, когда вы познакомились с основами оператора SELECT (предложениями SELECT, FROM и WHERE), пришло время узнать о его дополнительных возможностях. В следующих главах рассматриваются предложения ORDER BY (для сортировки результатов запроса), DISTINCT (для устранения повторяющихся строк) и агрегирующие функции (aggregate function), которые используются для вычисления сумм, максимальных и минимальных значений и подсчета строк. В этой главе описывается предложение GROUP BY (для группировки результатов), агрегирующие функции для работы с группами и предложение HAVING (для задания условий на группы).

СОРТИРОВКА РЕЗУЛЬТАТОВ ЗАПРОСА: ПРЕДЛОЖЕНИЕ ORDER BY

Предложение ORDER BY позволяет улучшить представление получаемых результатов. С его помощью можно сортировать результаты по любому столбцу или выражению, указанному в списке выбора. Данные могут сортироваться как по убыванию, так и по возрастанию.

Порядок сортировки

“А” находится перед “В”, не так ли? А что можно сказать о “А” и “а”? Или о “А” и “А”? Ответ зависит от используемого набора символов (character set) и порядка сортировки (sort order).

- Набор символов — это список соответствий между буквами, специальными символами и их внутренними компьютерными кодами. Наборы символов, помимо английских букв, могут включать символы национальных алфавитов и даже их различные комбинации.
- Порядок сортировки определяет порядок следования символов. Например, прописные буквы могут располагаться раньше строчных либо вообще могут считаться эквивалентными.

Наборы символов, внутренние коды и порядок сортировки не определяются в терминах команд SQL, но в большинстве систем управления базами данных можно выбрать нужный набор символов и соответствующий ему порядок сортировки. Однако эти изменения редко можно сделать “на лету”. Обычно набор символов и порядок сортировки выбираются при установке базы данных. SQL-92 предоставляет ряд таких команд для выбора набора символов и порядка сортировки: CREATE CHARACTER SET, DROP CHARACTER SET, CREATE COLLATION, DROP COLLATION, COLLATE и COLLATION FROM, но они редко реализуются в коммерческих системах. Найдите соответствующую информацию в руководстве к своей системе.

В примерах этой книги используется кодовая страница 850 и двоичный порядок сортировки. Список символов этой страницы выглядит следующим образом:

Binary Sort Order for Code Page 850 (cp850). Characters, in Order
!"#\$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNPOQRSTUVWXYZ[\]^_`abc
defghijklmnopqrstuvwxyz{|}~□□□,.,...†‡ˆ%Š<ŠŤŽŽ□"'"—
™™š>šŤžž ~ŁŁAŁŠ"©Š«-@Z°±,ł,µ¶,ąş»L"lZRAAAÄLCÇĆÉĚĚĚİİĐĐŃŃÓÓŌŌ×ŘŮŮ
ŮŮŸŢřřááääááłłććééěěííďďňňóóôóö+řúúúúŸŸ

Как выполняется сортировка

Предположим, что вы хотите получить список цен, идентификационных номеров и номеров издателей из таблицы *titles*. Для этого можно выполнить следующий запрос:

SQL:

```
select price, title_id, pub_id  
from titles
```

Результат:

price	title_id	pub_id
20	PC8888	1389
19.99	BU1032	1389
7.99	PS7777	0736
19.99	PS3333	0736
11.95	BU1111	1389
19.99	MC2222	0877
14.99	TC7777	0877
11.95	TC4203	0877
22.95	PC1035	1389
NULL	MC3026	0877
2.99	BU2075	0736
10.95	PS2091	0736
7	PS2106	0736
2.99	MC3021	0877
20.95	TC3218	0877
19.99	BU7832	1389
21.59	PS1372	0736
NULL	PC9999	1389

В этом списке представлена вся запрошенная вами информация, но разбираться в ней не удобно, так как результат неупорядочен. Чтобы сделать эту информацию более полезной, ее надо отсортировать по цене. Это делается следующим образом:

SQL:

```
select price, title_id, pub_id  
from titles  
order by price
```

Результат:

price	title_id	pub_id
NULL	MC3026	0877
NULL	PC9999	1389

2.99	MC3021	0877
2.99	BU2075	0736
7	PS2106	0736
7.99	PS7777	0736
10.95	PS2091	0736
11.95	BU1111	1389
11.95	TC4203	0877
14.99	TC7777	0877
19.99	MC2222	0877
19.99	BU7832	1389
19.99	PS3333	0736
19.99	BU1032	1389
20	PC8888	1389
20.95	TC3218	0877
21.59	PS1372	0736
22.95	PC1035	1389

Теперь строки будут отсортированы по цене.

Синтаксис предложения ORDER BY

Общая форма предложения ORDER BY в операторе SELECT имеет следующий вид:

```
SELECT список_выбора
FROM список_таблиц
[WHERE условия]
[ORDER BY {выражение [ASC | DESC] | позиция [ASC | DESC]}
 [ , {выражение [ASC | DESC] | позиция [ASC | DESC] } ]...]
```

В большинстве (но не во всех) системах требуется, чтобы каждый элемент, по которому выполняется сортировка (столбец или выражение), присутствовали в списке выбора. Для выражений имеется три возможности: можно использовать целое число, описывающее позицию выражения в списке выбора, использовать заголовок столбца, определенный в списке выбора (`price * ytd_sales as income`), или использовать выражение целиком. Детали этого для пяти различных систем приведены в Приложении Б. За подробностями также обращайтесь к руководству по своей системе.

Сортировка внутри сортировки

Итак, мы получили результат, отсортированный по ценам. Однако было бы неплохо, чтобы книги в каждой ценовой категории, выпущенные одним издателем, шли в списке рядом. Для этого в список ORDER BY нужно добавить столбец `pub_id`:

```
SQL:
select price, title_id, pub_id
from titles
order by price, pub_id
```

Результат:

price	title_id	pub_id
NULL	MC3026	0877
NULL	PC9999	1389
2.99	BU2075	0736

2.99	MC3021	0877
7	PS2106	0736
7.99	PS7777	0736
10.95	PS2091	0736
11.95	TC4203	0877
11.95	BU1111	1389
14.99	TC7777	0877
19.99	PS3333	0736
19.99	MC2222	0877
19.99	BU7832	1389
19.99	BU1032	1389
20	PC8888	1389
20.95	TC3218	0877
21.59	PS1372	0736
22.95	PC1035	1389

При использовании более одного столбца в предложении ORDER BY выполняется так называемая вложенная (nested) сортировка, т.е. сначала выполняется сортировка по цене, а затем по значениям *pub_id* в каждой ценовой категории.

Количество уровней сортировки может быть любым. Во многих системах требуется, чтобы элемент, по которому ведется сортировка, присутствовал в списке выбора, однако порядок их перечисления в предложении ORDER BY не обязан совпадать с порядком перечисления столбцов и выражений в операторе SELECT. Если в предыдущем примере поменять порядок сортировки так, чтобы результаты сначала упорядочивались по столбцу *pub_id*, а затем по столбцу *price*, результат изменится следующим образом:

SQL:

```
select price, title_id, pub_id
from titles
order by pub_id, price
```

Результат:

price	title_id	pub_id
2.99	BU2075	0736
7	PS2106	0736
7.99	PS7777	0736
10.95	PS2091	0736
19.99	PS3333	0736
21.59	PS1372	0736
NULL	MC3026	0877
2.99	MC3021	0877
11.95	TC4203	0877
14.99	TC7777	0877
19.99	MC2222	0877
20.95	TC3218	0877
NULL	PC9999	1389
11.95	BU1111	1389
19.99	BU7832	1389

19.99	BU1032	1389
20	PC8888	1389
22.95	PC1035	1389

Порядок столбцов (*price*, *title_id*, *pub_id*) по сравнению с предыдущим примером не изменился, однако строки расположились по-другому: сначала идут строки с идентификационным номером издателя — 0736, затем — с номером 0877 и наконец — с номером 1389.

Сортировка по возрастанию и по убыванию

С помощью ключевых слов ASC (по возрастанию) и DESC (по убыванию) можно изменить порядок сортировки в каждом отдельном случае. По умолчанию данные сортируются в порядке возрастания. Для изменения порядка сортировки нужно использовать ключевое слово DESC.

С помощью следующего запроса можно вывести цены в убывающем порядке:

SQL:

```
select price, title_id, pub_id
from titles
order by price desc, pub_id
```

Результат:

price	title_id	pub_id
22.95	PC1035	1389
21.59	PS1372	0736
20.95	TC3218	0877
20	PC8888	1389
19.99	PS3333	0736
19.99	MC2222	0877
19.99	BU7832	1389
19.99	BU1032	1389
14.99	TC7777	0877
11.95	TC4203	0877
11.95	BU1111	1389
10.95	PS2091	0736
7.99	PS7777	0736
7	PS2106	0736
2.99	BU2075	0736
2.99	MC3021	0877
NULL	MC3026	0877
NULL	PC9999	1389

Обратите внимание, что в рамках каждой ценовой категории идентификаторы издателей по-прежнему сортируются в порядке возрастания. В зависимости от вашей системы, нулевые (NULL) значения могут располагаться либо в начале, либо в конце списка. Чтобы изменить порядок сортировки в столбце *pub_id*, нужно выполнить следующий запрос:

SQL:

```
select price, title_id, pub_id
from titles
order by price desc, pub_id desc
```

Результат:

price	title_id	pub_id
22.95	PC1035	1389
21.59	PS1372	0736
20.95	TC3218	0877
20	PC8888	1389
19.99	BU7832	1389
19.99	BU1032	1389
19.99	MC2222	0877
19.99	PS3333	0736
14.99	TC7777	0877
11.95	BU1111	1389
11.95	TC4203	0877
10.95	PS2091	0736
7.99	PS7777	0736
7	PS2106	0736
2.99	MC3021	0877
2.99	BU2075	0736
NULL	PC9999	1389
NULL	MC3026	0877

А как насчет выражений?

Что делать, если нужно отсортировать результат по значению выражения из списка выбора? SQL позволяет использовать для этих целей позицию выражения в списке выбора (представленную целым числом) или заголовки (также называемые псевдонимами или метками). Некоторые системы позволяют использовать само выражение. Ниже приведен пример, в котором могут возникнуть проблемы при сортировке по выражению `price * ytd_sales` из списка выбора:

SQL:

```
select pub_id, price * ytd_sales, price, title_id
from titles
```

Результат:

pub_id	price	title_id
1389	81900	20
1389	81859.05	19.99
0736	26654.64	7.99
0736	81399.28	19.99
1389	46318.2	11.95
0877	40619.68	19.99
0877	61384.05	14.99
0877	180397.2	11.95
1389	201501	22.95
0877	NULL	NULL
0736	55978.78	2.99
0736	22392.75	10.95

0736	777	7	PS2106
0877	66515.54	2.99	MC3021
0877	7856.25	20.95	TC3218
1389	81859.05	19.99	BU7832
0736	8096.25	21.59	PS1372
1389	NULL	NULL	PC9999

Сортировка по позиции. Предположим, что сначала результаты надо отсортировать по издателям, а затем по значению `price * ytd_sales`. Поскольку это значение является выражением, для него нельзя использовать обычное имя столбца. Вместо него нужно использовать число 2, так как выражение является вторым элементом списка выбора. (Нумерация начинается с 1 и ведется слева направо. Числа со знаками (например, -2, +4 и т.д.) не допускаются и в любом случае не имеют никакого смысла.)

```
SQL:
select pub_id, price * ytd_sales, price, title_id
from titles
order by pub_id, 2
```

Результат:

pub_id		price	title_id
0736	777	7	PS2106
0736	8096.25	21.59	PS1372
0736	22392.75	10.95	PS2091
0736	26654.64	7.99	PS7777
0736	55978.78	2.99	BU2075
0736	81399.28	19.99	PS3333
0877	NULL	NULL	MC3026
0877	7856.25	20.95	TC3218
0877	40619.68	19.99	MC2222
0877	61384.05	14.99	TC7777
0877	66515.54	2.99	MC3021
0877	180397.2	11.95	TC4203
1389	NULL	NULL	PC9999
1389	46318.2	11.95	BU1111
1389	81859.05	19.99	BU7832
1389	81859.05	19.99	BU1032
1389	81900	20	PC8888
1389	201501	22.95	PC1035

Цифры можно использовать как для выражений, так и для обычных столбцов. Ключевые слова `ASC` и `DESC` аналогично применяются и к цифрам, и к заголовкам столбцов. В следующем примере данные сортируются в порядке возрастания по столбцу `pub_id`, а затем в порядке убывания по столбцу `price`:

```
SQL:
select pub_id, price * ytd_sales, price, title_id
from titles
order by pub_id, 3 desc
```

Результат:

pub_id	price	title_id
0736	8096.25	PS1372
0736	81399.28	PS3333
0736	22392.75	PS2091
0736	26654.64	PS7777
0736	777	PS2106
0736	55978.78	BU2075
0877	7856.25	TC3218
0877	40619.68	MC2222
0877	61384.05	TC7777
0877	180397.2	TC4203
0877	66515.54	MC3021
0877	NULL	MC3026
1389	201501	PC1035
1389	81900	PC8888
1389	81859.05	BU7832
1389	81859.05	BU1032
1389	46318.2	BU1111
1389	NULL	PC9999

При использовании номеров в предложении ORDER BY нужно внимательно следить за изменениями в списке выбора. При добавлении или удалении столбцов из списка выбора результаты таких запросов могут измениться до неузнаваемости.

Сортировка по заголовку выражения. Если в списке выбора выражение определено с заголовком, по нему можно выполнять сортировку. Фактически, благодаря этой возможности можно вообще отказаться от сортировки по позиции. В следующем примере, который как и предыдущий, приводит к аналогичным результатам, выражению `price * ytd_sales` назначен заголовок `income`. Именно этот заголовок и используется в предложении ORDER BY вместо номера позиции в списке выбора.

```
SQL:
select pub_id, price * ytd_sales as income, price, title_id
from titles
order by pub_id, income desc
```

Сортировка по выражению. В некоторых системах допускается выполнение сортировки по самому выражению из списка выбора, без использования его номера позиции или заголовка. Проверьте, допускается ли в вашей системе следующий запрос:

```
SQL:
select pub_id, price * ytd_sales, price, title_id
from titles
order by pub_id, price * ytd_sales desc
```

Как сортировать нулевые значения

Во все системы одинаково упорядочивают нулевые значения. В стандарте SQL-92 определено, что при сортировке нулевых значений последние должны быть либо больше, либо меньше всех ненулевых значений. Как будет выполняться сортировка у вас — зависит от конкретной реализации. В Sybase SQL Server нули (NULL) считаются меньшими всех ненулевых значений, а в Sybase SQL Anywhere нулевые значения всегда выводятся в начале списка, независимо от порядка сортировки (по возрастанию или по убыванию).

УСТРАНЕНИЕ ПОВТОРЯЮЩИХСЯ СТРОК: ПРЕДЛОЖЕНИЯ DISTINCT И ALL

С помощью ключевых слов DISTINCT и ALL в списке выбора можно определить, что делать с повторяющимися строками результата. ALL возвращает все строки, удовлетворяющие условиям запроса (этот режим используется по умолчанию). DISTINCT возвращает только неповторяющиеся строки.

Например, при поиске в таблице *titleauthors* всех идентификаторов авторов с использованием ключевого слова ALL получается следующий результат:

SQL:

```
select all au_id
from titleauthors
```

Результат:

```
au_id
-----
409-56-7008
486-29-1786
486-29-1786
712-45-1867
172-32-1176
213-46-8915
238-95-7766
213-46-8915
998-72-3567
899-46-2035
998-72-3567
722-51-5454
899-46-2035
807-91-6654
274-80-9391
427-17-2319
846-92-7186
756-30-7391
724-80-9391
724-80-9391
267-41-2394
672-71-3249
267-41-2394
472-27-2349
648-92-1872
```

Внимательно посмотрев на результаты запроса, можно обнаружить в нем повторяющиеся строки. С помощью ключевого слова DISTINCT их можно устранить.

SQL:

```
select distinct au_id
from titleauthors
```

Результат:

```
au_id
-----
172-32-1176
213-46-8915
```

238-95-7766
267-41-2394
274-80-9391
409-56-7008
427-17-2319
472-27-2349
486-29-1786
648-92-1872
672-71-3249
712-45-1867
722-51-5454
724-80-9391
756-30-7391
807-91-6654
846-92-7186
899-46-2035
998-72-3567

Таким образом из результатов предыдущего запроса будут удалены шесть повторяющихся строк.

Синтаксис предложения DISTINCT

Основная форма предложения DISTINCT имеет следующий вид:

```
SELECT [DISTINCT | ALL] список_выбора
```

Ключевые слова DISTINCT и ALL можно использовать в запросе только один раз и они должны стоять в начале списка выбора. В следующем примере умышленно совершена синтаксическая ошибка:

SQL (*неверно*):

```
select state, distinct city  
from authors
```

Другими словами, нельзя выбрать все штаты и при этом только неповторяющиеся города.

Почувствуйте разницу!

Если в списке выбора находится несколько элементов, при использовании ключевого слова DISTINCT выбираются только строки с уникальными комбинациями значений этих элементов.

Рассмотрим следующий пример. Для начала получим список всех значений столбцов *pub_id* и *types* из таблицы *titles*.

SQL:

```
select pub_id, type  
from titles  
order by pub_id
```

Результат:

pub_id	type
0736	psychology
0736	psychology

```

0736      psychology
0736      psychology
0736      psychology
0736      business
0877      NULL
0877      mod_cook
0877      trad_cook
0877      trad_cook
0877      mod_cook
0877      trad_cook
1389      popular_comp
1389      popular_comp
1389      business
1389      business
1389      business
1389      popular_comp

```

Результат состоит из восемнадцати строк, включая повторяющиеся. (Предложение ORDER BY используется исключительно для упрощения вида результата.) Если вы попытаете извлечь различающиеся номера издателей, то получите только три строки:

SQL:

```

select distinct pub_id
from titles
order by pub_id

```

Результат:

```

pub_id
-----
0736
0877
1389

```

При выборе только различающихся типов вы получите шесть следующих строк:

SQL:

```

select distinct type
from titles
order by type

```

Результат:

```

type
-----
NULL
business
mod_cook
popular_comp
psychology
trad_cook

```

А если вы захотите выбрать различающиеся комбинации значений “издатель-тип”, получите семь строк:

SQL:

```
select distinct pub_id, type
from titles
order by pub_id
```

Результат:

pub_id	type
0736	business
0736	psychology
0877	NULL
0877	mod_cook
0877	trad_cook
1389	business
1389	popular_comp

Теперь в этом списке представлены только уникальные комбинации значений “издатель-тип”. Издатель под номером 0736 выпустил книги двух типов, издатель под номером 0877 — двух типов (кроме того, информация о третьем типе неизвестна), издатель под номером 1389 выпустил также книги двух типов, что в сумме дает семь строк. Таким образом, ключевое слово DISTINCT применяется ко всему списку выбора, а не к отдельным столбцам.

А различаются ли нулевые значения? Хотя по определению нулевые значения никогда не равны друг другу, при использовании ключевого слова DISTINCT все нулевые значения в столбце считаются повторяющимися. Например, если бы издатель с номером 0877 имел несколько книг с неопределенным типом (NULL), то при выполнении запроса с ключевым словом DISTINCT в результате все равно была бы выбрана только одна комбинация “издатель-NULL”.

DISTINCT *. Если в вашей системе допускается конструкция DISTINCT *, сравните результаты выполнения двух следующих запросов:

SQL:

```
select distinct *
from titles
```

SQL:

```
select *
from titles
```

Скорее всего, вы получите одинаковые результаты, так как все строки в таблице должны быть уникальны. В противном случае нужно обратить серьезное внимание на структуру базы данных. Почему в одной таблице оказались строки с одинаковыми наборами значений? Как извлечь конкретную строку, если в таблице имеется несколько ее “двойников”?

DISTINCT и ORDER BY. В большинстве диалектов SQL каждый элемент предложения ORDER BY должен также находиться и в списке выбора. В системах, предоставляющих большую гибкость (когда элемент предложения ORDER BY может не включаться в список выбора), можно выполнять запросы с ключевым словом DISTINCT, примененным к списку выбора, и с предложением ORDER BY, содержащем элементы, не входящие в список выбора.

Например, в Sybase SQL Server при сортировке по столбцу, не входящему в список выбора, получается тот же результат, что и при включении этого столбца в список выбора с ключевым словом DISTINCT. В обоих случаях это приводит к увеличению количества выбранных строк. Обратите внимание на результаты следующего запроса (не пытайтесь выполнить его в SQL Anywhere — вы получите сообщение об ошибке):

SQL:

```
select distinct pub_id
from titles
order by type
```

Так как в базе представлены только три издателя, в результате можно было бы ожидать получение трех строк. Однако вот что получается на самом деле:

Результат:

```
pub_id
-----
0877
0736
1389
0877
1389
0736
0877
```

Почему в результате оказалось семь строк? Ответ заключен в предложении ORDER BY. Это же количество строк было бы получено и при сортировке по издателям и поиске различных типов.

SQL:

```
select distinct type
from titles
order by pub_id
```

Результат:

```
type
-----
business
psychology
NULL
mod_cook
trad_cook
business
popular_comp
```

Теперь становится понятно, откуда взялись семь строк. Transact-SQL находит все возможные уникальные комбинации значений элементов из списка выбора (с ключевым словом DISTINCT) и предложения ORDER BY (не входящие в список выбора). Таких комбинаций *pub_id* и *type* ровно семь.

Если в вашей системе также допускается использование в предложении ORDER BY элементов, не входящих в список выбора, поэкспериментируйте с приведенным выше запросом, чтобы узнать, как обрабатывается в ней подобная ситуация. Целый ряд примеров использования ключевого слова DISTINCT содержится в главе 12.

АГРЕГИРУЮЩИЕ ФУНКЦИИ

Агрегирующие функции используются для получения обобщающих значений. Их можно применять к наборам (set) строк: ко всем строкам таблицы, строкам, определенным в предложении WHERE, или к группам строк в предложении GROUP BY (это предложение описывается в следующей главе). В любом случае, независимо от структуры набора строк, для каждого из них получается единственное значение.

Обратите внимание на различие в результатах выполнения следующих запросов: первый находит для каждой строки таблицы *titles* соответствующее значение годового объема продаж, второй вычисляет общую сумму, полученную за год от продажи всех книг (одно значение для набора, состоящего из всех строк таблицы).

SQL:

```
select ytd_sales
from titles
```

Результат:

```
ytd_sales
-----
4095
4095
3336
4072
3876
2032
4095
15096
8780
NULL
18722
2045
111
22246
375
4095
375
NULL
```

SQL:

```
select sum(ytd_sales)
from titles
```

Результат:

```
-----
97446
```

В первом случае результат возвращается для каждой отдельной строки таблицы, во втором получается одно значение для всех строк. Получаемые таким образом столбцы результата выводятся без заголовков. В ряде случаев заголовков может генерироваться самой системой (как это, например, делается в SQL Anywhere). Чтобы сделать результат более понятным, можно использовать следующий запрос:

SQL:

```
select sum(ytd_sales) as Total
from titles
```

Результат:

```
Total
-----
97446
```

В большинстве диалектов SQL (включая SQL Anywhere) запрещается смешивать в одном предложении обычные значения и значения агрегирующих функций. Таким образом, список выбора может быть либо пустым, либо состоять из имен столбцов и выражений (обрабатывающих значения строк), либо включать только агрегирующие функции (применяющиеся к наборам значений). Единственным исключением является предложение GROUP BY, которое рассматривается в главе 6.

Ниже приведен пример обычно недопустимого запроса:

SQL (вариант):

```
select price, sum(price)
from titles
```

В этом случае проблема заключается в том, что *price* возвращает значение для каждой строки, тогда как *sum(price)* возвращает значение для целого набора строк (в данном случае для всех строк таблицы). Если в системе не предусмотрена обработка подобной ситуации, вы получите сообщение о синтаксической ошибке.

Если же подобный запрос будет обработан, то, скорее всего, вы получите следующий результат (как, например, в Sybase SQL Server):

Результат:

price	
19.99	236.26
11.95	236.26
2.99	236.26
19.99	236.26
19.99	236.26
2.99	236.26
NULL	236.26
22.95	236.26
20.00	236.26
NULL	236.26
21.59	236.26
10.95	236.26
7.00	236.26
19.99	236.26
7.99	236.26
20.95	236.26
11.95	236.26
14.99	236.26

В столбце без названия содержится значение общей стоимости всех книг, тогда как в левом столбце представлена цена каждой книги. Обратите внимание, что все значения в правом столбце одинаковы, так как в каждой строке вычислялась одна и та же сумма.

Синтаксис агрегирующих функций

Агрегирующие функции всегда имеют аргументы (*argument*). Аргументы являются выражениями и заключаются в скобки. Общая форма агрегирующей функции имеет следующий вид:

```
aggregate_function ([DISTINCT] выражение)
```

Список агрегирующих функций приведен на рис. 5.1.

Со всеми агрегирующими функциями, кроме COUNT(*), можно использовать ключевое слово DISTINCT. Тем не менее не имеет никакого смысла использо-

вать его с функциями MIN и MAX, так как получаемые значения в обоих случаях будут одинаковы.

В качестве аргументов агрегирующих функций обычно используются названия столбцов, но также допускаются константы, функции и любые их комбинации с арифметическими операторами.

Агрегирующая функция	Результат
SUM([DISTINCT] выражение)	Сумма (различных) значений
AVG([DISTINCT] выражение)	Средняя величина (различных) значений
COUNT([DISTINCT] выражение)	Количество (различных) ненулевых значений
COUNT(*)	Количество выбранных строк
MAX(выражение)	Максимальное значение
MIN(выражение)	Минимальное значение

Рис. 5.1. Агрегирующие функции

Например, с помощью следующего оператора можно найти среднюю стоимость всех книг в случае удвоения их цены:

SQL:

```
select avg(price * 2)
from titles
```

Результат:

```
-----
29.5325
```

Функции COUNT и COUNT(*). Несмотря на внешнюю схожесть, функции COUNT и COUNT(*) используются для разных целей. COUNT в качестве аргумента использует столбец или выражение и подсчитывает общее количество его ненулевых значений, тогда как COUNT(*) находит общее количество строк, независимо от наличия в них нулевых значений. Сравните эти функции на следующем примере:

SQL:

```
select count(price), count(*)
from titles
```

Результат:

```
-----
16                18
```

Получаемые с помощью этих функций результаты отличаются, так как две строки в таблице *titles* содержат нулевые значения в столбце *price*. При использовании столбца без нулевых значений результаты работы функций COUNT и COUNT(*) будут одинаковы.

SQL:

```
select count(title_id), count(*)
from titles
```

Результат:

```
-----
18                18
```

Совместное использование этих функций может быть полезно при подсчете нулевых значений в заданных столбцах.

Агрегирующие функции и типы данных. Функции SUM и AVG можно применять только к числовым столбцам. Функции MIN, MAX, COUNT и COUNT(*) работают со всеми типами данных.

Например, функцию MIN можно использовать для поиска наименьшего значения (в символьном столбце — самого первого значения в алфавитном порядке):

SQL:

```
select min(au_lname)
from authors
```

Результат:

```
-----
Bennet
```

Конечно, суммировать или находить среднее значение по столбцу с фамилиями авторов абсолютно бессмысленно.

DISTINCT и агрегирующие функции. С функциями SUM, AVG, COUNT, MIN и MAX можно использовать ключевое слово DISTINCT (оно помещается перед аргументом). Как уже отмечалось раньше, DISTINCT не влияет на результаты выполнения функций MAX и MIN.

При использовании ключевого слова DISTINCT при вычислении суммы, среднего значения или общего количества значений не учитываются повторяющиеся строки. Ниже приведен соответствующий пример:

SQL:

```
select count(price)
from titles
```

Результат:

```
-----
16
```

SQL:

```
select count(distinct price)
from titles
```

Результат:

```
-----
11
```

В первом запросе находится количество ненулевых значений цены в таблице *titles*. Во втором подсчитывается количество различных ненулевых значений цены. При сравнении полученных результатов видно, что пять книг имеют одинаковую стоимость.

В некоторых системах при использовании ключевого слова DISTINCT аргумент не может быть арифметическим выражением, а должен являться исключительно именем столбца. Другие системы более демократичны. Соответствующую информацию всегда можно найти в руководстве по конкретной системе. Также вы можете просто проверить, работают ли в вашей системе следующие запросы:

SQL:

```
select count(price * 2)
from titles
```

Результат:

16

SQL (вариант):

```
select count (distinct price * 2)
from titles
```

Результат:

11

В первом запросе удваиваются все цены и подсчитывается их общая сумма. Во втором выполняется та же операция, но учитываются только неповторяющиеся значения. Если ваша система не позволяет использовать выражение вместе с ключевым словом `DISTINCT`, то при выполнении второго запроса вы получите сообщение об ошибке.

`DISTINCT` также не применяется вместе с функцией `COUNT(*)`, потому что она всегда возвращает единственную строку. В этом случае использование `DISTINCT` абсолютно бессмысленно.

Как правило, ключевое слово `DISTINCT` в списке выбора можно использовать только один раз. При отсутствии агрегирующих функций `DISTINCT` применяется ко всему списку выбора, а не к отдельным столбцам. Однако это ограничение может привести к возникновению некоторых проблем при использовании агрегирующих функций. Для примера рассмотрим результат выполнения двух операций. При подсчете суммы и количества значений в столбце *price* получаются следующие результаты:

SQL:

```
select count(price), sum(price)
from titles
```

Результат:

16 236.26

Таким образом, средняя цена одной книги составляет \$236.26 делить на 16.

При использовании ключевого слова `DISTINCT` с одним из столбцов результат, скорее всего, будет менее полезен:

SQL:

```
select count(price), sum(distinct price)
from titles
```

Результат:

16 161.35

Различие результатов связано с существованием в столбце одинаковых значений цены, которые не учитываются при подсчете суммы. Тем не менее общее количество значений осталось неизменным. Поэтому на основе этих данных уже нельзя правильно вычислить среднюю стоимость книги. Точно так же неправильная средняя стоимость будет получена и при использовании ключевого слова `DISTINCT` в функции `COUNT`.

SQL:

```
select count(distinct price), sum(price)
from titles
```

Результат:

```
-----  
11          236.26
```

Очевидно, что в этом случае нужно либо вообще не использовать ключевое слово `DISTINCT`, либо использовать его в обеих функциях. Поэтому стандарт SQL-92 поддерживает многократное использование ключевого слова `DISTINCT` в агрегирующих функциях. Ниже приведен соответствующий пример:

SQL:

```
select count(distinct price), sum(distinct price)  
from titles
```

Результат:

```
-----  
11          161.35
```

Обратите внимание на то, что использование ключевого слова `DISTINCT` в списке выбора и в агрегирующей функции приводит к разным результатам:

SQL:

```
select count(au_id)  
from titleauthors
```

Результат:

```
-----  
25
```

SQL:

```
select count(distinct au_id)  
from titleauthors
```

Результат:

```
-----  
19
```

SQL:

```
select distinct count(au_id)  
from titleauthors
```

Результат:

```
-----  
25
```

С помощью первого запроса находится общее количество идентификационных номеров авторов. С помощью второго подсчитываются только различные номера. Применение ключевого слова `DISTINCT` ко всему списку выбора дает тот же результат, что и в первом запросе. Это связано с тем, что сначала выполняется функция, которая возвращает в результате одну строку, а затем к этой строке применяется ключевое слово `DISTINCT`. А так как эта строка единственная, по определению не может существовать никаких повторяющихся значений.

Агрегирующие функции и предложение `WHERE`. Агрегирующие функции можно использовать в списке выбора, как в предыдущих примерах, либо в предложении `HAVING` оператора `SELECT` (подробности смотрите в главе 6).

Агрегирующие функции нельзя использовать в предложении `WHERE`. В противном случае вы получите сообщение об ошибке. Однако предложение `WHERE`

можно использовать для ограничения числа строк, участвующих в агрегирующих вычислениях. С помощью следующего оператора вычисляется средний расход по книгам и годовой доход от продаж по всем строкам таблицы *titles*.

SQL:

```
select avg(advance), sum(ytd_sales)
from titles
```

Результат:

```
-----
5962.5      97446
```

Для получения аналогичных результатов только для книг по бизнесу можно выполнить следующий запрос:

SQL:

```
select avg(advance), sum(ytd_sales)
from titles
where type = 'business'
```

Результат:

```
-----
6281.25     30788
```

Как взаимодействуют предложение WHERE и агрегирующие функции? Сначала выполняется предложение WHERE, в результате чего находятся все книги по бизнесу. Затем к выбранным строкам применяются агрегирующие функции.

Нулевые значения и агрегирующие функции. Если в столбце, к которому применяется агрегирующая функция, имеются нулевые (NULL) значения, они просто игнорируются.

Например, при подсчете количества значений в столбцах *advance* и *title* таблицы *titles* с использованием функции COUNT получаются разные результаты, так как в столбце *advance* содержится два нулевых значения:

SQL:

```
select count(advance)
from titles
```

Результат:

```
-----
16
```

SQL:

```
select count(title)
from titles
```

Результат:

```
-----
18
```

Исключением является функция COUNT(*), которая всегда подсчитывает общее количество строк, независимо от наличия в них нулевых значений. Если условиям запроса не удовлетворяет ни одна строка, функция COUNT возвращает нуль. Все другие функции в этом случае возвращают значение NULL. Ниже приведен соответствующий пример:

SQL:

```
select count(distinct title)
```

```
from titles
where type = 'poetry'
```

Результат:

0

SQL:

```
select avg(advance)
from titles
where type = 'poetry'
```

Результат:

NULL

СКАЛЯРНЫЕ И ВЕКТОРНЫЕ ФУНКЦИИ

В следующей главе рассматриваются предложение GROUP BY и использование агрегирующих функций с группами для получения массива значений (по одному на группу).

Кроме того, описываются предложение HAVING и векторные агрегирующие функции.

Группировка данных и построение отчетов

ГРУППИРОВКА

В предыдущей главе рассматривались особенности оператора SELECT: использование ключевого слова DISTINCT и предложения ORDER BY. В ней также было введено понятие агрегирующих функций, которые применялись ко всей таблице в целом. Однако это всего лишь частный случай их использования. На практике агрегирующие функции чаще используются в комбинации с предложением GROUP BY.

В этой главе основное внимание уделяется предложению GROUP BY, которое возвращает в результате группы строк, и предложению HAVING, с помощью которого можно задать условия отбора в предложении GROUP BY. Кроме того, в этой главе подводятся итоги использования в базах данных нулевых значений.

ПРЕДЛОЖЕНИЕ GROUP BY

Предложение GROUP BY неразрывно связано с агрегирующими функциями, без них оно практически не используется. Предложение GROUP BY разделяет таблицу на наборы, а агрегирующая функция вычисляет для каждого из них итоговое значение. Эти значения называются агрегирующим вектором. (В предыдущей главе рассматривались скалярные агрегирующие функции, которые в результате возвращали единственное значение.) На рис. 6.1 представлены примеры запросов, в которых использованы скалярные и векторные агрегирующие функции.

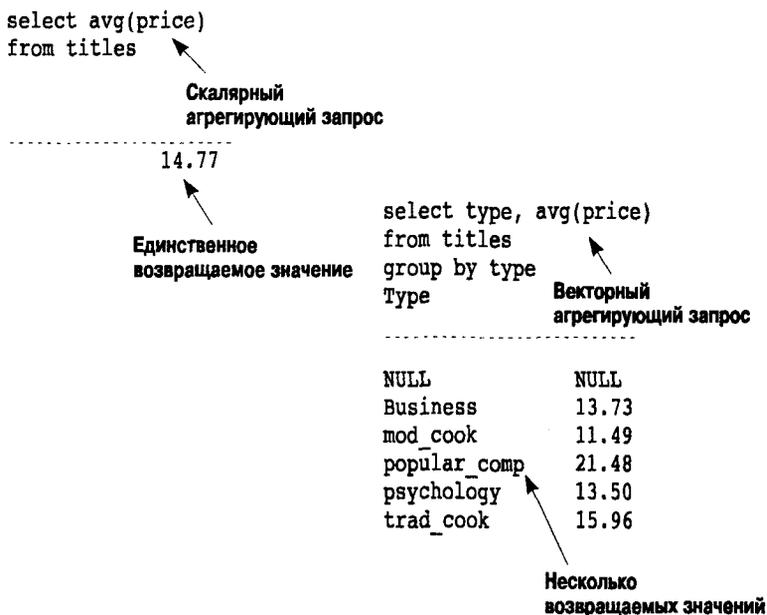


Рис. 6.1. Скалярные и векторные агрегирующие функции

Синтаксис предложения GROUP BY

В контексте оператора SELECT предложение GROUP BY имеет следующий вид:

```
SELECT список_выбора
FROM список_таблиц
[WHERE условия]
[GROUP BY список_группировки]
[ORDER by список_порядка]
```

В большинстве диалектов SQL каждый элемент из списка GROUP BY должен обязательно присутствовать в списке выбора — другими словами, группировать можно только выбираемые элементы. Разные системы, кроме имен столбцов, позволяют использовать в списке группировки выражения, заголовки столбцов и их порядковые номера в списке выбора. Некоторые возможные варианты приведены в Приложении Б. Кроме того, уточните соответствующие детали в руководстве по своей конкретной системе. Ниже приводится пример, в котором группировка выполняется по одному столбцу:

SQL:

```
select pub_id, count(type)
from titles
group by pub_id
```

Результат:

pub_id	
0736	6
0877	5
1389	6

В список выбора включается столбец, по которому выполняется группировка и агрегирующая функция. Все строки из первой группы имеют значение 0736 в столбце *pub_id*, из второй группы — 0877, из третьей — 1389. Функция COUNT вычисляет для каждой группы соответствующее ей единственное значение.

Группировка внутри групп. Путем сортировки одновременно по нескольким элементам можно создавать группы внутри групп. Разделяя элементы, по которым будет проводиться группировка, с помощью запятых можно разбить большие группы на подгруппы.

SQL:

```
select pub_id, type, count(type)
from titles
group by pub_id, type
```

Результат:

pub_id	type	
0736	business	1
0736	psychology	5
0877	NULL	0
0877	mod_cook	2
0877	trad_cook	3
1389	business	3
1389	popular_comp	3

Этот пример во многом аналогичен предыдущему, но в нем используются вложенные группы. Сначала строки таблицы разделяются по издателям, а затем

каждая полученная группа разделяется по типу. В результате получается семь групп, или наборов. После этого к каждому такому набору применяется агрегирующая функция, которая вычисляет для каждого издателя количество книг по каждой теме.

Ограничения. Несмотря на простой синтаксис, предложение GROUP BY часто является источником головной боли пользователей SQL.

Ниже приводится кажущийся резонным запрос, который однако не будет работать на большинстве систем:

```
SQL (вариант):
select pub_id, type, count(type)
from titles
group by pub_id
```

Так как таблица разделяется на наборы по издателям (group by pub_id), результат запроса должен состоять не более чем из трех строк, и каждый элемент из списка выбора должен иметь по одному значению для набора. Однако у издателей имеются книги нескольких типов, и поэтому в большинстве систем этот запрос не пройдет. Решить эту проблему можно, добавив в предложение GROUP BY столбец type, как в предыдущем примере.

Другое ограничение в некоторых реализациях SQL касается использования выражений. В предложении GROUP BY всегда можно использовать имена столбцов, а поддержка выражений, заголовков столбцов и их порядковых номеров в списке выбора зависит от конкретной системы.

А как быть с множественными итоговыми значениями при наличии нескольких уровней группировки? Предположим, что выполняется сортировка по столбцам pub_id и type. Требуется найти общее количество книг для каждого издателя и количество книг по каждой теме. На первый взгляд с этой задачей должен справиться следующий запрос:

```
SQL:
select pub_id, count(title_id), type, count(title_id)
from titles
group by pub_id, type
```

Однако результат свидетельствует об обратном:

Результат:

pub_id		type	
0736	1	business	1
0736	5	psychology	5
0877	1	NULL	1
0877	2	mod_cook	2
0877	3	trad_cook	3
1389	3	business	3
1389	3	popular_comp	3

Количество книг по издателям и по типам совпадают, что явно не соответствует тому, что требовалось получить. В числовых столбцах содержится количество книг для каждой комбинации издатель/тип, так как это нижний уровень группы. Для получения требуемых результатов нужно выполнить два отдельных запроса: в первом задать группировку по издателям, во втором — по издателям, а затем по типам.

```
SQL:
select pub_id, count(title_id)
from titles
group by pub_id
```

Из результатов первого запроса становится ясно, что каждый издатель выпустил по шесть книг.

Результат:

pub_id	
0736	6
0877	6
1389	6

Во втором запросе находится общее количество книг для каждой комбинации издатель/тип.

SQL:

```
select pub_id, type, count(title_id)
from titles
group by pub_id, type
```

Результат:

pub_id	type	
0736	business	1
0736	psychology	5
0877	NULL	1
0877	mod_cook	2
0877	trad_cook	3
1389	business	3
1389	popular_comp	3

В случае группировки только по типу результаты будут другими:

SQL:

```
select type, count(title_id)
from titles
group by type
```

Результат:

type	
NULL	1
business	4
mod_cook	2
popular_comp	3
psychology	5
trad_cook	3

В последнем случае в бизнес-категорию попадают четыре книги. Однако эти книги были выпущены двумя издателями, три книги — одним издателем и одна — другим. Таким образом, из результатов этого запроса неясно, сколько каких книг выпустил каждый издатель.

Поскольку зачастую возникает необходимость в просмотре итоговых значений на разных уровнях группировки, многие поставщики систем управления базами данных предоставляют своим пользователям специальные генераторы отчетов. Однако эти генераторы обычно являются отдельным приложением, а не частью SQL, так как подобные отчеты нельзя получить в рамках чисто реляционной модели.

Sybase SQL Server предоставляет для решения этой задачи специальное расширение. Ниже приводится соответствующий пример:

SQL (расширение):

```
select pub_id, type, title_id
from titles
order by pub_id, type
compute count (title_id) by pub_id, type
compute count (title_id) by pub_id
```

Результат:

pub_id	type	title_id
0736	business	BU2075
count		

1		
pub_id	type	title_id
0736	psychology	PS1372
0736	psychology	PS2091
0736	psychology	PS2106
0736	psychology	PS3333
0736	psychology	PS7777
count		

5		
count		

6		
pub_id	type	title_id
0877	NULL	MC3026
count		

1		
pub_id	type	title_id
0877	mod_cook	MC2222
0877	mod_cook	MC3021
count		

2		
pub_id	type	title_id
0877	trad_cook	TC3218
0877	trad_cook	TC4203
0877	trad_cook	TC7777
count		

3		
count		

6		
pub_id	type	title_id
1389	business	BU1032
1389	business	BU1111
1389	business	BU7832
count		

3		
pub_id	type	title_id

1389	popular_comp	PC1035
1389	popular_comp	PC8888
1389	popular_comp	PC9999
	count	-----
		3
	count	-----
		6

В этом отчете содержатся значения строк, итоговые данные по группам и подгруппам. В отличие от других запросов SQL, эти результаты не являются реляционными и не могут использоваться в других операторах SQL.

Нулевые значения и группы. Так как нулевые (NULL) значения представляют неизвестные данные, то нельзя определенно сказать, что одно нулевое значение больше или меньше другого нулевого значения.

Однако, если столбец, по которому выполняется группировка, содержит больше одного нулевого значения, все они будут собраны в одну группу.

Столбец *type* в таблице *titles* содержит нулевое значение. Поэтому при группировке по этому столбцу и подсчете строк в каждой группе будет получен следующий результат:

SQL:

```
select type, count(*)
from titles
group by type
```

Результат:

type	
NULL	1
business	4
mod_cook	2
popular_comp	3
psychology	5
trad_cook	3

В результате содержится строка с типом NULL. Если в запросе вместо функции *count(*)* использовать функцию *count(type)*, то вместо 1 во втором столбце этой строки будет стоять 0:

SQL:

```
select type, count(type)
from titles
group by type
```

Результат:

type	
NULL	0
business	4
mod_cook	2
popular_comp	3
psychology	5
trad_cook	3

Почему в результате до сих пор присутствует нулевая группа, несмотря на значение 0 в правом столбце? Тогда как функция *count(*)* подсчитывает общее количество строк в группе, независимо от значений в конкретных столбцах, функция *count()* работает с определенным столбцом и учитывает при подсчете только ненулевые значения. При выполнении предложения **GROUP BY** система обнаруживает существование типа **NULL** и создает для него отдельную группу. Затем функция *count()* подсчитывает количество элементов в этой группе. В результате находится только значение **NULL**, которое не учитывается при подсчете, и поэтому в правый столбец записывается ноль.

А что, если в столбце, по которому ведется группировка, имеется несколько нулевых значений? Чтобы ответить на этот вопрос, рассмотрим следующий запрос:

SQL:

```
select advance, count(*)
from titles
group by advance
```

Результат:

```
advance
-----
NULL          2
0              1
2000          1
2275          1
4000          2
5000          3
6000          1
7000          3
8000          2
10125         1
15000         1
```

Затраты на две книги неизвестны. Обе они включаются в группу **NULL**. Обратите внимание, что книги без затрат (*advance* = 0) составляют отдельную группу, так **NULL** и 0 — это абсолютно разные значения.

Предложение GROUP BY без агрегирующих функций. Без агрегирующих функций предложение **GROUP BY** напоминает предложение **DISTINCT**. Оно разделяет таблицу на группы и для каждой из них возвращает по одной строке. Помните, что при использовании предложения **GROUP BY** для каждого элемента из списка выбора будет генерироваться по одному значению на набор. Несколько следующих примеров помогут вам в этом разобраться. Сначала получим список всех издателей из таблицы *titles*:

SQL:

```
select pub_id
from titles
```

Результат:

```
pub_id
-----
1389
1389
0736
0736
1389
```

0877
0877
0877
1389
0877
0736
0736
0736
0877
0877
1389
0736
1389

Теперь сгруппируем данные по издателям:

SQL:

```
select pub_id
from titles
group by pub_id
```

Результат:

```
pub_id
-----
0736
0877
1389
```

Такой же результат получился бы и при использовании ключевого слова DISTINCT. Одинаковые результаты также получаются при группировке книг по типам и использовании конструкции SELECT DISTINCT *type*:

SQL:

```
select type
from titles
group by type
```

Результат:

```
type
-----
NULL
business
mod_cook
popular_comp
psychology
trad_cook
```

Предложение GROUP BY с агрегирующими функциями. В предыдущей главе рассматривались агрегирующие функции, текущая глава посвящена предложению GROUP BY. Теперь мы наконец объединим их. Фактически агрегирующие функции и предложение GROUP BY просто созданы друг для друга. GROUP BY создает набор, а агрегирующие функции вычисляют для него конкретное значение. С их помощью можно получить очень полезную информацию.

Давайте рассмотрим несколько характерных запросов. С помощью следующего оператора находятся средние затраты и сумма доходов от продаж *по каждому типу книг*:

SQL:

```
select type, avg(advance), sum(ytd_sales)
from titles
group by type
```

Результат:

type		

NULL	NULL	NULL
business	6281.25	30788
mod_cook	7500	24278
popular_comp	7500	12875
psychology	4255	9939
trad_cook	6333.333333333333	19566

Суммарные значения, полученные с помощью оператора SELECT, предложения GROUP BY и агрегирующих функций составляют два новых столбца результата.

С помощью следующего запроса можно проследить зависимость между ценовыми категориями и средними расходами:

SQL:

```
select price, avg(advance)
from titles
group by price
```

Результат:

price	

NULL	NULL
2.99	12562.5
7	6000
7.99	4000
10.95	2275
11.95	4500
14.99	8000
19.99	3000
20	8000
20.95	7000
21.59	7000
22.95	7000

GROUP BY с предложением WHERE. Как вы могли заметить, при отсутствии в запросе предложения GROUP BY агрегирующие функции применяются ко всей таблице целиком. В этом случае для выбора строк, участвующих в вычислениях, можно было использовать предложение WHERE. Это же верно и при наличии групп.

Совместная работа предложений WHERE и GROUP BY происходит следующим образом. Сначала находятся все строки, удовлетворяющие условиям предложения WHERE. Затем предложение GROUP BY делит отобранные строки на

группы. Строки, не удовлетворяющие условиям предложения WHERE, не включаются ни в одну группу. Ниже приведен соответствующий пример:

SQL:

```
select type, avg(price)
from titles
where advance > $5000
group by type
```

Результат:

type	
business	2.99
mod_cook	2.99
popular_comp	21.48
psychology	14.30
trad_cook	17.97

Этот же запрос без предложения WHERE приводит к другим результатам:

SQL:

```
select type, avg(price)
from titles
group by type
```

Результат:

type	
NULL	NULL
business	13.73
mod_cook	11.49
popular_comp	21.47
psychology	13.50
trad_cook	15.96

В результате выполнения этого запроса появляется новая строка (с типом и средней ценой NULL), а также изменяются все средние цены, за исключением книг по компьютерам. Появление новой строки объяснить очень просто — в предложении WHERE ищутся все строки со значением расходов выше \$5000, поэтому строки с нулевым значением расходов сразу отвергаются. Поэтому, в отличие от последнего запроса, в первом запросе группа NULL отсутствует.

Если выполнить следующий запрос, а затем вручную применить к его результатам предложение WHERE (поставив звездочками строки со значением расходов больше \$5000), то станет ясно, каким образом были получены результаты в первом запросе (с предложениями GROUP BY и WHERE):

SQL:

```
select type, price, advance
from titles
```

Результат:

type	price	advance
business	11.95	5000
business	19.99	5000
business	2.99	10125***

business	19.99	5000
mod_cook	2.99	15000***
mod_cook	19.99	0
NULL	NULL	NULL
popular_comp	22.95	7000***
popular_comp	20	8000***
popular_comp	NULL	NULL
psychology	10.95	2275
psychology	7	6000***
psychology	7.99	4000
psychology	19.99	2000
psychology	21.59	7000***
trad_cook	11.95	4000
trad_cook	14.99	8000***
trad_cook	20.95	7000

В результат включаются только строки со значением расходов выше \$5000. Так как в группе *business* имеется только одна строка, удовлетворяющая этому условию, средняя цена будет равна цене, содержащейся в этой строке. С другой стороны, в группе *popular_comp* условиям предложения WHERE удовлетворяют две строки, и поэтому средняя цена будет равна среднеарифметическому от цен, содержащихся в этих строках.

Обратите внимание, что столбцы в предложении WHERE никоим образом не связаны со столбцами из списка выбора и списка группировки.

Упорядоченные группы

Предложение GROUP BY разделяет строки на наборы, но при этом не упорядочивает их. Чтобы расположить результаты в определенном порядке, нужно использовать предложение ORDER BY. Не забывайте, что предложения в операторе SELECT должны следовать в строго определенной последовательности и предложение ORDER BY всегда располагается после предложения GROUP BY. Например, чтобы найти среднюю стоимость книг по каждому типу, затраты на которые превысили \$5000, и упорядочить результаты по цене, нужно выполнить следующий запрос:

```
SQL:
select type, avg(price)
from titles
where advance >$5000
group by type
order by 2
```

Результат:

```
type
-----
mod_cook      2.99
business      2.99
```

```
psychology      14.295
trad_cook       17.97
popular_comp    21.475
```

ПРЕДЛОЖЕНИЕ HAVING

В самом общем смысле, предложение HAVING работает аналогично предложению WHERE, но применяется к группам. WHERE накладывает ограничения на строки, а HAVING — на группы. Как правило, предложение HAVING используется совместно с предложением GROUP BY.

Если в списке выбора имеются агрегирующие функции, предложение WHERE выполняется перед ними, тогда как предложение HAVING применяется ко всему запросу в целом, после вычисления значений функций и разбиения на группы. Чтобы не запутаться в этих тонкостях, нужно просто запомнить порядок предложений в операторе SELECT. Предложение WHERE всегда стоит после предложения FROM, а предложение HAVING — после предложения GROUP BY. Примеры в конце этого раздела помогут вам уяснить различия между предложениями WHERE и HAVING.

С точки зрения синтаксиса условного выражения, предложения HAVING и WHERE идентичны, отличие состоит лишь в том, что в условии предложения WHERE не могут находиться агрегирующие функции. Кроме того, в большинстве систем элементы предложения HAVING должны включаться в список выбора. На предложение WHERE подобное ограничение не распространяется. В предложении HAVING может содержаться любое количество условий.

Разновидности предложения HAVING

Предложение HAVING работает следующим образом: сначала GROUP BY разделяет строки на наборы (по типу), затем на полученные группы накладываются условия предложения HAVING (в следующем примере, например, удаляются наборы, содержащие только одну книгу):

SQL:

```
select type, count(*)
from titles
group by type
having count(*) > 1
```

Результат:

type	
business	4
mod_cook	2
popular_comp	3
psychology	5
trad_cook	3

Обратите внимание, что в этом случае нельзя просто заменить предложение HAVING на WHERE, так как последнее не допускает использования агрегирующих функций. Ниже приведен пример использования предложения HAVING без агрегирующих функций. В данном случае строки таблицы *titles* группируются по типу и удаляются типы, название которых не начинается с буквы "p":

SQL:

```
select type
from titles
```

```
group by type
having type like 'p%'
```

Результат:

```
type
-----
popular_comp
psychology
```

Если в предложении HAVING есть несколько условий, они объединяются с помощью операторов AND, OR и NOT. Например, оператор, группирующий строки таблицы *titles* по издателям и включающий в конечный результат только группы издателей с идентификационными номерами, большими 0800, суммарными расходами, превышающими \$15000, и средней ценой книг меньше \$20, имеет следующий вид:

SQL:

```
select pub_id, sum(advance), avg(price)
from titles
group by pub_id
having sum(advance) > $15000
and avg(price) < $20
and pub_id > '0800'
```

Результат:

```
pub_id
-----
0877          34000          14.174
1389          30000          18.976
```

В следующем примере иллюстрируется совместное использование предложений GROUP BY, HAVING, WHERE и ORDER BY в операторе SELECT. В результате его выполнения получают те же группы, что и в предыдущем примере, но все вычисления выполняются без учета книг со стоимостью меньше \$5. Результаты также упорядочиваются по столбцу *pub_id*.

SQL:

```
select pub_id, sum(advance), avg(price)
from titles
where price >= $5
group by pub_id
having sum(advance) > $15000
and avg(price) < $20
and pub_id > '0800'
order by pub_id
```

Результат:

```
pub_id
-----
0877          19000          16.97
1389          30000          18.976
```

Предложения HAVING и WHERE

Хотя предложения HAVING и WHERE во многом аналогичны, нужно всегда помнить о различиях в выполняемых ими действиях. Например, если условие *price* >= \$5 вместо WHERE поместить в предложение HAVING, то получится совсем другой результат. (В большинстве систем элементы предложения HAVING должны

включаться в список выбора. Если это имеет место и в вашей системе, то при попытке выполнить следующий запрос вы получите сообщение об ошибке, так как столбец *price* не содержится в списке выбора.)

SQL (вариант):

```
select pub_id, sum(advance), avg(price)
from titles
group by pub_id
having sum(advance) > $15000
and avg(price) < $20
and pub_id > '0800'
and price >= $5
order by pub_id
```

Причина этих различий заключается в том, что предложение WHERE отсеивает строки до группировки, а предложение HAVING — после.

Ниже приводится пара более простых запросов, с помощью которых можно прочувствовать различия между этими двумя предложениями. (Второй запрос не будет работать в системах, требующих присутствия элементов предложения HAVING в списке выбора.)

SQL:

```
select pub_id, type, count(advance)
from titles
where advance > $10000
group by pub_id, type
```

Результат:

pub_id	type	
0736	business	1
0877	mod_cook	1

SQL:

```
select pub_id, type, count(advance)
from titles
group by pub_id, type
having advance > 10000
```

Результат:

pub_id	type	
0736	business	1
0877	mod_cook	2

В первом запросе (при использовании предложения WHERE) создаются две группы, по одной книге в каждой. Во втором примере (при использовании предложения HAVING) также создаются две группы, но в одну из них помещается две книги. Почему?

Ответ состоит в том, что при выполнении предложения WHERE в первом запросе просматриваются все строки и для последующей группировки отбираются только строки со значением расходов больше \$10000 (в следующей таблице эти строки отмечены звездочками).

Pub_id	type	advance
1389	popular_comp	8000
1389	business	5000

0736	psychology	4000
0736	psychology	2000
1389	business	5000
0877	mod_cook	0
0877	trad_cook	8000
0877	trad_cook	4000
1389	popular_comp	7000
0877	NULL	NULL
0736	business	10125***
0736	psychology	2275
0736	psychology	6000
0877	mod_cook	15000***
0877	trad_cook	7000
1389	business	5000
0736	psychology	7000
1389	popular_comp	NULL

После выполнения предложения WHERE для группировки и вычислений остается только две строки. Естественно, что в результате получается две группы, по одной книге в каждой.

Предложение же HAVING применяется ко всему запросу, а не к отдельным строкам. Поэтому результаты второго запроса несколько отличаются. В запросе, использующем предложение HAVING, сначала выполняется группировка и вычисляются значения агрегирующих функций. На первом шаге находятся группы.

pub_id	type	
0736	business	1
0736	psychology	5
0877	NULL	1
0877	mod_cook	2
0877	trad_cook	3
1389	business	3
1389	popular_comp	3

Затем к полученным группам применяются условия предложения HAVING: какие группы содержат одну или несколько строк со значение затрат больше \$10000? Этому условию удовлетворяют две группы (0736 — одна книга по бизнесу и 0877 — две книги по приготовлению пищи).

Тогда как предложение WHERE отсеивает строки перед группировкой данных, предложение HAVING сначала группирует, а затем удаляет строки, не удовлетворяющие заданным в нем условиям.

Еще раз обратите внимание на то, что если ваша система ограничивает элементы предложения HAVING только элементами, перечисленными в списке выбора, то второй запрос выполняться не будет, так как столбец *advance* не включен в список выбора.

ЕЩЕ О НУЛЕВЫХ ЗНАЧЕНИЯХ

В этой и предыдущих главах не раз поднимались вопросы, связанные с использованием нулевых (NULL) значений. В этом разделе мы соберем все эти разрозненные части воедино и подробно обсудим концептуальные вопросы, касающиеся нулевых значений. Это поможет вам избежать некоторых ошибок, связанных с недостаточным пониманием понятия нулевого значения.

Сначала заметим, как это не покажется странным, что даже специалисты считают нули источником головной боли. К. Дж. Дейт, например, долгие годы боролся с ними. В своей статье (“Null Values in Database Management”, *Relational Database: Selected Writings, 1986*) он аргументировал, что нулевые значения должны быть навсегда устранены из баз данных. Дейт писал, что “как правило, вопросы, связанные с использованием нулевых значений, понимаются поверхностно, и любые попытки реализации нулевых значений в прикладных системах являются явно преждевременными”.

На практике могут существовать различные типы неполной информации. Некоторые данные могут вообще быть неизвестны, например записи типа “дата рождения неизвестна” или “будет объявлено позднее”.

Иногда данные, неизвестные в настоящее время, могут стать известными в последствии, как, например, в столбцах *price* и *advance* в нашей базе данных. Другие данные могут пропускаться из-за своей неприменимости. В современных системах управления базами данных нулевые значения используются для представления как неприменимых, так и неизвестных значений.

В ряде случаев точное значение неизвестно, однако определенная информация о нем имеется. Например, точная дата рождения может быть неизвестна, но при этом известно, что она находится в промежутке между 1860 и 1880 годом. Такой тип неизвестной информации называется **различаемыми нулями (distinguished nulls)** — их точные значения неизвестны, но о них имеются некоторые сведения.

Различаемые нули поддерживаются в ряде компьютерных программ, например в статистических пакетах. Однако в системах управления базами данных их использование весьма проблематично. В настоящее время нам неизвестна ни одна подобная система.

В некоторых системах нулевые значения вообще не поддерживаются. Дейт предложил альтернативное SQL понятие нулевого значения, которое будет кратко описано дальше в этой главе. Однако в большинстве современных СУБД нулевые значения используются в том виде, в котором они описаны в этой книге. Давайте обобщим все, что мы говорили о нулевых значениях в предыдущих главах.

Нули и проектирование баз данных

В системах, поддерживающих нулевые значения, при создании таблицы можно указать, в каких столбцах они допускаются. При этом система сможет корректно обработать две следующие ситуации.

- Пользователь добавляет новую строку в таблицу, но не знает, какую информацию нужно вводить в некоторых полях. Система автоматически вводит в них нулевые значения.
- Вы переделываете таблицу, добавляя в нее новый столбец. Что вводить в этот столбец для уже существующих в таблице строк? Система снова должна рассматривать их как нулевые (NULL) значения.

Сравнение нулевых значений

Как отмечалось в главе 3, поскольку нули представляют неизвестные значения, их нельзя сравнить с другими значениями, даже нулевыми. (При этом также нельзя определенно сказать, что величина, представленная значением NULL, не равна какому-либо конкретному значению.)

Например, расходы по книге под названием *Net Etiquette* представлены значением NULL. Означает ли это, что они составляют больше \$5000? Или меньше \$5000? Кроме того, они больше или меньше расходов по книге *The Psychology of Computer Cooking*, которые также представлены нулевым значением?

Ответы на эти вопросы неизвестны. Единственное, что можно сказать — это “может быть”. Другими словами, нулевые значения приводят нас к трехзначной логике, в которой помимо значений “истина” и “ложь” имеется и третье значение — “может быть”.

Трехзначная логика по своей природе часто вызывает определенные затруднения. Предположим, что не слишком опытный пользователь получил список всех названий книг, затраты на которые превысили \$5000 (8 книг). Затем он поинтересовался книгами, затраты на которые составили меньше \$5000, и получил в результате 5 названий. Наконец, он нашел три книги, расходы на которые составили ровно \$5000. После этого пользователь может сделать вывод, что в таблице содержится шестнадцать книг, забыв при этом, что в результатах этих запросов не учитывались строки с нулевыми значениями затрат.

Существуют и другие источники неприятностей. При сортировке книг по затратам, в зависимости от настроек системы, все строки с нулевыми затратами будут сгруппированы либо в начале, либо в конце отчета. Это может ввести пользователя в заблуждение, так как, увидев эти строки рядом друг с другом в отсортированном списке с одинаковым значением затрат (NULL), он может посчитать их равными.

С другой стороны, при использовании ключевого слова DISTINCT в результате остается только одно нулевое значение. Это косвенно свидетельствует о том, что нулевые значения считаются равными друг другу!

Таким образом, при выполнении запроса всегда нужно помнить, что в таблице могут существовать нулевые значения. Предположим, например, что вы хотите уведомить всех авторов, суммарный годовой гонорар которых составил меньше \$600, о том, что они могут не указывать эту сумму в своей налоговой декларации. Чтобы ваше уведомление получили и авторы, данные о гонорарах которых в таблице отсутствуют, вы должны найти их с помощью специального оператора типа IS NULL.

Нули и вычисления

Реальные сложности, связанные с нулевыми значениями, проявляются при выполнении с ними различного рода вычислений.

Ясно, что при выполнении арифметических операций над неизвестными значениями результат также будет неизвестен. Например, можно удвоить значения затрат по всем книгам, но затраты по книге *Net Etiquette* все равно останутся равными NULL.

А что если нужно найти среднее значение затрат? Соответствующий запрос имеет следующий вид:

SQL:

```
select avg(advance)
from titles
```

Результат:

```
-----
5962.5
```

А теперь проверим результаты, полученные SQL. Для этого сначала найдем сумму всех расходов, а затем общее количество книг:

SQL:

```
select sum(advance)
from titles
```

Результат:

```
-----
95400
```

SQL:

```
select count(*)
from titles
```

Вы можете вполне доверять своему калькулятору. В результате деления 95400 на 18 получается совсем не то значение, которое было получено при вычислении средних расходов в операторе SQL. Причина такого расхождения связана с тем, что SQL при подсчете среднего значения не учитывает два нулевых значения. Однако при этом нули учитываются в функции COUNT(*).

Ситуация весьма скользкая. Вы поинтересовались величиной средних расходов и вроде бы получили точный результат. Однако он в принципе не может быть точным, так как основывается на неполных и неточных данных.

SQL сделал все, что мог. Однако в подобных ситуациях все его попытки выглядят довольно жалкими.

Нули и группы

Во многих диалектах SQL при выполнении предложения GROUP BY все нулевые значения помещаются в одну группу.

Значения по умолчанию в качестве альтернативы нулевым значениям

Одной из альтернатив нулевым значениям являются значения по умолчанию, которые автоматически вводятся системой управления базой данных в случае, если пользователь не ввел конкретное значение. Эти значения по умолчанию можно определять в команде CREATE TABLE.

Одно из преимуществ значений по умолчанию перед нулевыми значениями состоит в том, что они обеспечивают вместо неопределенности конкретные значения в столбцах. Подходящим значением по умолчанию для столбца *type* может быть значение “неизвестно”, для столбца *date* — текущая дата.

Большинство специалистов утверждают, что значения по умолчанию никогда не решат проблем, связанных с нулями, которые со всей своей сложностью, хотим мы этого или нет, всегда будут с нами.

Для удобства в некоторых системах управления базами данных вместо нулей можно использовать другие значения. Это делается с помощью специального приложения или с использованием расширения SQL. Например, в следующем примере выводятся типы книг, имеющиеся в базе данных, и соответствующее им количество строк:

SQL:

```
select type, count(*)
from titles
group by type
```

Результат:

```
type
-----
NULL                1
business            4
mod_cook            2
popular_comp       3
psychology         5
trad_cook          3
```

Один из типов не определен. С помощью функции Transact-SQL, под названием ISNULL(), этому типу можно задать другое название:

SQL (вариант)

```
select isnull(type, 'What?'), count(*)
from titles
group by type
```

Результат:

type	
What?	1
business	4
mod_cook	2
popular_comp	3
psychology	5
trad_cook	3

Функция ISNULL() имеет два аргумента — имя столбца и значение, используемое вместо всех найденных в этом столбце нулей. Естественно, реальные значения в базе данных при этом не изменяются.

Ниже приведен запрос, с помощью которого выводятся значения *title_id*, *advance* и *price* для книг, расходы на которые либо составили меньше \$6000, либо неизвестны:

SQL:

```
select title_id, advance, price
from titles
where advance < $6000 or advance is null
order by price
```

Результат:

title_id	advance	price
PC9999	NULL	NULL
MC3026	NULL	NULL
PS7777	4000	7.99
PS2091	2275	10.95
TC4203	4000	11.95
BU1111	5000	11.95
BU7832	5000	19.99
MC2222	0	19.99
PS3333	2000	19.99
BU1032	5000	19.99

В двух строках значения расходов и цены неизвестны (NULL). Чтобы заменить нули на другие значения, попробуйте выполнить следующий запрос:

SQL (вариант):

```
select title_id, isnull(advance, 4000), isnull(price,35.00)
from titles
where advance < $6000 or advance is null
order by price
```

Результат:

title_id	advance	price
PC9999	4000	35.00
MC3026	4000	35.00

PS7777	4000	7.99
PS2091	2275	10.95
TC4203	4000	11.95
BU1111	5000	11.95
BU7832	5000	19.99
MC2222	0	19.99
PS3333	2000	19.99
BU1032	5000	19.99

В результате изменились значения в двух строках. Чтобы узнать, имеются ли подобные возможности в вашей системе, загляните в соответствующее руководство.

РАБОТА С НЕСКОЛЬКИМИ ТАБЛИЦАМИ

В двух следующих главах мы продолжим изучение оператора `SELECT`. В главе 7 рассказывается об объединении таблиц, в главе 8 описываются вложенные запросы, или подзапросы.

Объединение таблиц и сложный анализ данных

ЧТО ТАКОЕ ОБЪЕДИНЕНИЕ

Объединение — последний из трех операторов (выбор, проектирование и объединение), которые должен поддерживать любой язык реляционных запросов. С помощью операции объединения в одном операторе SELECT можно манипулировать данными из разных таблиц. Объединение таблиц сродни сборке из отдельных частей конструктора единого сооружения.

SQL-92 предоставляет для операции объединения несколько ключевых слов (CROSS JOIN, NATURAL, INNER, OUTER), однако они редко используются в коммерческих реализациях. В большинстве систем объединение определяется в предложении WHERE оператора SELECT. Подобно операции проектирования, которая задается в списке выбора оператора SELECT, объединение также определяется неявным образом. Каждое объединение задается для двух таблиц, хотя в одном операторе SELECT может выполняться несколько объединений. При этом столбец, по которому проводится объединение, называется **столбцом соединения** (**connecting column**), или **столбцом объединения** (**join column**). Столбцы соединения должны иметь совпадающие или легко сравнимые значения, описывающие одинаковые или сходные данные в объединяемых таблицах. Например, столбец *title_id* из таблицы *titles* совпадает со столбцом *title_id* из таблицы *salesdetails*.

Столбцы соединения обычно должны иметь одинаковый тип данных. Тогда значения в этих столбцах будут **совместимыми для объединения** (**join-compatible**), так будут принадлежать к одному общему классу данных.

Синтаксис операции объединения

В общих чертах, синтаксис операции объединения имеет следующий вид:

```
SELECT список_выбора
FROM таблица_1, таблица_2 [ , таблица_3]...
WHERE [таблица_1.]столбец оператор_объединения [таблица_2.]столбец
```

В списке таблиц предложения FROM по меньшей мере должно содержаться две таблицы, а столбцы в предложении WHERE должны быть совместимыми для объединения. Если столбцы, по которым выполняется объединение, имеют одинаковые имена, то в списке выбора и в предложении WHERE нужно указать соответствующие им таблицы. Например, чтобы узнать имена редакторов книги *Secrets of Silicon Valley*, необходимо выполнить объединение по столбцам *ed_id*, имеющимся в обеих таблицах:

```
SQL:
select ed_lname, ed_fname, ed_pos
from editors, titleditors
where editors.ed_id = titleditors.ed_id
and titleditors.title_id = 'PC8888'
```

Результат:

<u>ed_lname</u>	<u>ed_fname</u>	<u>ed_pos</u>
DeLongue	Martinella	project
Samuelson	Bernard	project
Kaspchek	Christof	acquisition

В результате этого объединения выбираются имена редакторов со значением "PC8888" в столбце *title_id*. Объединение выполняется по столбцу *ed_id*, содержащемуся в обеих таблицах.

ПОЧЕМУ НЕОБХОДИМО ОБЪЕДИНЕНИЕ

В базе данных, разработанной в соответствии с правилами нормализации, одна таблица вряд ли будет содержать всю необходимую информацию о конкретном объекте. Для выполнения сложного анализа данных, скорее всего, их придется собирать из разных таблиц. Используя реляционную модель, в рамках которой правила нормализации рекомендуют разбивать данные на множество таблиц, описывающих отдельные объекты, с помощью операции объединения вы сможете выполнять самые разнообразные запросы и получать сложные отчеты. Таким образом, операция объединения занимает в реляционной модели одно из ключевых мест.

Объединения и реляционная модель

Объединения стали возможными благодаря реляционной модели, но в то же время и сама реляционная модель просто не может существовать без них.

Операция объединения допускается в системах управления реляционными базами данных вследствие того, что в рамках этой модели можно брать данные из разных таблиц и создавать на их основе новые таблицы, и отслеживать непредусмотренные заранее зависимости. Зависимости между данными проявляются при выполнении запроса, а не при создании базы данных.

Вам не нужно заранее знать, какие данные будут впоследствии объединяться. Новые зависимости между данными из разных таблиц становятся видны при их объединении. Например, чтобы узнать имена членов организации, которые внесли свой вклад в избирательную кампанию, можно использовать следующий запрос:

SQL:

```
select board_members.name
from board_members, political_contributors
where board_members.name = political_contributors.name
```

Или, используя нашу базу данных, с помощью объединения можно найти авторов, которые когда-либо выполняли обязанности редакторов:

SQL:

```
select ed_lname
from editors, authors
where ed_id = au_id
```

Этот запрос "с изюминкой". Так как авторы и редакторы могут иметь одинаковые фамилии, объединение выполняется по столбцу с идентификационными номерами, которые однозначно идентифицируют авторов и редакторов. Таким образом, первичные ключи часто могут оказываться весьма полезными и при объединении таблиц.

Операция объединения обеспечивает неограниченную гибкость при добавлении в базу данных новых типов данных. Вы всегда можете создать таблицу, содержащую информацию о новых объектах. Если в этой таблице имеется подходящий столбец, то с помощью объединения ее можно связать с существующими таблицами.

Этот процесс иллюстрирует рост реляционной базы данных: если возникает потребность в разбиении таблиц, с помощью объединения всегда можно перестроить строки, если же нужны новые таблицы, то с помощью того же объединения их можно связать с существующими данными.

Например, в базе данных *bookbiz* для хранения информации о выписанных счетах может потребоваться таблица *invoices*. Ее можно связать с таблицами *sales* и *salesdetails* по столбцу *sonum* (номер накладной) и, может быть, с новой таблицей *stores* по столбцу *stor_id*.

При проектировании структуры базы данных в таблицах нужно обязательно предусматривать столбцы, по которым впоследствии может выполняться объединение. Чаще всего для этих целей используются столбцы первичных или внешних ключей. Например, в нашей базе данных первичный ключ (*titles.title_id*) объединяется с соответствующими внешними ключами (*salesdetails.title_id*, *titleauthors.title_id* и *titleditors.title_id*). Без таких столбцов выполнение запросов ко многим таблицам будет существенно затруднено.

ПРИМЕР ОБЪЕДИНЕНИЯ

Для выполнения объединения, как минимум, нужно выполнить два действия.

- Поместить имя одной или нескольких таблиц в список предложения FROM.
- Добавить условия в предложение WHERE, на основе которых будет выполняться объединение по заданным столбцам.

Для определения связи между столбцами, по которым будет выполняться объединение, можно использовать любые реляционные операторы (однако, чаще всего, применяется оператор равенства). Например, чтобы узнать, кто из редакторов живет недалеко от офиса издателя, нужно выполнить соответствующее объединение. В следующем запросе находятся имена редакторов, живущих в том же городе, в котором расположена компания Algodata Infosystems:

SQL:

```
select ed_lname, ed_id, editors.city, pub_name, publishers.city
from editors, publishers
where editors.city = publishers.city
and pub_name = 'Algodata Infosystems'
```

Результат:

ed_lname	ed_id	editors.city	pub_name	publishers.city
Kaspchek	943-88-7920	Berkeley	Algodata Infosystems	Berkeley
DeLongue	321-55-8906	Berkeley	Algodata Infosystems	Berkeley

В результате выполнения запроса на объединение находятся два редактора (DeLongue и Kaspchek), живущие в том же городе (Berkeley), в котором расположена Algodata Infosystems. Первоначально эта информация нигде не фигурировала. Фактически, нам нужно было знать только имя издателя, а вся дополнительная информация была получена с помощью операции объединения. Мы просто сказали системе: “Выполни объединение по городам и найди живущих в них редакторов”.

Проверка правильности объединения

В реляционной теории объединение является проектированием и отображением произведения. Произведение (или **декартово произведение (Cartesian product)**) — это набор всевозможных комбинаций строк из двух таблиц.

Сначала система находит все возможные комбинации строк из двух таблиц, затем устраняет из рассмотрения строки, не удовлетворяющие условиям проектирования (выбор столбцов) и отображения (выбор строк). Детали этой процедуры на самом деле достаточно сложны и зависят от конкретной реализации системы.

(Более подробная информация содержится в разделе “Как объединения обрабатываются системой” в конце этой главы.)

Запросы на объединение, в большей степени, чем другие запросы SQL, нуждаются в тщательной проверке, чтобы быть уверенными, что в результате их выполнения получается именно то, что вы хотите. Недостаточно конкретный запрос может вернуть результат, с которым придется долго возиться, чтобы извлечь из него действительно полезную информацию.

КАК ПОЛУЧИТЬ ХОРОШЕЕ ОБЪЕДИНЕНИЕ

Какой столбец лучше всего выбирать для объединения? Идеально для этого подходят ключевые столбцы таблицы — первичные или внешние. Если ключ является составным, то объединение можно выполнять по всем входящим в него столбцам.

Так как первичный ключ логически связан с соответствующими внешними ключами других таблиц, то ключевые столбцы обычно наилучшим образом подходят для объединения. Такие объединения, скорее всего, будут полезными и логически оправданными, так как их в свое время планировал разработчик базы данных. Объединение на основе пары “первичный ключ-внешний ключ” подразумевает их совместимость, чтобы обеспечить целостность базы данных.

Чтобы в результате объединения получались полноценные результаты, сравниваемые столбцы должны содержать аналогичные значения — значения, принадлежащие одному общему классу данных. Такие столбцы должны иметь одинаковые или близкие типы данных и сравнимые значения. Кроме типа, конечно, нужно учитывать и семантику. Например, можно выполнить объединение по столбцу с возрастом авторов (25, 30, 50) и столбцу *qty_shipped* из таблицы *salesdetails*, однако полученные результаты не будут представлять никакой ценности. Эти столбцы соответствуют друг другу по типу, но логически абсолютно не связаны.

Столбцы, по которым выполняется объединение, не обязательно должны иметь одинаковые имена (хотя чаще всего именно так и бывает). Типы данных объединяемых столбцов должны быть совместимыми — система должна уметь преобразовывать их друг в друга. Например, такие преобразования могут выполняться между числовыми столбцами (типы *integer*, *decimal*, *float*) или между символьными столбцами (типы *character*, *varchar*, *datetime*). Другими словами, совместимыми являются как символьные, так и числовые типы. Кроме того, можно явно задать преобразование типов, используя для этого функции преобразования типов, описанные в главе 3.

Объединения и нулевые значения

Если в столбцах, по которым проводится объединение, имеются нулевые (NULL) значения, они пропускаются, так как нет никаких оснований считать одно неизвестное значение совпадающим с другим неизвестным значением.

УЛУЧШЕНИЕ ЧИТАЕМОСТИ РЕЗУЛЬТАТОВ ОБЪЕДИНЕНИЯ

При объединении таблиц система сравнивает данные в указанных полях и отображает результаты в виде таблицы, в которую входят строки, соответствующие каждому успешному объединению. При этом могут повторяться одни и те же значения. Так, в предыдущем примере в результатах дважды фигурировала компания *Algodata Infosystem*, хотя этот издатель только раз появляется в таблице *publishers*.

Запрос на объединение не вносит никаких изменений в таблицы, а просто позволяет системе манипулировать с данными из разных таблиц так, как если бы они принадлежали одной таблице.

Вам не нужно дважды указывать имя столбца объединения в списке выбора, кроме того, вы можете вообще не включать его в результаты для успешного выполнения запроса. Однако в списке выбора или в предложении *WHERE* вам может понадобиться описать принадлежность столбца таблице.

Предыдущий пример можно переделать следующим образом:

SQL:

```
select ed_lname
from editors, publishers
where editors.city = publishers.city
and pub_name = 'Algodata Infosystems'
```

Результат:

```
ed_lname
-----
Kaspchek
DeLongue
```

Выбор столбцов для запросов на объединение

Как и в любом другом операторе SELECT, столбцы, перечисляемые после ключевого слова SELECT в запросе на объединение, будут в указанном порядке включаться в результат.

При использовании оператора SELECT * столбцы помещаются в результат в порядке их определения в операторе CREATE TABLE.

Чтобы найти названия книг, связанные с заказом номер 1, необходимо объединить таблицы *salesdetails* и *titles* по столбцу *title_id*:

SQL:

```
select *
from titles, salesdetails
where titles.title_id = salesdetails.title_id
and sonum =1
```

Результат:

title_id	title	type	pub_id	price
PS2091	Is Anger the Enemy?	Psychology	0736	10.95

продолжение

advance	ytd_sales	contract	notes	pubdate
2275	2045	1	Carefully researched study of the effects of strong emotions on the body. Metabolic charts included.	15.06.85

продолжение

sonum	qty_ordered	qty_shipped	title_id	date_shipped
1	75	75	PS2091	15.09.85

В результате получается таблица с одной строкой и пятнадцатью столбцами. Так как она по ширине не помещается на экране, система будет тем или иным способом размещать ее на нескольких строках, что будет создавать определенные трудности для понимания результатов. Поэтому мы рекомендуем вам очень аккуратно выбирать столбцы для объединения и с большой осторожностью пользоваться шаблонами (звездочками).

Имена обеих таблиц должны указываться в списке таблиц. Порядок их следования при этом будет влиять на результат только при использовании в списке выбора оператора SELECT *.

В списке таблиц должны перечисляться все участвующие в объединении таблицы. Вы можете объединить более двух таблиц, соответствующие примеры содержатся дальше в этой главе.

Псевдонимы в списке таблиц улучшают читаемость запросов

Чтобы ускорить ввод запросов и сделать их более понятными, в списке таблиц можно определить псевдонимы таблиц (сокращенные имена). Это особенно полезно при выполнении объединения по столбцам с одинаковыми именами, когда всякий раз при их использовании приходится указывать имена соответствующих таблиц.

Вот как можно использовать псевдонимы в запросе, с помощью которого найдутся авторы, живущие в одном городе с каким-либо издателем:

SQL:

```
select au_lname, au_fname
from authors a, publishers p
where a.city = p.city
```

Результат:

```
au_lname      au_fname
-----
Carson        Cheryl
Bennet        Abraham
```

Если вы хотите вывести и название города, соответствующий столбец нужно поместить в список выбора вместе с одним из псевдонимов таблиц.

ОПРЕДЕЛЕНИЕ УСЛОВИЙ ОБЪЕДИНЕНИЯ

Объединение обычно основывается на равенстве или совпадении значений в объединяемых столбцах. Объединение, основанное на равенстве, легко распознать по логическому оператору “=” в соответствующей объединению части предложения WHERE. Объединения также могут быть построены и на других условиях: любом логическом операторе или специальном операторе для определения внешнего объединения (outer join), которое мы обсудим далее.

Помимо равенства, для задания условия могут использоваться следующие логические операторы:

<i>Символ</i>	<i>Значение</i>
>	Больше чем
>=	Больше или равно
<	Меньше чем
<=	Меньше или равно
!= (или <>)	Не равно

В Transact-SQL также имеются операторы !> и !<, соответственно эквивалентные операторам <= и >=.

На жаргоне реляционных баз данных объединения, построенные только на операторах сравнения, называются **theta-объединениями**.

Объединения, основанные на равенстве

Объединения, основанные на равенстве, как бы “сшивают” столбцы из участвующих в объединении таблиц.

В следующем запросе находятся названия книг, связанные с определенным номером заказа:

SQL:

```
select title, t.title_id, sonum, sd.title_id
from titles t, salesdetails sd
where sonum = 14
and t.title_id = sd.title_id
```

Результат:

title	t.title_id	sonum	sd.title_id
Computer Phobic and Non-Phobic Individuals: Behavior Variations	PS1372	14	PS1372
Life Without Fear	PS2106	14	PS2106
Prolonged Data Deprivation: Four Case Studies	PS3333	14	PS3333
Emotional Security: A New Algorithm	PS7777	14	PS7777

Так как в данном случае нет никакой необходимости в дублировании информации, из результата можно изъять один из столбцов объединения. Такое отображение называется **естественным объединением (natural join)**. Предыдущий запрос примет следующий вид:

SQL:

```
select title, t.title_id, sonum
from titles t, salesdetails sd
where sonum = 14
and t.title_id = sd.title_id
```

Результат:

title	title_id	sonum
Computer Phobic and Non-Phobic Individuals: Behavior Variations	PS1372	14
Life Without Fear	PS2106	14
Prolonged Data Deprivation: Four Case Studies	PS3333	14
Emotional Security: A New Algorithm	PS7777	14

Теперь мы получили естественное объединение, так как столбец *title_id* не появляется в результатах дважды. В некоторых системах естественное объединение выполняется по умолчанию, что упрощает восприятие результатов. Не имеет значения, из какой таблицы включается столбец *title_id*, однако в списке выбора вы должны обязательно указать его принадлежность определенной таблице.

Объединения, не основанные на равенствах

Объединения, не основанные на равенствах, могут быть описаны в терминах используемых в них условий, например “объединение меньше чем”, “объединение больше чем” и т.д. В следующем примере объединения типа меньше чем находятся все заказы, отправленные после получения денег:

SQL:

```
select distinct s.sonum, s.stor_id, s.sdate, sd.date_shipped
from sales s, salesdetails sd
where s.sdate < sd.date_shipped
and s.sonum = sd.sonum
```

Результат:

sonum	stor_id	sdate	date_shipped
1	7066	13.09.85	15.09.85
2	7067	14.09.85	15.09.85
3	7131	14.09.85	18.09.85

4	7131	14.09.85	18.09.85
6	8042	14.09.85	22.09.85
7	6380	13.09.85	20.09.85
9	8042	11.03.88	28.03.88
10	7896	28.10.87	29.10.87
11	7896	12.12.87	12.01.88
12	8042	22.05.87	24.05.87
14	7131	29.05.87	13.06.87
15	7067	15.06.87	17.06.87
19	7896	21.02.88	15.03.88

В следующем примере по столбцу *title_id* объединяются таблицы *titles* и *roysched*:

SQL:

```
select t.title_id, t.ytd_sales, r.royalty
from titles t, roysched r
where t.title_id = r.title_id
and t.ytd_sales >= r.lorange and t.ytd_sales <= r.hirange
```

Результат:

title_id	ytd_sales	royalty
PC8888	4095	0.1
BU1032	4095	0.1
PS7777	3336	0.1
PS3333	4072	0.1
BU1111	3876	0.1
MC2222	2032	0.12
TC7777	4095	0.1
TC4203	15096	0.14
PC1035	8780	0.16
BU2075	18722	0.18
PS2091	2045	0.12
PS2106	111	0.1
MC3021	22246	0.2
TC3218	375	0.1
BU7832	4095	0.1
PS1372	375	0.1

Объединение таблицы с самой собой: самообъединение

Самообъединение является другим вариантом объединения на основе равенства. При самообъединении сравниваются значения внутри столбца одной таблицы.

Например, самообъединение можно использовать для поиска авторов, живущих в Окленде, Калифорния, и имеющих одинаковые почтовые коды. Так как в этом запросе таблица *authors* объединяется сама с собой, то она выступает сразу в двух ролях. Поэтому для различения этих ролей в списке таблиц предложения FROM ей нужно задать два разных псевдонима — *au1* и *au2*. Эти псевдонимы также будут использоваться при задании имен столбцов. Вот как выглядит этот запрос:

SQL:

```
select au1.au_fname, au1.au_lname, au1.zip
```

```
from authors aul, authors au2
where aul.city = 'Oakland'
and aul.zip = au2.zip
```

Результат:

au_fname	au_lname	zip
Marjorie	Green	94618
Dirk	Stringer	94609
Dirk	Stringer	94609
Dirk	Stringer	94609
Dick	Straight	94609
Dick	Straight	94609
Dick	Straight	94609
Livia	Karsen	94609
Livia	Karsen	94609
Livia	Karsen	94609
Stearns	MacFeather	94612

В этих результатах достаточно сложно разобраться и они не кажутся правильными. Чтобы прояснить ситуацию, давайте сначала устраним повторяющиеся строки для трех авторов (Straight, Stringer и Karsen). Для этого используем в списке выбора ключевое слово DISTINCT.

SQL:

```
select distinct aul.au_fname,aul.au_lname, aul.zip
from authors aul, authors au2
where aul.city = 'Oakland'
and aul.zip = au2.zip
```

Результат:

au_fname	au_lname	zip
Dick	Straight	94609
Dirk	Stringer	94609
Livia	Karsen	94609
Marjorie	Green	94618
Stearns	MacFeather	94612

Теперь в результате нет повторяющихся строк, но при этом в нем просто перечисляются все авторы, живущие в Окленде, а не только имеющие одинаковые почтовые коды. Что происходит?

При самообъединении все значения почтовых кодов сравниваются сами с собой и поэтому в результат автоматически попадают все авторы из Окленда. Чтобы устранить из результата авторов, чьи почтовые коды совпадают только с их же собственными кодами, в предложение WHERE нужно добавить дополнительное условие:

SQL:

```
select distinct aul.au_fname,aul.au_lname, aul.zip
from authors aul, authors au2
where aul.city = 'Oakland'
and aul.zip = au2.zip
and aul.au_id != au2.au_id
```

Результат:

au_fname	au_lname	zip
Dick	Straight	94609
Dirk	Stringer	94609
Livia	Karsen	94609

Если не использовать в запросе ключевое слово `DISTINCT`, в результате снова появятся повторяющиеся строки:

SQL:

```
select aul.au_fname,aul.au_lname, aul.zip
from authors aul, authors au2
where aul.city = 'Oakland'
and aul.zip = au2.zip
and aul.au_id != au2.au_id
```

Результат:

au_fname	au_lname	zip
Dirk	Stringer	94609
Dirk	Stringer	94609
Dick	Straight	94609
Dick	Straight	94609
Livia	Karsen	94609
Livia	Karsen	94609

Устранение дублирующихся строк особенно важно, когда требуется узнать “сколько?”, а не “кто именно?”

Использование при самообъединении оператора неравенства

В предыдущем примере для устранения повторяющихся строк при самообъединении использовался оператор неравенства. Вместо него также может применяться ключевое слово `NOT`. Выражение `NOT имя_столбца = имя_столбца` эквивалентно выражению `имя_столбца != имя_столбца`. Таким образом предыдущий запрос можно записать в следующем виде:

SQL:

```
select distinct aul.au_fname,aul.au_lname, aul.zip
from authors aul, authors au2
where aul.city = 'Oakland'
and aul.zip = au2.zip
and not aul.au_id = au2.au_id
```

В качестве другого примера использования при самообъединении оператора неравенства рассмотрим запрос, с помощью которого находятся значения идентификаторов книг и авторов для всех книг, написанных в соавторстве. Другими словами, находятся все строки в таблице `titleauthors` с одинаковыми значениями `title_id`, но разными значениями `au_id`.

SQL:

```
select distinct t1.title_id, t1.au_id
from titleauthors t1, titleauthors t2
where t1.title_id = t2.title_id
and t1.au_id != t2.au_id
order by t1.title_id
```

Результат:

title_id	au_id
BU1032	213-46-8915
BU1032	409-56-7008
BU1111	267-41-2394
BU1111	724-80-9391
MC3021	722-51-5454
MC3021	899-46-2035
PC8888	427-17-2319
PC8888	846-92-7186
PS1372	724-80-9391
PS1372	756-30-7391
PS2091	899-46-2035
PS2091	998-72-3567
TC7777	267-41-2394
TC7777	472-27-2349
TC7777	672-71-3249

Объединение нескольких таблиц

Таблица *titleauthors* в нашей базе данных хорошо иллюстрирует ситуацию, когда может потребоваться объединение более двух таблиц. Для получения полной информации о книгах и их авторах в случае базы данных *bookbiz* необходимо объединить три таблицы.

Например, чтобы найти названия всех книг определенного типа (*trad_cook*) и имена их авторов, нужно выполнить следующий запрос:

SQL:

```
select au_lname, au_fname, title
from authors a, titles t, titleauthors ta
where a.au_id = ta.au_id
and t.title_id = ta.title_id
and t.type= 'trad_cook'
```

Результат:

au_lname	au_fname	title
O'Leary	Michael	Sushi, Anyone?
Gringlesby	Burt	Sushi, Anyone?
Yokomoto	Akiko	Sushi, Anyone?
Blotchet-Halls	Reginald	Fifty Years in Buckingham Palace Kitchens
Panteley	Sylvia	Onions, Leeks, and Garlic: Cooking Secrets of the Mediterranean

В структуре нашей базы данных *titleauthors* используется в качестве промежуточной таблицы для объединения таблиц *authors* и *titles*. Такое “трехступенчатое” объединение необходимо даже для выяснения такого простого факта, кто написал какую книгу.

Несколько условий объединения почти всегда соединяются с помощью оператора AND, как, например, в предыдущем примере. Оператор OR в этом случае используется крайне редко, поскольку получаемые с его помощью условия недостаточно ограничивают объединение.

Внешние объединения

При выполнении описываемых до сих пор объединений в результат включались только строки, удовлетворяющие условиям объединения. Однако вы можете захотеть поместить туда также и строки одной из таблиц, не удовлетворяющие условиям объединения.

При внешнем объединении обычно в результат дополнительно включаются строки из первой таблицы, не удовлетворяющие условиям объединения. С помощью другого оператора внешнего объединения в результат можно включить все строки из второй таблицы, не удовлетворяющие условиям объединения, или даже такие строки из обеих таблиц. SQL-92 поддерживает только либо левое (LEFT), либо правое (RIGHT) внешнее объединение. Так как производители поддерживали эти возможности еще до появления общего стандарта, различные их реализации могут сильно различаться. В одних системах внешнее объединение определяется в предложении FROM, в других — в предложении WHERE. За подробной информацией обратитесь к руководству по своей системе. В Transact-SQL для определения внешнего определения в предложении WHERE используются следующие обозначения:

Символ	Значение
*=	Включить все строки первой таблицы
=*	Включить все строки второй таблицы

Эти же обозначения поддерживаются и в SQL Anywhere.

Как вы помните, в результате выполнения запроса, в котором ищутся авторы, живущие в одном городе с издателем, возвращается два имени (Abraham Bennet и Cheryl Carson).

Чтобы включить в результат все имена из таблицы *authors*, независимо от того, удовлетворяют ли они условиям объединения, можно использовать внешнее объединение:

SQL:

```
select au_fname, au_lname, pub_name
from authors, publishers
where authors.city *= publishers.city
```

Результат:

au_fname	au_lname	pub_name
Abraham	Bennet	Algodata Infosystems
Marjorie	Green	NULL
Cheryl	Carson	Algodata Infosystems
Albert	Ringer	NULL
Anne	Ringer	NULL
Michel	DeFrance	NULL
Sylvia	Panteley	NULL
Heather	McBadden	NULL
Dirk	Stringer	NULL
Dick	Straight	NULL
Livia	Karsen	NULL
Stearns	MacFeather	NULL
Ann	Dull	NULL
Akiko	Yokomoto	NULL
Michael	O'Leary	NULL
Burt	Gringlesby	NULL
Morningstar	Greene	NULL

Johnson	White	NULL
Innes	del Castillo	NULL
Sheryl	Hunter	NULL
Chastity	Locksley	NULL
Reginald	Blotchet-Halls	NULL
Meander	Smith	NULL

Оператор внешнего объединения “*=” заставляет систему включить в результат все строки из первой таблицы (в нашем случае *authors*), независимо от того, совпадают ли значения в ее столбце *city* с соответствующими значениями из таблицы *publishers*.

Оператор внешнего объединения “=*” заставляет систему включить в результат все строки из второй таблицы, независимо от того, совпадают ли их значения со значениями из первой таблицы.

Подставив в предыдущий запрос оператор правого внешнего объединения, вы получите имена издателей, находящихся в городе, в котором живет какой либо автор, а также имена издателей, не удовлетворяющих этому условию:

SQL:

```
select au_fname, au_lname, pub_name
from authors, publishers
where authors.city =* publishers.city
```

Результат:

au_fname	au_lname	pub_name
NULL	NULL	New Age Books
NULL	NULL	Binnet & Hardley
Cheryl	Carson	Algodata Infosystems
Abraham	Bennet	Algodata Infosystems

Как и в случае обычных объединений, для ограничения результатов внешнего объединения можно использовать условные операторы. Давайте сначала рассмотрим объединение, основанное на равенстве, а затем сравним его результаты с результатами внешнего объединения. Например, чтобы найти названия книг, количество проданных экземпляров которых превысило 50, можно использовать следующий запрос:

SQL:

```
select sonum, title
from salesdetails sd, titles t
where qty_ordered > 50
and sd.title_id = t.title_id
```

Результат:

sonum	title
1	Is Anger the Enemy?

Чтобы, кроме того, найти все названия книг, не удовлетворяющих этому условию, нужно воспользоваться внешним объединением:

SQL:

```
select sonum, title
from salesdetails sd, titles t
where qty_ordered > 50
and sd.title_id =* t.title_id
```

Результат:

sonum	title
NULL	Secrets of Silicon Valley
NULL	The Busy Executive's Database Guide
NULL	Emotional Security: A New Algorithm
NULL	Prolonged Data Deprivation: Four Case Studies
NULL	Cooking with Computers: Surreptitious Balance Sheets
NULL	Silicon Valley Gastronomic Treats
NULL	Sushi, Anyone?
NULL	Fifty Years in Buckingham Palace Kitchens
NULL	But Is It User Friendly?
NULL	The Psychology of Computer Cooking
NULL	You Can Combat Computer Stress!
1	Is Anger the Enemy?
NULL	Life Without Fear
NULL	The Gourmet Microwave
NULL	Onions, Leeks, and Garlic: Cooking Secrets of the Mediterranean
NULL	Straight Talk About Computers
NULL	Computer Phobic and Non-Phobic Individuals: Behavior Variations
NULL	Net Etiquette

КАК ОБЪЕДИНЕНИЯ ОБРАБАТЫВАЮТСЯ СИСТЕМОЙ

В запросе на объединение совсем не сложно допустить ошибку. Если в получаемом результате содержится слишком много строк или повторов, то, скорее всего, в запрос нужно внести какие-нибудь изменения. Однако по получаемым результатам очень трудно разобраться в том, как запросы на объединение обрабатываются в реляционных системах.

Первым шагом обработки запроса на объединение является построение декартова произведения таблиц — всех возможных комбинаций строк из обеих таблиц. После получения декартова произведения столбцы из списка выбора используются для выполнения операций проектирования, а условия в предложении WHERE — для выбора, чтобы устранить строки, которые не удовлетворяют условиям объединения.

Декартово произведение — это матрица всех возможных комбинаций, которые могут удовлетворять условиям объединения. Если в каждой таблице содержится по одной строке, то и комбинация будет единственной. Для примера рассмотрим две таблицы, содержащие по одной строке:

one	two
a	b
three	four
c	d

Декартово произведение этих таблиц имеет следующий вид:

one	two	three	four
a	b	c	d

Если в каждой таблице содержится по две строки, то в декартовом произведении будет четыре строки:

Первая таблица:

one	two
a	b
c	d

Вторая таблица:

three	four
c	d
e	f

Декартово произведение:

one	two	three	four
a	b	c	d
a	b	e	f
c	d	c	d
c	d	e	f

Количество строк в декартовом произведении равно числу строк первой таблицы, умноженному на число строк второй таблицы.

Чтобы получить декартово произведение, можно выполнить запрос на объединение, не указывая при этом условия объединения.

ОПЕРАТОР UNION

Хотя оператор UNION не выполняет объединение, он позволяет объединить результаты нескольких запросов. Этот оператор полезен, когда вы хотите одновременно увидеть аналогичные данные из различных таблиц. Упрощенный синтаксис оператора UNION имеет следующий вид:

```
оператор_select  
UNION  
оператор_select
```

В результате выполнения следующего запроса находятся все авторы и редакторы, живущие в Окленде или Беркли:

SQL:

```
select au_fname, au_lname, city  
from authors  
where city in ('Oakland', 'Berkeley')  
union  
select ed_fname, ed_lname, city  
from editors  
where city in ('Oakland', 'Berkeley')
```

Обратите внимание, что в обоих запросах используется одинаковое число элементов с совместимыми типами данных. В качестве заголовков столбцов UNION использует имена столбцов из первого запроса:

Результат:

au_fname	au_lname	city
Abraham	Bennet	Berkeley
Bernard	Samuelson	Oakland
Cheryl	Carson	Berkeley
Christof	Kaspchek	Berkeley

Dick	Straight	Oakland
Dirk	Stringer	Oakland
Livia	Karsen	Oakland
Marjorie	Green	Oakland
Martinella	DeLongue	Berkeley
Stearns	MacFeather	Oakland

Хотя каждый оператор SELECT может иметь свое собственное предложение WHERE, весь запрос может использовать только одно предложение ORDER BY. Оно должно располагаться в последнем операторе SELECT и применяться ко всему результату.

В следующем запросе для получения лучше читаемых результатов назначаются новые заголовки столбцов и используется предложение ORDER BY для сортировки по названию города.

SQL:

```
select au_fname as First_name, au_lname as Last_name, city as City
from authors
where city in ('Oakland', 'Berkeley')
union
select ed_fname, ed_lname, city
from editors
where city in ('Oakland', 'Berkeley')
order by 3
```

Результат:

First_name	Last_name	City
Abraham	Bennet	Berkeley
Cheryl	Carson	Berkeley
Christof	Kaspchek	Berkeley
Martinella	DeLongue	Berkeley
Bernard	Samuelson	Oakland
Dick	Straight	Oakland
Dirk	Stringer	Oakland
Livia	Karsen	Oakland
Marjorie*	Green	Oakland
Stearns	MacFeather	Oakland

По умолчанию оператор UNION устраняет из результата повторяющиеся строки. Это может привести к определенным недоразумениям, особенно если в запросе фигурирует только один столбец. Например, в таблице *authors* присутствует 23 города, а в таблице *publishers* — 3. А при выполнении оператора UNION в результате получается только 18 строк.

SQL:

```
select count(city)
from authors
```

Результат:

SQL:

```
select count(city)
from publishers
```

Результат:

```
-----
3
```

SQL:

```
select city
from authors
union
select city
from publishers
```

Результат:

```
city
-----
Ann Arbor
Berkeley
Boston
Corvallis
Covelo
Gary
Lawrence
Menlo Park
Nashville
Oakland
Palo Alto
Rockville
Salt Lake City
San Francisco
San Jose
Vacaville
Walnut Creek
Washington
```

Почему только 18? Оператор UNION устранил из результата все повторяющиеся строки.

Чтобы отобразить все строки, после оператора UNION нужно добавить ключевое слово ALL. За дополнительной информацией обращайтесь к руководству по своей системе.

Полезный трюк с оператором UNION

Оператор UNION можно использовать в качестве разновидности оператора IF для отображения различных значений для одного поля, в зависимости от значений в других полях. Без оператора UNION для получения аналогичного результата потребовалось бы выполнение нескольких запросов.

Пусть, например, нужно получить список книг, указав для каждой из них процентное снижение цены и новую стоимость. Стоимость книг до \$7 снижается на 20 процентов, между \$7 и \$15 — на 10 процентов, выше \$15 — на 30 процентов.

Без оператора UNION вам потребовалось бы выполнить три отдельных запроса и поместить результаты в новую таблицу.

Оператор UNION выполняет все это за один проход:

SQL:

```
select '20% off', title, price, price * .80
from titles
where price < $7.00
union
select '10% off', title, price, price * .90
from titles
where price between $7.00 and $15.00
union
select '30% off', title, price, price * .70
from titles
where price > $15.00
```

Добавив заголовки столбцов, вы получите следующий результат:

Результат:

discount	title	old	new
10% off	Cooking with Computers: Surreptitious Balance Sheets	11.95	10.755
10% off	Emotional Security: A New Algorithm	7.99	7.191
10% off	Fifty Years in Buckingham Palace Kitchens	11.95	10.755
10% off	Is Anger the Enemy?	10.95	9.855
10% off	Life Without Fear	7	6.3
10% off	Sushi, Anyone?	14.99	13.491
20% off	The Gourmet Microwave	2.99	2.392
20% off	You Can Combat Computer Stress!	2.99	2.392
30% off	But Is It User Friendly?	22.95	16.065
30% off	Computer Phobic and Non-Phobic Individuals: Behavior Variations	21.59	15.113
30% off	Onions, Leeks, and Garlic: Cooking Secrets of the Mediterranean	20.95	14.665
30% off	Prolonged Data Deprivation: Four Case Studies	19.99	13.993
30% off	Secrets of Silicon Valley	20	14
30% off	Silicon Valley Gastronomic Treats	19.99	13.993
30% off	Straight Talk About Computers	19.99	13.993
30% off	The Busy Executive's Database Guide	19.99	13.993

Чтобы отсортировать результаты по цене, можно воспользоваться предложением ORDER BY.

ПОДЗАПРОСЫ

В следующей главе мы обратимся к подзапросам. Подзапросы обеспечивают другой метод для выполнения запросов к нескольким таблицам. Хотя объединения и подзапросы часто могут заменять друг друга, в некоторых случаях желаемый результат можно получить только с помощью подзапроса.

Структурированные запросы и подзапросы

ЧТО ТАКОЕ ПОДЗАПРОС

Подзапрос (subquery) — это дополнительный метод манипуляций с несколькими таблицами. Это — оператор `SELECT`, вложенный:

- в предложение `WHERE`, `HAVING` или `SELECT` другого оператора `SELECT`;
- в оператор `INSERT`, `UPDATE` или `DELETE`;
- в другой подзапрос.

Именно возможность вложения операторов SQL друг в друга является причиной, по которой SQL первоначально был назван Structured Query Language (язык структурированных запросов). Термин *подзапрос* часто используется для ссылки на всю совокупность операторов, которая включает один или несколько подзапросов, а также на отдельное вложение. Каждый включающий оператор — следующий по старшинству уровень в подзапросе — представляет собой внешний уровень для внутреннего подзапроса.

Подзапросы — достаточно сложная тема, поскольку существует два типа их обработки — некоррелированная и коррелированная — и три возможности соединения подзапроса с внешним предложением. После ознакомления с общим синтаксисом подзапроса мы посмотрим, как работают некоррелированный и коррелированный подзапросы, и сравним их с объединениями. Затем мы углубимся в вопросы соединений и рассмотрим некоррелированный и коррелированный примеры каждого из них.

Упрощенный синтаксис подзапроса

Упрощенная форма синтаксиса подзапроса (рис. 8.1) иллюстрирует вложение подзапроса в операторе `SELECT`. (В большинстве примеров, приведенных в этой главе, используется именно этот вид структуры подзапроса.)

Условия поиска (в том числе объединения), относящиеся к другому запросу, также могут встречаться в предложении `WHERE` внешнего запроса — до или после внутреннего запроса.

```

SELECT [DISTINCT]
FROM список_таблиц ← Начало внешнего
                     оператора SELECT
WHERE
{выражение { [NOT] IN | оператор_сравнения {ANY | ALL} }
 | [NOT] EXISTS}
(SELECT [DISTINCT] список_выбора_подзапроса } Подзапрос,
 FROM список_таблиц } заключенный в скобки
 WHERE условия)

[GROUP BY список_группировки
 [HAVING условия]] ← Необязательные части
[ORDER BY порядок]  внешнего оператора SELECT
  
```

Рис. 8.1. Подзапрос в операторе `SELECT`

КАК РАБОТАЮТ ПОДЗАПРОСЫ

Подзапросы возвращают результаты внутреннего запроса во внешнее предложение и имеют две основные формы: **некоррелированную (noncorrelated)** и **коррелированную (correlated)**. Первая реализуется (концептуально) “изнутри наружу”, т.е. внешний запрос выполняет то или иное действие, основываясь на результатах выполнения внутреннего запроса. Вторую (коррелированную) форму подзапроса можно представлять себе как обратное действие: внешний оператор SQL предоставляет значения для внутреннего подзапроса, которые будут использоваться при его выполнении. Затем результаты выполнения подзапроса возвращаются на внешний запрос. Как коррелированные, так и некоррелированные подзапросы бывают трех типов (подробнее об этом несколько позже), в зависимости от элементов в предложении WHERE внешнего запроса.

1. Подзапросы, которые не возвращают ни одного или возвращают несколько элементов (начинаются с IN или с оператора сравнения, содержат ключевые слова ANY или ALL).
2. Подзапросы, которые возвращают единственное значение (начинаются с простого оператора сравнения).
3. Подзапросы, которые представляют собой тест на существование (начинаются с EXISTS).

На рис. 8.2 сравниваются некоррелированный и коррелированный подзапросы с идентичными получаемыми результатами. Оба они начинаются с IN и отыскивают названия издательств, выпускающих книги по бизнесу.

SQL (некоррелированный):

```
select pub_name
from publishers
where pub_id in
  (select pub_id
   from titles
   where type = 'business')
```

Внутренний запрос выполняется независимо, передавая результаты во внешний запрос

SQL (коррелированный):

```
select pub_name
from publishers p
where 'business' in
  (select type
   from titles
   where pub_id = p.pub_id)
```

Внутренний запрос для своего выполнения должен получить данные из внешнего запроса

Результат:

Pub_name

New Age Books

Algodata Infosystems

Рис. 8.2. Некоррелированный и коррелированный подзапросы

Как это обычно имеет место в подзапросах, этот запрос можно также сформулировать как запрос на объединение. Чтобы названия каждого издательства отображались только один раз, добавьте в список выбора ключевое слово DISTINCT.

SQL:

```
select distinct pub_name
from publishers p, titles t
where p.pub_id = t.pub_id
and type = 'business'
```

Некоррелированная обработка

С концептуальной точки зрения внешний запрос и некоррелированный подзапрос (или внутренний запрос) реализуются за два шага. Сначала внутренний запрос возвращает идентификационные номера тех издательств, которые опубликовали книги по бизнесу (1389 и 0736):

SQL:

```
select pub_id
from titles
where type = 'business'
```

Результат:

```
pub_id
-----
1389
1389
0736
1389
```

Затем эти значения передаются на внешний запрос, который отыскивает названия издательств, соответствующие указанным идентификационным номерам в таблице *publishers*:

SQL:

```
select pub_name
from publishers
where pub_id in ('1389', '0736')
```

Результат:

```
pub_name
-----
New Age Books
Algodata Infosystems
```

Определение имен столбцов. Принадлежность столбцов в подзапросах таблицам неявно задается в их предложениях FROM. Это означает, что столбец *pub_id* в предложении WHERE внешнего запроса принадлежит таблице из предложения FROM внешнего запроса — т.е. таблице *publishers*. Столбец *pub_id* в списке выбора подзапроса принадлежит таблице, указанной в предложении FROM этого подзапроса — т.е. таблице *titles*.

Вот как выглядел бы этот запрос в случае явного указания принадлежности столбцов таблицам:

SQL:

```
select pub_name
from publishers
where publishers.pub_id in
(select titles.pub_id
 from titles
 where type = 'business')
```

Вы никогда не ошибетесь, указав явно имя таблицы. Кроме того, с помощью явных определений вы всегда можете отменить неявно выраженные предположения относительно имен таблиц.

Коррелированная обработка

Коррелированные запросы хотя и не столь элегантны, зато весьма эффективны (с их помощью можно решать проблемы, не имеющие простых решений, если пользоваться объединениями или некоррелированными запросами). Пока что вам достаточно будет понять синтаксис этих запросов и получить общее представление о том, как они работают. Поэкспериментировав немного с этими запросами, вы почувствуете себя значительно увереннее.

В коррелированном подзапросе внутренний запрос не может быть реализован немедленно: он ссылается на внешний запрос и выполняется поочередно для каждой строки во внешнем запросе. В примере, представленном на рис. 8.2, внешняя таблица (*publishers*) имеет три строки, поэтому внутренний запрос будет выполняться трижды.

С концептуальной точки зрения обработка включает следующие этапы. Внешний запрос отыскивает первое имя в таблице *publishers* (допустим, New Age Books). Чтобы найти требуемые строки в таблице *titles*, внутренний запрос объединяет соответствующий *publishers.pub_id* (0736) с *titles.pub_id* (в результате находится шесть строк). Затем внутренний запрос возвращает этот результат в предложение *IN* внешнего запроса, где *titles.type* сравнивается со строкой “business”. Только одна книга из шести является книгой по бизнесу, но этого достаточно: New Age Books удовлетворяет требуемым критериям. Затем снова начинает работать подзапрос, на этот раз пользуясь *pub_id* второй строки из таблицы *publishers* (0877). Он отыскивает еще шесть строк, причем все они относятся к различным типам книг по кулинарии; это издательство (Binnet & Hardley) не удовлетворяет указанным нами критериям. Подзапрос выполняется в третий раз с *publishers.pub_id*, равным 1389 (Algodata Infosystems), и находит еще шесть строк. Три из них относятся к типу “business”. В результирующий список также попадет и Algodata Infosystems.

Определение имен столбцов. Коррелированные запросы требуют явного указания имен столбцов из внешнего запроса (можно пользоваться псевдонимами, например *p.pub_id* вместо *publishers.pub_id* на рис. 8.2). Принадлежность таблицам столбцов во внутреннем запросе задается неявно.

Однако вы в любом случае можете указать обе таблицы:

SQL (коррелируемый):

```
select pub_name
from publishers
where 'business' in
(select type
 from titles
 where titles.pub_id = publishers.pub_id)
```

ОБЪЕДИНЕНИЯ ИЛИ ПОДЗАПРОСЫ?

В многотабличных запросах SQL можно использовать как объединения, так и подзапросы. В каких случаях лучше использовать подзапросы, а в каких объединения? Одни пользователи предпочитают всегда использовать объединения, другие — подзапросы.

Однако каждый из этих подходов имеет преимущества и недостатки.

Подзапросы!

Рассмотрим, например, задачу составления перечня всех книг с ценами, равными минимальной цене книги. С помощью объединений эту задачу можно было бы выполнить в два приема.

1. Найти минимальную цену.

SQL:

```
select min(price)
from titles
```

Результат:

2.99

2. Получить названия всех книг, продаваемых по этой цене.

SQL:

```
select title, price
from titles
where price = 2.99
```

Результат:

title	price
You Can Combat Computer Stress!	2.99
The Gourmet Microwave	2.99

Если сделать это с помощью подзапроса, то вам потребуется только один оператор:

SQL:

```
select title, price
from titles
where price =
    (select min(price)
     from titles)
```

Результат:

title	price
You Can Combat Computer Stress!	2.99
The Gourmet Microwave	2.99

Способность вычислять значение агрегирующей функции “на лету” и возвращать его во внешний запрос для сравнения относится к преимуществам подзапросов; объединение не справляется с этой задачей.

Подзапрос:	Объединение:
<pre>select pub_name from publishers where city in (select city from authors)</pre>	<pre>select distinct pub_name from publishers, authors where publishers.city = authors.city</pre>
Результат	
pub_name	
Algodata Infosystems	

Рис. 8.3. Сравнение объединения и подзапроса

Объединения!

Но и у объединений есть свои сильные стороны. Например, два запроса на рис. 8.3 (оба выполняют поиск названий издательств, находящихся в том же городе, что и какой-то из авторов) возвращают идентичные результаты.

Однако объединение предоставляет больше возможностей, поскольку в него можно включить результаты из обеих таблиц. Подзапрос не допускает такой возможности.

SQL:

```
select pub_name, au_fname, au_lname
from publishers, authors
where publishers.city =
authors.city
```

Результат:

pub_name	au_fname	au_lname
Algodata Infosystems	Cheryl	Carson
Algodata Infosystems	Abraham	Bennet

Подзапрос может отображать информацию только из внешней таблицы (*publishers*), поэтому он находит название издательства и останавливается на этом. Если же вы хотите, чтобы результаты включали информацию из обеих таблиц, пользуйтесь объединением.

Подзапросы или самообъединения?

Многие операторы, в которых подзапрос и внешний запрос ссылаются на одну и ту же таблицу, можно реализовать с помощью самообъединения. Используя подзапрос, можно было бы, например, отыскать авторов, проживающих в том же городе, что и Livia Karsen:

SQL:

```
select au_lname, au_fname, city
from authors
where city in
(select city
 from authors
 where au_fname = 'Livia'
 and au_lname = 'Karsen')
```

Результат:

au_lname	au_fname	city
Green	Marjorie	Oakland
Stringer	Dirk	Oakland
Straight	Dick	Oakland
Karsen	Livia	Oakland
MacFeather	Stearns	Oakland

А можно было бы воспользоваться и самообъединением:

SQL:

```
select aul.au_lname, aul.au_fname, aul.city
from authors aul, authors au2
where aul.city = au2.city
and au2.au_lname = 'Karsen'
and au2.au_fname = 'Livia'
```

Что лучше?

Окончательное решение о том, каким же именно средством воспользоваться, — подзапросом или объединением, — когда приходится работать с несколькими таблицами, обычно определяется стилем, в котором вы предпочитаете работать. Большинство людей находят одно из этих средств “более интуитивным”. (Программисты, например, часто предпочитают подзапросы, поскольку они напоминают им столь привычные подпрограммы.) Тем не менее бывают случаи, когда приходится выбирать.

- Подзапросы — когда вам необходимо сравнивать значения агрегирующих функций с другими значениями.
- Объединения — когда вы отображаете информацию из нескольких таблиц.

ПРАВИЛА ПОДЗАПРОСОВ

Теперь, когда вы получили общее представление о некоррелированных и коррелированных подзапросах и сравнили их с объединениями, рассмотрим более детально правила, по которым “живут” подзапросы. Эти правила в основном относятся к списку выбора подзапроса с некоторыми дополнительными ограничениями на функции, которые можно использовать в подзапросе. В вашем SQL может быть больше или меньше тех или иных ограничений, здесь же будут приведены самые распространенные из них.

- Список выбора внутреннего подзапроса, начинающийся с оператора сравнения или IN, может включать только одно выражение или имя столбца. Столбец, имя которого вы указываете в предложении WHERE внешнего оператора, должен быть совместимым для объединения со столбцом, имя которого вы указываете в списке выбора подзапроса.
- Список выбора подзапроса, начинающийся с EXISTS, почти всегда включает “звездочку” (*). В данном случае нет необходимости указывать имена столбцов, поскольку вы лишь выполняете проверку на существование (или “несуществование”) любых строк, которые удовлетворяют указанным критериям. (Вы можете задать принадлежность строк таблицам в предложении WHERE подзапроса.) В остальных отношениях правила списка выбора для подзапроса, начинающегося с EXISTS, идентичны правилам для стандартного списка выбора.
- Подзапросы, начинающиеся с немодифицированного оператора сравнения (оператор сравнения, за которым не следует ключевое слово ANY или ALL), не могут включать предложения GROUP BY и HAVING, если только вы не определили заранее, что в результате группировки будет возвращаться единственное значение.
- Подзапросы не могут манипулировать своими результатами внутри себя, т.е. подзапрос не может включать предложение ORDER BY или ключевое слово INTO.

Обзор трех основных типов подзапросов позволит прояснить смысл этих ограничений и причины их появления. Как уже упоминалось выше, можно выделить следующие типы подзапросов.

1. Подзапросы, которые возвращают от нуля до нескольких элементов (начинаются с IN или с оператора сравнения, модифицируемого с помощью ANY или ALL).
2. Подзапросы, которые возвращают единственное значение (начинаются с немодифицированного оператора сравнения).
3. Подзапросы, которые реализуют тест на существование (начинаются с EXISTS).

Каждый тип описывается в одном из последующих разделов. (Эти запросы могут быть некоррелированными или коррелированными.)

ПОДЗАПРОСЫ, НЕ ВОЗВРАЩАЮЩИЕ ЗНАЧЕНИЙ ИЛИ ВОЗВРАЩАЮЩИЕ НЕСКОЛЬКО ЗНАЧЕНИЙ

Эта группа включает подзапросы, начинающиеся с IN, NOT IN или оператора сравнения с ключевыми словами ANY или ALL.

Подзапросы, начинающиеся с IN

Подзапросы, начинающиеся с ключевого слова IN, имеют следующую общую форму:

```
Начало операторов SELECT, INSERT, UPDATE, DELETE или подзапроса  
WHERE выражение [NOT] IN (подзапрос)  
/Конец операторов SELECT, INSERT, UPDATE, DELETE или подзапроса/
```

Результатом внутреннего подзапроса является список, включающий от нуля до нескольких значений. После того как подзапрос возвратит результаты, к их обработке приступит внешний запрос.

Ниже приведен пример оператора, который можно сформулировать либо с помощью подзапроса, либо с помощью объединения. Если этот запрос изложить по-русски, он будет выглядеть так: “Найти фамилии всех вторых авторов, проживающих в Калифорнии и получающих менее 30 процентов авторского гонорара за книги, соавторами которых они являются”. Если воспользоваться подзапросом, эта фраза примет следующий вид:

SQL:

```
select au_lname, au_fname  
from authors  
where state = 'CA'  
and au_id in  
  (select au_id  
   from titleauthors  
   where royaltyshare < .30  
   and au_ord =2)
```

Результат:

```
----- au_fname -----  
MacFeather      Stearns
```

Сначала выполняется внутренний запрос, генерирующий список идентификаторов, состоящий из двух вторых авторов, удовлетворяющих заданному нами критерию поиска (получение менее 30 процентов авторского гонорара). После этого система выполняет внешний запрос.

Обратите внимание на допустимость включения нескольких условий в предложение WHERE как внутреннего, так и внешнего запроса. Подзапрос может даже включать объединение — объединения и подзапросы никоим образом не являются взаимоисключающими элементами.

Если воспользоваться объединением, запрос можно выразить так:

SQL:

```
select au_lname, au_fname  
from authors, titleauthors  
where state = 'CA'  
and authors.au_id = titleauthors.au_id  
and royaltyshare < .30  
and au_ord = 2
```

Результат:

au_lname	au_fname
MacFeather	Stearns

“Какие авторы являются единственными авторами, а какие — соавторами?” — вот вопрос, который легче прочитать и понять в виде подзапроса (хотя ту же информацию можно получить и с помощью сложного объединения).

SQL:

```
select authors.au_id, au_lname, au_fname
from authors, titleauthors
where royaltyshare < 1.0
and authors.au_id = titleauthors.au_id
and authors.au_id in
    (select distinct authors.au_id
     from authors, titleauthors
     where titleauthors.royaltyshare = 1.0
     and authors.au_id = titleauthors.au_id)
```

Результат:

au_id	au_lname	au_fname
213-46-8915	Green	Marjorie
998-72-3567	Ringer	Albert

Сначала внутренний запрос выбирает идентификационные номера авторов, доля авторского гонорара которых равна 100 процентам, а затем внешний запрос сравнивает эти идентификаторы со своей выборкой авторов, доля авторского гонорара которых составляет менее 100 процентов.

Ниже приведен пример объединения, которое отыскивает ту же информацию. Список выбора в этом случае составлен несколько иначе, что позволяет отображать оба значения гонорара (что не удастся при использовании подзапроса, когда можно отображать только одно такое значение):

SQL:

```
select al.au_id, au_lname, tal.royaltyshare, ta2.royaltyshare
from authors al, titleauthors tal, titleauthors ta2
where tal.royaltyshare < 1.0
and al.au_id = tal.au_id
and al.au_id = ta2.au_id
and ta2.royaltyshare = 1.0
```

Результат:

au_id	au_lname	tal.royaltyshare	ta2.royaltyshare
213-46-8915	Green	0.4	1
998-72-3567	Ringer	0.5	1

Подзапросы, начинающиеся с NOT IN

Подзапросы, начинающиеся с NOT IN, также возвращают список, включающий от нуля до нескольких значений. Следующий запрос отыскивает названия издательств, которые не опубликовали книги по бизнесу (пример, обратный приведенному выше):

SQL:

```
select distinct pub_name
from publishers
```

```

where pub_id not in
  (select pub_id
   from titles
   where type = 'business')

```

Результат:

```

pub_name
-----
Binnet & Hardley

```

Этот запрос ничем не отличается от запроса, приведенного выше, за исключением того, что вместо IN в нем используется конструкция NOT IN. Однако этот оператор NOT IN нельзя преобразовать в “не равное” объединение. Аналогичное “не равное” объединение имеет иной смысл: оно отыскивает названия издательств, которые опубликовали *какую-то* книгу, которая не является книгой по бизнесу.

SQL:

```

select distinct pub_name
from publishers, titles
  where publishers.pub_id = titles.pub_id
  and type != 'business'

```

Результат:

```

pub_name
-----
Algodata Infosystems
Binnet & Hardley
New Age Books

```

Если вам требуется восстановить в своей памяти сведения об объединениях, основанных на неравенствах, обратитесь к главе 7.

Коррелированные подзапросы с IN

Фамилии всех авторов, получающих 100 процентов авторского гонорара, можно найти с помощью следующего оператора:

SQL:

```

select distinct au_lname, au_fname
from authors
where 1.00 in
  (select royaltyshare
   from titleauthors
   where au_id = authors.au_id)

```

Результат:

au_lname	au_fname
Blotchet-Halls	Reginald
Carson	Cheryl
del Castillo	Innes
Green	Marjorie
Locksley	Chastity
Panteley	Sylvia
Ringer	Albert
Straight	Dick
White	Johnson

В отличие от большинства предыдущих примеров подзапросов, вы не можете выполнить подзапрос этого оператора независимо от внешнего запроса. Значение, которое требуется этому подзапросу для *authors.au_id*, является *переменной*: оно изменяется по мере того, как система анализирует разные строки таблицы *authors*.

Вот как, с концептуальной точки зрения, система обрабатывает этот запрос. Она проверяет каждую строку таблицы *authors* на соответствие указанному условию. Допустим, что система сначала анализирует строку для Cheryl Carson. Переменная *authors.au_id* принимает значение “238-95-7766”, которое система подставляет во внутренний запрос:

```
SQL:
select royaltyshare
from titleauthors
where au_id = '238-95-7766'
```

Результатом будет 1.00, поэтому внешний запрос примет следующий вид:

```
SQL:
select au_lname, au_fname
from authors
where 1.00 in (1.00)
```

Поскольку это соответствует истине, строка для Cheryl Carson будет включена в результаты. Если ту же процедуру выполнить со строкой для Abraham Bennet, то вы увидите, что эта строка не “вписывается” в указанные условия.

Коррелированные IN-подзапросы к одной таблице. Коррелированный подзапрос к одной таблице, начинающийся с IN, можно использовать, например, для того, чтобы выяснить, какие типы книг являются общими для нескольких издательств:

```
SQL:
select distinct t1.type
from titles t1
where t1.type in
      (select t2.type
       from titles t2
        where t1.pub_id != t2.pub_id)
```

Результат:

```
type
-----
business
```

Чтобы различать две разные роли, которые выполняет таблица *titles*, здесь требуется применение псевдонимов. Этот вложенный запрос эквивалентен оператору самообъединения:

```
SQL:
select distinct t1.type
from titles t1, titles t2
where t1.type = t2.type
and t1.pub_id != t2.pub_id
```

Коррелированные IN-подзапросы в предложении HAVING. Коррелированный подзапрос можно также использовать в предложении HAVING. Такой вариант запроса можно, например, использовать для нахождения типов книг, максимальный аванс за которые, по меньшей мере, в два раза превышает величину среднего аванса для этого типа книг.

```
SQL:
select t1.type
from titles t1
```

```
group by t1.type
having max(t1.advance) in
    (select 2 * avg(t2.advance)
 from titles t2
  where t1.type = t2.type)
```

Результат:

```
type
-----
mod_cook
```

В этом случае подзапрос выполняется по одному разу для каждой группы, определенной во внешнем запросе, — т.е. по одному разу для каждого типа книг.

Подзапросы, начинающиеся с операторов сравнения и включающие ключевые слова ANY или ALL

В другом виде подзапросов, которые не возвращают или возвращают несколько строк, используется оператор сравнения, модифицированный ключевыми словами ANY или ALL. Подзапросы, начинающиеся с модифицированного оператора сравнения, имеют общую форму следующего вида:

```
Начало операторов SELECT, INSERT, UPDATE, DELETE или подзапроса
WHERE выражение оператор_сравнения [ANY | ALL] (подзапрос)
[Конец операторов SELECT, INSERT, UPDATE, DELETE или подзапроса]
```

Что такое ALL и ANY. Если в качестве примера воспользоваться оператором сравнения “>”, то “> ALL” означает “больше, чем каждое значение” (другими словами, “больше, чем наибольшее значение”). Таким образом, “> ALL (1, 2, 3)” означает “больше, чем 3”. “> ANY” означает “больше, чем, по крайней мере, одно значение” (другими словами, “больше, чем наименьшее значение”). Таким образом, “> ANY (1, 2, 3)” означает “больше, чем 1”. Рис. 8.4 иллюстрирует различия между ключевыми словами ANY и ALL.

Ключевые слова ALL и ANY иногда доставляют пользователям немало хлопот, поскольку компьютеры не терпят неоднозначности, которой эти слова отличаются в английском языке. (Действительно, слову “all” посвящена целая страница убористого текста в *Новом большом англо-русском словаре*, а слову “any” — примерно треть страницы. — *Прим. перев.*)

ALL	Результат	ANY	Результат
>ALL(1, 2, 3)	>3	>ANY(1, 2, 3)	>1
< ALL(1, 2, 3)	<1	< ANY(1, 2, 3)	<3
= ALL(1, 2, 3)	=1 или =2 или =3	=ANY(1, 2, 3)	=1 или =2 или =3

Рис. 8.4. Сравнение ключевых слов ANY и ALL

Подзапросы с ключевым словом ALL. Вы могли бы, например, задать следующий вопрос: “Аванс за какие книги превышает аванс за любую книгу, опубликованную издательством New Age Books?” Этот вопрос можно было бы перефразировать, чтобы прояснить его “перевод” на язык SQL: “Аванс за какие книги превышает наибольший аванс, выплаченный издательством New Age Books?” Ключевое слово ALL (но не ключевое слово ANY!) — именно то, что требуется в данном случае:

```
SQL:
select title
from titles
where advance > all
    (select advance
```

```
from publishers, titles
  where titles.pub_id = publishers.pub_id
  and pub_name = 'New Age Books')
```

Результат:

```
title
```

```
-----
The Gourmet Microwave
```

Для каждого названия внутренний запрос находит список значений авансов, выплаченных издательством New Age Books. Внешний запрос находит наибольшее значение в списке и определяет, не выплачен ли за книгу, которая рассматривается в данный момент, еще больший аванс.

Если внутренний подзапрос, начинающийся с ALL и оператора сравнения, возвращает в качестве одного из своих значений NULL, считается, что запрос в целом завершился неудачно. Например, значения авансов за книги издательства Algodata Infosystems выглядят так:

SQL:

```
select advance
from publishers, titles
where titles.pub_id = publishers.pub_id
  and pub_name = 'Algodata Infosystems'
```

Результат:

```
advance
```

```
-----
8000
```

```
NULL
```

```
5000
```

```
5000
```

```
5000
```

```
7000
```

Если вас интересуют авансы, превышающие любые авансы, выплаченные издательством Algodata Infosystems, вы не получите никаких результатов, поскольку невозможно сказать, что больше значения NULL.

SQL:

```
select title
from titles
where advance > all
  (select advance
   from publishers, titles
   where titles.pub_id = publishers.pub_id
   and pub_name = 'Algodata Infosystems')
```

Результат:

```
title
```

Поэкспериментируйте со своей системой и выясните, что происходит, когда внутренний запрос не возвращает никаких результатов, как в следующем примере — с ложным условием (нет издательства, которое называлось бы “Demo Books”).

SQL:

```
select title
```

```

from titles
where advance > all
  (select advance
   from publishers, titles
   where titles.pub_id = publishers.pub_id
   and pub_name = 'Demo Books')

```

Подзапросы с ключевым словом ANY. Запрос с ключевым словом ANY находит значения, превышающие “некоторое” значение из подзапроса. Приведенный ниже запрос находит книги, за которые был выплачен аванс, превышающий минимальное значение аванса (\$5000), выплаченного издательством Algodata Infosystems.

```

SQL:
select title, advance
from titles
where advance > all
  (select advance
   from titles, publishers
   where titles.pub_id = publishers.pub_id
   and pub_name = 'Algodata Infosystems')

```

Результат:

title	advance
Secrets of Silicon Valley	8000
Sushi, Anyone?	8000
But Is It User Friendly?	7000
You Can Combat Computer Stress!	10125
Life Without Fear	6000
The Gourmet Microwave	15000
Onions, Leeks, and Garlic: Cooking Secrets of the Mediterranean	7000
Computer Phobic and Non-Phobic Individuals: Behavior Variations	7000

Для каждой книги внутренний запрос находит список величин аванса, выплаченных издательством Algodata Infosystems. Внешний запрос просматривает все значения в этом списке и определяет, не выплачен ли за книгу, которая рассматривается в данный момент, аванс больший, чем любая из этих величин.

Если подзапрос не возвращает никаких значений, то считается, что запрос в целом завершился неудачно.

Сравнение ключевых слов IN, ANY и ALL. Оператор “= ANY” полностью эквивалентен оператору IN. Если, например, требуется найти авторов, которые проживают в том же городе, в котором расположено какое-то определенное издательство, можно воспользоваться либо IN, либо = ANY:

```

SQL:
select au_lname, au_fname, city
from authors
where city in
  (select city
   from publishers)
order by city

```

или

```

SQL:
select au_lname, au_fname, city
from authors

```

```

where city = any
      (select city
       from publishers)
order by city

```

Результат:

au_lname	au_fname	city
Carson	Cheryl	Berkeley
Bennet	Abraham	Berkeley

Однако оператор "<> ANY" (или оператор "!= ANY" — это зависит от синтаксиса, принятого в конкретной системе) отличается от оператора NOT IN. "<> ANY" означает "не = a или не = b или не = c". NOT IN означает "не = a и не = b и не = c". Допустим, вы хотите найти авторов, которые проживают в городе, в котором нет ни одного издательства. Для этого можно было бы воспользоваться следующим запросом:

SQL:

```

select au_lname, au_fname
from authors
where city <> any
      (select city
       from publishers)

```

Результат:

au_lname	au_fname
Bennet	Abraham
Green	Marjorie
Carson	Cheryl
Ringer	Albert
Ringer	Anne
DeFrance	Michel
Panteley	Sylvia
McBadden	Heather
Stringer	Dirk
Straight	Dick
Karsen	Livia
MacFeather	Stearns
Dull	Ann
Yokomoto	Akiko
O'Leary	Michael
Gringlesby	Burt
Greene	Morningstar
White	Johnson
del Castillo	Innes
Hunter	Sheryl
Locksley	Chastity
Blotchet-Halls	Reginald
Smith	Meander

Полученные результаты включают всех 23 авторов. Это произошло потому, что каждый автор проживает в каком-то городе, где нет какого-нибудь издательства

(поскольку каждый автор живет в одном и только одном городе). Внутренний запрос находит все города, в которых расположены издательства, а затем внешний запрос — для *каждого* города — находит авторов, которые там не живут. Вот что произойдет, если в этот запрос подставить оператор NOT IN:

SQL:

```
select au_lname, au_fname, city
from authors
where city not in
      (select city
       from publishers)
order by city
```

Результат:

au_lname	au_fname	city
del Castillo	Innes	Ann Arbor
Blotchet-Halls	Reginald	Corvallis
Gringlesby	Burt	Covelo
DeFrance	Michel	Gary
Smith	Meander	Lawrence
White	Johnson	Menlo Park
Greene	Morningstar	Nashville
MacFeather	Stearns	Oakland
Stringer	Dirk	Oakland
Straight	Dick	Oakland
Karsen	Livia	Oakland
Green	Marjorie	Oakland
Dull	Ann	Palo Alto
Hunter	Sheryl	Palo Alto
Panteley	Sylvia	Rockville
Ringer	Anne	Salt Lake City
Ringer	Albert	Salt Lake City
Locksley	Chastity	San Francisco
O'Leary	Michael	San Jose
McBadden	Heather	Vacaville
Yokomoto	Akiko	Walnut Creek

Именно эти результаты вам и требовалось получить. Они включают всех авторов, за исключением Cheryl Carson и Abraham Bennet, которые проживают в Беркли, где расположено издательство Algodata Infosystems.

ПОДЗАПРОСЫ, ВОЗВРАЩАЮЩИЕ ЕДИНСТВЕННОЕ ЗНАЧЕНИЕ

Подзапрос, начинающийся с **немодифицированного оператора сравнения** (оператор сравнения, не сопровождаемый ключевыми словами ANY или ALL), должен возвращать единственное значение. (В противном случае вы получите сообщение об ошибке, и запрос не будет обработан.) Такие подзапросы имеют общую форму следующего вида:

```
Начало операторов SELECT, INSERT, UPDATE, DELETE или подзапроса
WHERE выражение оператор_сравнения (подзапрос)
[Конец операторов SELECT, INSERT, UPDATE, DELETE или подзапроса]
```

В идеале, чтобы воспользоваться этим видом подзапроса, вы должны достаточно хорошо знать свои данные и суть решаемой вами задачи, чтобы быть уверенным в том, что подзапрос возвратит именно одно и только одно значение. Если же вы рассчитываете получить несколько значений, воспользуйтесь ключевым словом IN или модифицированным оператором сравнения.

Если, например, вы предполагаете, что каждое издательство размещается только в одном городе, и хотите найти фамилии авторов, проживающих в городе, где размещается издательство Algodata Infosystems, то можете написать оператор SQL с подзапросом, начинающимся с простого оператора сравнения "=":

SQL:

```
select au_lname, au_fname
from authors
where city =
      (select city
       from publishers
       where pub_name = 'Algodata Infosystems')
```

Результат:

au_lname	au_fname
Bennet	Abraham
Carson	Cheryl

Агрегирующие функции гарантируют единственное значение

Подзапросы с оператором сравнения часто включают агрегирующие функции, поскольку эти функции гарантированно возвращают единственное значение. Например, если требуется найти названия всех книг с ценами выше текущей минимальной цены, надо выполнить следующий запрос:

SQL:

```
select title
from titles
where price >
      (select min(price)
       from titles)
```

Результат:

title
Secrets of Silicon Valley
The Busy Executive's Database Guide
Emotional Security: A New Algorithm
Prolonged Data Deprivation: Four Case Studies
Cooking with Computers: Surreptitious Balance Sheets
Silicon Valley Gastronomic Treats
Sushi, Anyone?
Fifty Years in Buckingham Palace Kitchens
But Is It User Friendly?
Is Anger the Enemy?
Life Without Fear
Onions, Leeks, and Garlic: Cooking Secrets of the Mediterranean
Straight Talk About Computers

Сначала внутренний запрос находит минимальную цену в таблице *titles*, затем внешний запрос использует это значение для выбора соответствующих названий.

Предложения GROUP BY и HAVING должны возвращать единственное значение

Подзапросы с оператором сравнения не могут включать предложения GROUP BY и HAVING, если вы не уверены, что они возвращают единственное значение. Например, следующий запрос находит книги, имеющие цену выше, чем самая дешевая книга в категории *trad_cook*:

```
SQL:
select title, type
from titles
where price >
      (select min(price)
       from titles
       group by type
       having type = 'trad_cook')
```

Результат:

title	type
Secrets of Silicon Valley	popular_comp
The Busy Executive's Database Guide	business
Prolonged Data Deprivation: Four Case Studies	psychology
Silicon Valley Gastronomic Treats	mod_cook
Sushi, Anyone?	trad_cook
But Is It User Friendly?	popular_comp
Onions, Leeks, and Garlic: Cooking Secrets of the Mediterranean	trad_cook
Straight Talk About Computers	business
Computer Phobic and Non-Phobic Individuals: Behavior Variations	psychology

Коррелированные подзапросы с операторами сравнения

Если требуется найти книги, заказанное количество которых меньше среднего объема заказа, надо выполнить следующий запрос:

```
SQL:
select s1.sonum, s1.title_id, s1.qty_ordered
from salesdetails s1
where qty_ordered <
      (select avg(qty_ordered)
       from salesdetails s2
       where s1.title_id = s2.title_id)
order by title_id
```

Результат:

sonum	title_id	qty_ordered
8	BU1032	5

5	MC3021	15
7	PS2091	3
3	PS2091	20
2	PS2091	10

Внешний запрос выбирает строки из таблицы *salesdetails* (т.е. из *s1*) одна за одной. Подзапрос вычисляет среднее количество по каждому рассматриваемому заказу для выбора во внешнем запросе. Для каждого возможного значения *s1* система выполняет подзапрос и включает данную строку в результаты, если соответствующее количество оказывается меньше, чем вычисленное среднее для этой таблицы.

В этом, а также в следующем запросе коррелированный подзапрос имитирует действие оператора GROUP BY. Нет необходимости выполнять группирование по типу в явном виде, поскольку с помощью самообъединения в предложении WHERE данного подзапроса определяются средние цены по каждому типу. Чтобы найти книги, цена которых выше средней для книг этого типа, надо выполнить следующий запрос:

SQL:

```
select t1.type, t1.title
from titles t1
where t1.price >
      (select avg(t2.price)
       from titles t2
       where t1.type = t2.type)
```

Результат:

type	title
business	The Busy Executive's Database Guide
psychology	Prolonged Data Deprivation: Four Case Studies
mod_cook	Silicon Valley Gastronomic Treats
popular_comp	But Is It User Friendly?
Trad_cook	Onions, Leeks, and Garlic: Cooking Secrets of the Mediterranean
business	Straight Talk About Computers
psychology	Computer Phobic and Non-Phobic Individuals: Behavior Variations

Для каждого возможного значения *t1* система выполняет подзапрос и включает соответствующую строку в результаты, если величина цены в этой строке оказывается больше вычисленного среднего значения.

ПОДЗАПРОСЫ, ВЫПОЛНЯЮЩИЕ ПРОВЕРКУ НА СУЩЕСТВОВАНИЕ

Когда тот или иной подзапрос начинается с ключевого слова EXISTS, этот подзапрос функционирует как “тест на существование”. Ключевое слово EXISTS в предложении WHERE выполняет проверку на существование (или несуществование) данных, которые удовлетворяют критериям соответствующего подзапроса.

Подзапрос, который начинается с ключевого слова EXISTS, имеет общую форму следующего вида:

```
Начало операторов SELECT, INSERT, UPDATE, DELETE или подзапроса
WHERE [NOT] EXISTS (подзапрос)
[Конец операторов SELECT, INSERT, UPDATE, DELETE или подзапроса]
```

Если вам требуется найти названия всех издательств, которые публикуют книги по бизнесу, надо выполнить следующий запрос:

SQL:

```
select distinct pub_name
from publishers
where exists
    (select *
     from titles
     where pub_id = publishers.pub_id
     and type = 'business')
```

Результат:

```
pub_name
-----
Algodata Infosystems
New Age Books
```

EXISTS выполняет проверку на наличие или отсутствие “пустого набора” строк. Если подзапрос возвращает хотя бы одну строку, этот результат оценивается как “истина”. Это означает, что результат выполнения EXISTS будет успешным, а результат выполнения NOT EXISTS будет неудачным. Если подзапрос возвращает пустой набор (строк нет), этот результат оценивается как “ложь”. Это означает, что результат выполнения фразы NOT EXISTS будет успешным, а результат выполнения фразы EXISTS будет неудачным.

В этом случае первым названием издательства будет Algodata Infosystems (с идентификационным номером 1389). Проходит ли Algodata Infosystems тест на существование? Другими словами, есть ли какие-то строки в таблице *titles*, в которых *pub_id* равен 1389, а *type* соответствует бизнесу? Если есть, то “Algodata Infosystems” должно быть одним из выбранных значений. Тот же процесс повторяется для каждого из названий остальных издательств.

Обратите внимание, что синтаксис подзапросов, начинающихся с EXISTS, отличается от синтаксиса других подзапросов в следующих отношениях.

- Ключевому слову EXISTS не предшествует имя столбца, константа или какое-то другое выражение.
- Список выбора подзапроса, начинающегося с EXISTS, почти всегда состоит из “звездочки” (*). Нет никакого смысла вводить имена столбцов, поскольку вы лишь выполняете тест на существование строк, которые удовлетворяют условиям подзапроса, а они указываются в предложении WHERE этого подзапроса (а не в предложении SELECT этого подзапроса).

Ключевое слово EXISTS — чрезвычайно важный элемент, поскольку зачастую отсутствует альтернатива использованию подзапроса. На практике EXISTS-подзапрос почти всегда является коррелированным подзапросом. Вместо использования внешнего запроса для обработки значений, поставляемых внутренним запросом, внешний запрос можно использовать для выработки одного за другим значений, которые будут тестироваться внутренним запросом.

SQL:

```
select title
from titles
where pub_id in
    (select pub_id
     from publishers
     where city like 'B%')
```

SQL:

```
select title
from titles
where pub_id in
    (select *
     from publishers
     where pub_id = titles.pub_id
     and city like 'B%')
```

Рис. 8.5. Сравнение подзапросов с IN и EXISTS

Ниже приведено несколько примеров операторов, использующих EXISTS, и эквивалентные им альтернативы.

На рис. 8.5 показаны два запроса, которые находят названия книг, опубликованных любым издательством, расположенном в городе, название которого начинается с буквы *B*.

Оба запроса выдают одинаковые результаты. Из этих результатов следует, что двенадцать книг в таблице *titles* опубликованы издательствами, расположенными либо в Бостоне (Boston), либо в Беркли (Berkeley).

Результат:

title

```
-----  
Secrets of Silicon Valley  
The Busy Executive's Database Guide  
Emotional Security: A New Algorithm  
Prolonged Data Deprivation: Four Case Studies  
Cooking with Computers: Surreptitious Balance Sheets  
Net Etiquette  
You Can Combat Computer Stress!  
But Is It User Friendly?  
Is Anger the Enemy?  
Life Without Fear  
Straight Talk About Computers  
Computer Phobic and Non-Phobic Individuals: Behavior Variations
```

NOT EXISTS отыскивает пустой набор

NOT EXISTS выполняет действие, обратное EXISTS. Успешный результат выполнения запросов с NOT EXISTS соответствует случаю, когда подзапрос вообще не возвращает строк.

Чтобы найти, например, названия издательств, которые не публикуют книги по бизнесу, запрос должен иметь следующий вид:

SQL:

```
select pub_name  
from publishers  
where not exists  
      (select *  
       from titles  
       where pub_id = publishers.pub_id  
       and type = 'business')
```

Результат:

```
pub_name  
-----  
Binnet & Hardley
```

Следующий запрос позволяет найти названия книг, ни одна из которых еще не была продана:

SQL:

```
select title  
from titles  
where not exists  
      (select title_id
```

```
from salesdetails
where title_id = titles.title_id)
```

Результат:

```
title
-----
The Psychology of Computer Cooking
Net Etiquette
```

Использование EXISTS для поиска пересечения и разности

Подзапросы, начинающиеся с EXISTS и NOT EXISTS, можно использовать для выполнения двух операций из теории множеств: **пересечения** и **разности**. Пересечение двух множеств содержит все элементы, которые принадлежат обоим исходным множествам. Разность двух множеств содержит элементы, которые принадлежат только первому из двух множеств.

Пересечением *authors* и *publishers* по столбцу *city* является множество городов, в которых проживает какой-то автор и расположено какое-то издательство:

SQL:

```
select distinct city
from authors
where exists
    (select *
     from publishers
     where authors.city = publishers.city)
```

Результат:

```
city
-----
Berkeley
```

Разностью между *authors* и *publishers* по столбцу *city* является множество городов, в которых проживает какой-то автор, но нет никакого издательства (т.е. все города, за исключением Беркли):

SQL:

```
select distinct city
from authors
where not exists
    (select *
     from publishers
     where authors.city = publishers.city)
```

Результат:

```
city
-----
Ann Arbor
Corvallis
Covelo
Gary
Lawrence
Menlo Park
Nashville
Oakland
Palo Alto
```

Rockville
Salt Lake City
San Francisco
San Jose
Vacaville
Walnut Creek

ПОДЗАПРОСЫ С РАЗНЫМИ УРОВНЯМИ ВЛОЖЕНИЯ

Подзапрос сам может включать один или несколько других подзапросов. Вы можете организовать вложенную структуру, включающую любое число подзапросов.

Примером задачи, которая может быть решена с помощью оператора с несколькими уровнями вложенных запросов, является следующая: “Найти фамилии авторов, участвовавших в написании по крайней мере одной книги по компьютерам”.

SQL:

```
select au_lname, au_fname
from authors
where au_id in
      (select au_id
       from titleauthors
       where title_id in
       (select title_id
        from titles
        where type = 'popular_comp' ) )
```

Результат:

au_lname	au_fname
Carson	Cheryl
Dull	Ann
Hunter	Sheryl
Locksley	Chastity

Самый внутренний запрос возвращает идентификационные номера названий PC1035, PC8888 и PC9999. Запрос на следующем по старшинству уровне выполняется на основе этих идентификаторов названий и возвращает идентификационные номера авторов. Наконец, внешний запрос использует найденные идентификаторы авторов для поиска их фамилий.

Этот запрос можно также представить в виде объединения:

SQL:

```
select au_lname, au_fname
from authors, titles, titleauthors
where authors.au_id = titleauthors.au_id
and titles.title_id = titleauthors.title_id
and type = 'popular_comp'
```

ПОДЗАПРОСЫ В ОПЕРАТОРАХ UPDATE, DELETE И INSERT

Подзапросы могут быть вложены в операторы UPDATE, DELETE и INSERT так же, как и в операторы SELECT.

Приведенный ниже запрос увеличивает в два раза цену всех книг, опубликованных издательством New Age Books. Данный оператор обновляет таблицу *titles*; в его подзапросе есть обращение к таблице *publishers*.

SQL:

```
update titles
set price = price * 2
where pub_id in
(select pub_id
 from publishers
 where pub_name = 'New Age Books')
```

Эквивалентный оператор UPDATE, в котором используется объединение (для систем, которые допускают наличие предложения FROM в операторе UPDATE), имеет следующий вид:

SQL:

```
update titles
set price = price * 2
from titles, publishers
where titles.pub_id = publishers.pub_id
and pub_name = 'New Age Books'
```

С помощью следующего вложенного оператора SELECT вы можете удалить все записи, связанные с заказами на продажи книг по бизнесу:

SQL:

```
delete salesdetails
where title_id in
(select title_id
 from titles
 where type = 'business')
```

Эквивалентный оператор DELETE, в котором используется объединение (для систем, которые в операторе DELETE допускают наличие нескольких таблиц в предложении FROM), имеет следующий вид:

SQL:

```
delete salesdetails
 from salesdetails, titles
where salesdetails.title_id = titles.title_id
and type = 'business'
```

В ПОЛЕ ЗРЕНИЯ КУРСОРА

Теперь, когда вы уже умеете создавать как простые, так и довольно сложные запросы с помощью функций, объединений и подзапросов, в вашем распоряжении оказалась почти вся мощь языка SQL. Нам осталось только разобраться, как представить данные в самом удобном для вас виде и как обеспечить целостность и безопасность своей базы данных.

Курсоры, речь о которых пойдет в следующей главе, являются средством просмотра конкретной совокупности данных из одной или нескольких таблиц. Воспользовавшись курсором, вы можете представить такую совокупность данных в виде виртуальной таблицы. Кроме того, курсоры можно использовать как механизм обеспечения безопасности системы (подробнее об этом вы узнаете в главе 10).

Создание и использование виртуальных таблиц (курсоров)

КУРСОР ОБЕСПЕЧИВАЕТ ГИБКОСТЬ

Подобно операции объединения, курсоры являются неотъемлемой частью реляционной модели. Курсор создается с помощью оператора SELECT и обеспечивает гибкость при анализе и обработке данных. Курсор можно представить себе как подвижный кадр или окно, через которое пользователь видит данные.

В предыдущих главах было продемонстрировано, как пользоваться оператором SELECT для выбора строк, комбинирования таблиц, переименования столбцов и выполнения вычислений, чтобы получить нужную вам информацию в конкретном виде. Создание курсора с помощью оператора SELECT позволяет вам без труда анализировать и обрабатывать именно те данные, которые вам (или кому-то другому) нужны — ни больше ни меньше.

Курсоры — это не какие-то отдельные копии данных из таблицы (таблиц) или другого курсора (курсоров), из которых они получены. Курсоры часто называют виртуальными таблицами, поскольку они не существуют как независимые объекты в базе данных (как, например, “настоящие” таблицы). (Термину “курсор” в ANSI соответствует термин **просматриваемая таблица (viewed table)**, а “настоящая” таблица базы данных называется **базовой таблицей (base table)**.) Запросы к курсорам в основном выполняются так же, как и к таблицам. Однако модификация данных посредством курсоров носит ограниченный характер.

Курсор определяется в операторе SELECT. Когда пользователь обращается к тому или иному курсору, система баз данных ассоциирует с ним соответствующие данные. Курсор находится на вершине этого процесса, скрывая от пользователя все его технические подробности. Прелесть курсора заключается в его простоте и универсальности: начинающих пользователей не пугают всевозможные объединения, опытные же пользователи не испытывают искушения манипулировать с данными, до которых им, вообще говоря, нет никакого дела, а нетерпеливым пользователям не приходится замедлять свою работу из-за необходимости ввода длинных операторов SQL.

СОЗДАНИЕ КУРСОРОВ

Вот упрощенный синтаксис оператора, определяющего курсор:

```
CREATE VIEW имя_курсора [(имя_столбца [, имя_столбца]...)]
AS
SELECT_оператор
```

В следующем примере создается курсор, который отображает фамилии авторов, живущих в Окланде и Калифорнии, а также названия их книг:

```
SQL:
create view oaklanders
as
select au_fname, au_lname, title
```

```
from authors, titles, titleauthors
where authors.au_id = titleauthors.au_id
and titles.title_id = titleauthors.title_id
and city = 'Oakland'
```

SQL:

```
select *
from oaklanders
```

Результат:

au_fname	au_lname	title
Marjorie	Green	The Busy Executive's Database Guide
Marjorie	Green	You Can Combat Computer Stress!
Dick	Straight	Straight Talk About Computers
Livia	Karsen	Computer Phobic and Non-Phobic Individuals: Behavior Variations
Stearns	MacFeather	Computer Phobic and Non-Phobic Individuals: Behavior Variations
Stearns	MacFeather	Cooking with Computers: Surreptitious Balance Sheets

Присваивая курсору имя, обязательно соблюдайте правила создания идентификаторов, предусмотренные в вашей системе. Первая строка оператора CREATE VIEW присваивает курсору имя, а последующий оператор SELECT определяет его. Как вы уже видели, вовсе не обязательно, чтобы оператор SELECT выполнял простой выбор строк и столбцов в одной конкретной таблице. Курсор можно создать на основе нескольких таблиц, других курсоров или того и другого; при этом оператор SELECT может иметь почти любую сложность, а для определения столбцов и строк, которые требуется включить в курсор, используются операции проектирования и выбора.

Удаление курсоров

Во многих версиях SQL имеется команда для удаления курсоров. Ее синтаксис имеет примерно следующий вид:

```
DROP VIEW имя_курсора
```

Если какой-то курсор зависит от таблицы (или другого курсора), которая была удалена, вы не сможете им воспользоваться. Однако если вы создадите под тем же именем новую таблицу (или курсор), которая заменит соответствующий удаленный объект, вы снова сможете пользоваться этим курсором (разумеется, если существуют столбцы, ссылка на которые содержится в его определении). Более подробную информацию по этому вопросу вы найдете в справочных руководствах по своей системе.

ПРЕИМУЩЕСТВА КУРСОРОВ

Чтобы показать преимущества использования курсоров, предположим, что у базы данных *bookbiz* есть несколько пользователей с разными профессиональными интересами. Допустим, что руководитель отдела рекламы хочет знать, какие авторы с какими книгами связаны и кто из них указан на обложке первым, кто вторым и кто третьим. Цены, объемы продаж, затраты, авторские гонорары и личные адреса авторов руководителя отдела рекламы не интересуют, но ему нужна определенная информация из каждой следующих трех таблиц: *titles*, *authors* и *titleauthors*. Если не пользоваться курсором, то для этого можно было бы применить запрос примерно следующего вида:

SQL:

```
select titles.title_id, au_ord, au_lname, au_fname
from authors, titles, titleauthors
where authors.au_id = titleauthors.au_id and
      titles.title_id = titleauthors.title_id
```

Для выполнения этого запроса нам пришлось бы вводить довольно много информации с клавиатуры, и не исключено, что в процессе этого ввода мы сделали бы не одну ошибку. Помимо этого, от нас потребовалось бы проявить определенные познания в области баз данных. Создание курсора под названием *books*, базирующегося на следующем операторе SELECT, облегчило бы использование этой конкретной совокупности данных. Вот оператор, с помощью которого можно создать такой курсор:

SQL:

```
create view books
as
select titles.title_id, au_ord, au_lname, au_fname
from authors, titles, titleauthors
where authors.au_id = titleauthors.au_id and
      titles.title_id = titleauthors.title_id
```

Теперь руководитель отдела рекламы может воспользоваться этим курсором для получения тех же результатов, не задумываясь об объединениях, списках выбора и условиях поиска:

SQL:

```
select *
from books
```

Результат:

title_id	au_ord	au_lname	au_fname
BU1032	1	Bennet	Abraham
PS7777	1	Locksley	Chastity
PC9999	1	Locksley	Chastity
MC2222	1	del Castillo	Innes
PS3333	1	White	Johnson
BU1032	2	Green	Marjorie
PC1035	1	Carson	Cheryl
BU2075	1	Green	Marjorie
PS2091	1	Ringer	Albert
PS2091	2	Ringer	Anne
PS2106	1	Ringer	Albert
MC3021	1	DeFrance	Michel
MC3021	2	Ringer	Anne
TC3218	1	Panteley	Sylvia
BU7832	1	Straight	Dick
PC8888	1	Dull	Ann
PC8888	2	Hunter	Sheryl
PS1372	1	Karsen	Livia
PS1372	2	MacFeather	Stearns
BU1111	1	MacFeather	Stearns

BU1111	2	O'Leary	Michael
TC7777	1	Yokomoto	Akiko
TC7777	2	O'Leary	Michael
TC7777	3	Gringlesby	Burt
TC4203	1	Blotchet-Halls	Reginald

Курсор можно использовать в операторе SELECT так, как если бы он был обычной таблицей. Например, руководителю отдела рекламы может понадобиться упорядочить результаты просмотра в алфавитном порядке (по фамилиям авторов). Например:

```
SQL:
select *
from books
order by au_lname
```

Бухгалтеру может потребоваться другое представление. Его не интересует, кто из авторов является первым, а кто вторым. Допустим, что бухгалтера интересует только нижняя строка: на чье имя должны быть выписаны чеки и на какую сумму. Такой запрос связан с вычислением того, сколько книг было продано и по какой цене, а также какой процент от суммы полагается каждому автору:

```
SQL:
select au_fname, au_lname,
       sum (price*ytd_sales*royalty*royaltyshare) as Total_Income
from authors, titles, titleauthors, roysched
where authors.au_id = titleauthors.au_id
       and titles.title_id = titleauthors.title_id
       and titles.title_id = roysched.title_id
       and ytd_sales between lorange and hirange
group by au_lname, au_fname
```

Если бухгалтер воспользуется указанным ниже оператором SELECT для создания курсора под именем *royaltychecks*, то эквивалентный запрос будет иметь следующий вид:

```
SQL:
select *
from royaltychecks
```

Результаты (кто получает чек и на какую сумму) будут иметь следующий вид:

Результат:

au_lname	au_fname	Total_Income
Bennet	Abraham	4911.54
Yokomoto	Akiko	2455.36
Ringer	Albert	1421.27
Dull	Ann	4095.00
Ringer	Anne	4669.35
Gringlesby	Burt	1841.52
Locksley	Chastity	2665.46
Carson	Cheryl	32240.16
Straight	Dick	8185.91
del Castillo	Innes	4874.36
White	Johnson	8139.93
Karsen	Livia	607.22

Green	Marjorie	13350.54
O'Leary	Michael	3694.25
DeFrance	Michel	9977.33
Blotchet-Halls	Reginald	25255.61
Hunter	Sheryl	4095.00
MacFeather	Stearns	2981.50
Panteley	Sylvia	785.63

Наконец, рассмотрим пример со служащим головной издательской компании, который хочет выяснить положение дел с различными категориями книг в каждом из филиалов. Для этого можно воспользоваться запросом следующего вида:

```
SQL:
select pub_id, type, sum(price*ytd_sales),
       avg(price), avg(ytd_sales)
from titles
group by pub_id, type
```

Однако служащему могут не нравиться все эти сложности. Что ж, можно воспользоваться куда более простым оператором:

```
SQL:
select *
from currentinfo
```

Результат:

PUB#	TYPE	INCOME	AVG_PRICE	AVG_SALES
0736	business	55978.78	2.990	18722.0
0736	psychology	139319.92	13.504	1987.8
0877	NULL	NULL	NULL	NULL
0877	mod_cook	107135.22	11.49	12139.0
0877	trad_cook	249637.50	15.96	6522.0
1389	business	210036.30	17.31	4022.0
1389	popular_comp	283401.00	21.475	6437.5

Воспользовавшись этим курсором, занятой служащий может быстро увидеть, какие издания приносят доход, и установить взаимосвязь между прибылью, средней ценой и средним уровнем продаж.

Почему же все-таки курсор?

Как видно из предыдущих примеров, курсорами можно пользоваться для “фокусировки”, упрощения и “настройки” базы данных под конкретного пользователя. Кроме того, с помощью курсоров реализуется механизм обеспечения безопасности. Наконец, они могут защитить пользователей при изменении структуры базы данных.

Фокусировка, упрощение и настройка. Курсор позволяет руководителю отдела рекламы, бухгалтеру и служащему в приведенных выше примерах сфокусировать свое внимание на конкретных данных и задачах. При этом никакая излишняя или отвлекающая внимание информация ему не встретится.

Упрощается и работа с данными. Если самые необходимые операции объединения, проектирования и/или выбора уже определены с помощью курсоров, существенно упрощается процесс добавления других операторов. Самостоятельное построение такого развернутого запроса, с другой стороны, могло бы представлять собой весьма устрашающую перспективу.

Курсоры — превосходный способ настройки (или подгонки) общей базы данных к нуждам конкретного пользователя, каждый из которых может иметь свои собственные

интересы, потребности и уровни квалификации. Три наших пользователя видят данные по-разному, даже если они одновременно просматривают одни и те же три таблицы.

Безопасность. Курсоры обеспечивают безопасность базы данных, скрывая секретные или не относящиеся к конкретной задаче части базы данных. В системе с правильно установленными полномочиями каждый пользователь имеет доступ только к информации, необходимой ему для выполнения своих профессиональных обязанностей. Подробно об использовании курсоров для обеспечения безопасности речь пойдет в главе 10.

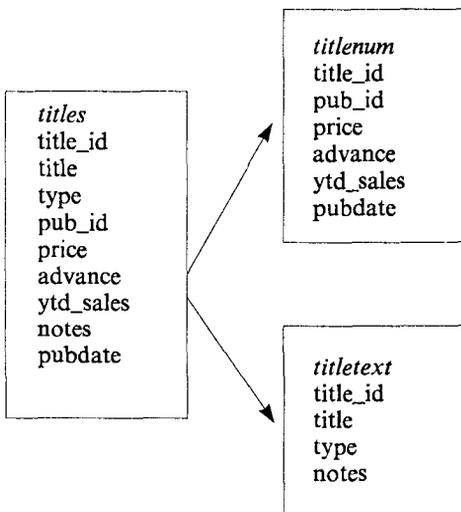


Рис. 9.1. Разбиение таблицы на две

Независимость. Наконец, поговорим о независимости. Время от времени структура базы данных может изменяться. Но от этих изменений в первую очередь не должен страдать пользователь. Предположим, что таблица *titles* была преобразована в две таблицы и затем удалена. Новые таблицы представлены на рис. 9.1.

Отметим, что таблица *titles* может быть восстановлена с помощью объединения новых таблиц по столбцу *title_id*. Чтобы скрыть от пользователей изменения в структуре базы данных, можно создать курсор, объединяющий две новые таблицы (рис. 9.2). Можно даже назвать его *titles* (хотя у нас он называется *titlesview*).

К сожалению, применительно к курсорам существует ряд ограничений на выполнение некоторых операторов модификации данных.

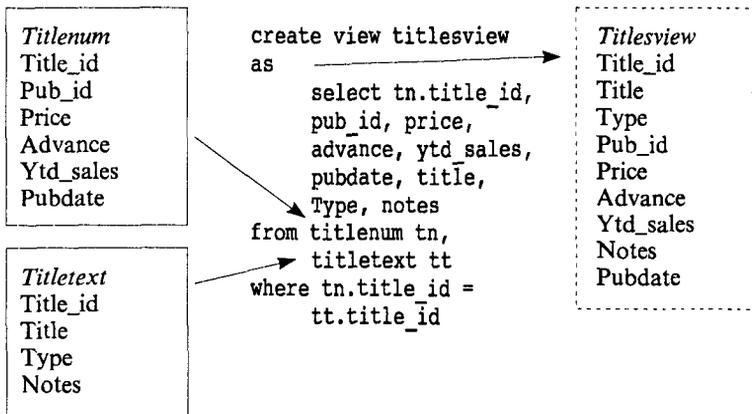


Рис. 9.2. Использование двух базовых таблиц в одном курсоре

КАК РАБОТАЮТ КУРСОРЫ

Что на самом деле происходит в базе данных при создании и использовании курсора?

Создание курсора означает его определение в терминах базовых таблиц. Определение курсора сохраняется в словаре данных без связанных с ним данных. Пользуясь курсорами, вы получаете доступ к связанным с ними таблицам. Другими словами, при создании и использовании курсора не создаются ни какие новые копии данных.

Запрос к курсору выглядит абсолютно аналогично запросу к любой другой таблице базы данных. Однако на изменение данных в курсорах накладываются некоторые ограничения, которые будут рассмотрены далее в этой главе. А пока рассмотрим самый простой случай, когда курсор основан на одной таблице. Изменяя данные в таком курсоре, на самом деле вы изменяете данные в связанной с ним базовой таблице. И наоборот, все изменения в базовой таблице автоматически отражаются в созданном на ее основе курсоре.

Предположим, что вы интересуетесь книгами, стоимость которых больше \$15 и затраты на которые превысили \$5000. Для этого можно просто использовать следующий оператор SELECT:

```
SQL:
select *
from titles
where price > $15
and advance > $5000
```

Предположим теперь, что вам надо выполнить по отношению к этой совокупности данных ряд операций выборки и обновления. Вы могли бы, конечно, скомбинировать условия, отображенные в предыдущем запросе, с любой задаваемой вами командой. Однако для удобства можно создать курсор, в котором будут видны только интересующие вас записи:

```
SQL:
create view hiprice
as
select *
from titles
where price > $15
and advance > $5000
```

Когда SQL получает эту команду, он на самом деле не выполняет оператор SELECT, который следует за ключевым словом AS. Вместо этого он запоминает этот оператор SELECT (который, в сущности, является определением курсора *hiprice*) в словаре данных.

Теперь, когда вы отображаете или выполняете какие-то другие операции по отношению к *hiprice*, SQL комбинирует ваш оператор с запомненным определением *hiprice*. Процедура изменения всех цен в *hiprice*, например, ничем не отличалась бы от внесения изменений в любую другую таблицу.

```
SQL:
update hiprice
set price = price*2
```

В действительности SQL находит определение курсора в словаре данных и преобразует эту команду обновления в следующий оператор:

```
SQL:
update titles
set price = price*2
where price > $15
and advance > $5000
```

Другими словами, из определения курсора SQL знает, что подлежащие обновлению данные находятся в таблице *titles*. Он также знает, что цены следует увеличивать только в тех строках, которые соответствуют условиям для столбцов *price* и *advance*, заданным в определении курсора.

Выполнив оператор обновления (обновление *hiprice*), вы можете увидеть соответствующий результат либо с помощью курсора, либо в таблице *titles*. И наоборот, если бы вы создали курсор, а затем выполнили оператор обновления (который воздействует непосредственно на базовую таблицу), изменившиеся цены также можно было бы увидеть с помощью курсора.

Если вы обновляете таблицу, на основе которой создан курсор, таким образом, что его условиям начинает удовлетворять большее число строк, их можно будет просматривать с помощью этого курсора. Допустим, например, что вы увеличиваете цену книги *You Can Combat Computer Stress* до \$25.95. Поскольку эта книга теперь удовлетворяет “квалификационным условиям” в операторе определения курсора, она становится частью этого курсора.

Правила присвоения имен столбцам курсора

Столбцам курсора можно присваивать псевдонимы. Если вы не задаете имена в операторе CREATE VIEW, столбцы курсора “наследуют” их от столбцов в таблице (таблицах), на основе которой построен данный курсор. Если вы действительно хотите задать новые имена, укажите их в скобках вслед за именем курсора, отделяя их друг от друга запятыми. (В некоторых системах можно также переименовать столбцы в списке выбора.)

Бывают, однако, ситуации, когда новые имена для столбцов курсора являются обязательными. (Если вы переименовываете какой-либо столбец, то должны перечислить все столбцы.)

- Один или несколько столбцов курсора получаются с помощью некоторого арифметического преобразования, встроенной функции или константы.
- Курсору приходится иметь дело с несколькими столбцами под одним и тем же именем (так как разные столбцы в объединяемых таблицах могут иметь одно и то же имя).

Первую ситуацию можно проиллюстрировать оператором CREATE VIEW для курсора с именем *currentinfo*, который мы обсуждали ранее в этой главе.

```
SQL:
create view currentinfo (PUB#, TYPE, INCOME, AVG_PRICE, AVG_SALES)
as
select pub_id, type, sum(price*ytd_sales),
       avg(price), avg(ytd_sales)
from titles
group by pub_id, type
```

Вычисляемые столбцы в списке выбора на самом деле не имеют имен, поэтому вы должны присвоить их в операторе CREATE VIEW. В противном случае вы не сможете сослаться на них. Когда вы работаете с курсором *currentinfo*, всегда пользуйтесь новыми именами, например:

```
SQL:
select PUB#, AVG_SALES
from currentinfo
```

Использование старых имен, таких как *pub_id* или *avg(ytd_sales)*, вам не поможет.

Вторая ситуация, в которой присвоение столбцам новых имен является обязательным, обычно возникает, если объединяемые в операторе SELECT столбцы имеют одинаковые имена. Даже если в операторе SELECT для них указаны разные имена таблиц, для разрешения возникающей неоднозначности вам придется переименовать их:

```
SQL:
create view cities (Author, Authorcity, Pub, Pubcity)
as
select au_lname, authors.city, pub_name, publishers.city
from authors, publishers
where authors.city = publishers.city
```

Конечно, ничто не мешает вам переименовывать столбцы в операторе определения курсора в тех случаях, когда это представляется целесообразным. Однако необходимо помнить, что при переименовании любого столбца в курсоре вам придется перечислить все столбцы: число имен столбцов в скобках должно соответствовать числу элементов в списке выбора.

Независимо от того, переименовываете вы столбец курсора или нет, тип его данных и нулевой статус зависят от того, как он был определен в своей базовой таблице (таблицах).

Создание курсоров с объединениями и подзапросами

В приведенных выше примерах фигурировали курсоры, определенные посредством вычисляемых столбцов и агрегирующих функций. Возможно также определение курсоров с объединениями и подзапросов. Ниже приведен пример определения курсора, которое включает три объединения и один подзапрос. В нем находится идентификатор автора, идентификатор названия, имя издателя и цены на книги, стоимость которых превышает среднеарифметическое значение цен всех книг. (Включение идентификатора автора приводит к тому, что вы увидите несколько строк для книг с несколькими авторами.)

```
SQL:
create view highaverage
as
select authors.au_id, titles.title_id, pub_name, price
from authors, titleauthors, titles, publishers
where authors.au_id = titleauthors.au_id and
      titles.title_id = titleauthors.title_id and
      titles.pub_id = publishers.pub_id and
      price >
      (select avg(price)
       from titles)
```

Теперь, когда создан требуемый курсор, им можно воспользоваться для отображения результатов.

```
SQL:
select *
from highaverage
```

Результат:

au_id	title_id	pub_name	price
172-32-1176	PS3333	New Age Books	19.9900
756-30-7391	PS1372	New Age Books	21.5900
724-80-9391	PS1372	New Age Books	21.5900
712-45-1867	MC2222	Binnet & Hardley	19.9900
672-71-3249	TC7777	Binnet & Hardley	14.9900
267-41-2394	TC7777	Binnet & Hardley	14.9900
472-27-2349	TC7777	Binnet & Hardley	14.9900

807-91-6654	TC3218	Binnet & Hardley	20.9500
427-17-2319	PC8888	Algodata Infosystems	20.0000
846-92-7186	PC8888	Algodata Infosystems	20.0000
409-56-7008	BU1032	Algodata Infosystems	19.9900
213-46-8915	BU1032	Algodata Infosystems	19.9900
238-95-7766	PC1035	Algodata Infosystems	22.9500
274-80-9391	BU7832	Algodata Infosystems	19.9900

Давайте воспользуемся курсором *highaverage*, чтобы проиллюстрировать еще одну разновидность курсоров: курсор, получаемый из другого курсора. Вот как можно создать курсор, с помощью которого отображаются книги, опубликованные издательством Binnet & Hardley, цена которых превышает среднеарифметическое значение цен всех книг.

SQL:

```
create view highBandH
as select *
from highaverage
where pub_name = 'Binnet & Hardley'

select *
from highBandH
```

Результат:

au_id	title_id	pub_name	price
712-45-1867	MC2222	Binnet & Hardley	19.9900
672-71-3249	TC7777	Binnet & Hardley	14.9900
267-41-2394	TC7777	Binnet & Hardley	14.9900
472-27-2349	TC7777	Binnet & Hardley	14.9900
807-91-6654	TC3218	Binnet & Hardley	20.9500

Ограничения на создание курсоров

Несмотря на все разнообразие курсоров, которые можно создавать, всегда есть определенные ограничения, зависящие от конкретного SQL. В принципе, существует два источника ограничений.

- Элементы, использование которых не допускается в операторах CREATE VIEW (ORDER BY и иногда операторы с предложениями UNION, SELECT INTO или COMPUTE). Подробнее об этом см. в справочных руководствах по своей системе.
- Элементы, допускаемые в операторах CREATE VIEW (вычисляемые столбцы, агрегирующие функции), при использовании которых могут возникать проблемы с интерпретацией операторов модификации данных. Проблемы, связанные с обновлением, описаны ниже в этой главе. Если вы хотите, чтобы пользователи имели возможность выполнять большинство функций посредством курсора, ничто не мешает вам модифицировать совершенно “законный” оператор CREATE VIEW, избегая таким образом указанных ограничений.

Предложение Check Option

Одна из проблем, связанных с курсорами, заключается в возможности их некорректного изменения. Рассмотрим, например, курсор, который отображает все книги, цена которых меньше \$5.00. Что произойдет, если одну из цен этих книг

обновить посредством курсора (изменив ее, например, на \$5.99?). Для разрешения такой проблемы предназначено необязательное предложение WITH CHECK OPTION, располагаемое после оператора SELECT:

```
CREATE VIEW имя_курсора [(имя_столбца [, имя_столбца]...)]
AS
SELECT_оператор
[WITH CHECK OPTION]
```

Предложение WITH CHECK OPTION заставляет SQL отвергать любые попытки модификации курсора, приводящие к тому, что одна или несколько строк перестают удовлетворять условиям этого курсора. Другими словами, если оператор модификации данных (UPDATE, INSERT или DELETE) приводит к исчезновению каких-то строк из курсора, такой оператор считается недопустимым.

В качестве примера вернемся к курсору *hiprice*, который включает названия всех книг, цена которых превышает \$15 и величина затрат на которые превышает \$5000:

```
SQL:
create view hiprice
as
select title, price, advance
from titles
where price > $15
and advance > $5000

select *
from hiprice
```

Результат:

title	price	advance
Secrets of Silicon Valley	20.0000	8000.0000
But Is It User Friendly?	22.9500	7000.0000
Onions, Leeks, and Garlic: Cooking Secrets Mediterranean	20.9500	7000.0000
Computer Phobic and Non-Phobic Individuals Behavior Variations	21.5900	7000.0000

Cktle.obq оператор обновляет одну из книг, видимых посредством курсора *hiprice*, изменяя ее цену на \$14.99.

```
SQL:
update hiprice
set price = $14.99
where title = 'Secrets of Silicon Valley'
```

Если оператор WITH CHECK OPTION был включен в определение курсора *hiprice*, то оператор UPDATE будет отвергнут, поскольку новая цена книги *Secrets of Silicon Valley* сделает ее неприемлемой для данного курсора. Поскольку определение курсора *hiprice* не включает оператор WITH CHECK OPTION, оператор UPDATE будет выполнен. Но когда вы в следующий раз будете просматривать данные посредством курсора *hiprice*, вы уже не увидите книги *Secrets of Silicon Valley*.

Опция проверки должна использоваться только в том случае, если определяемый курсор является обновляемым. В сущности предложение WITH CHECK OPTION позволяет вам указывать правила модификации данных посредством курсоров. Если условия, указанные в определении курсора, нарушаются каким-нибудь из операторов модификации данных, такой оператор считается недопустимым. Подробнее об этом см. в справочных руководствах по своей системе.

Разборка курсора

Процесс реализации запроса к какому-либо курсору с его запомненным определением и преобразования его в запрос к таблицам, лежащим в основе этого курсора, называется разборкой курсора. В ходе этого процесса может возникнуть несколько проблем.

Если какие-то таблицы, курсоры или столбцы, лежащие в основе курсора, были удалены или переименованы, или если в результате реструктуризации возникли несовместимости между типами данных, система не сможет интерпретировать курсор, а вы в свою очередь не сможете им воспользоваться. Вместо этого на экране появится сообщение об ошибке, выданное SQL.

Если в таблицу, на которой основан курсор, добавить столбцы, то эти новые столбцы могут не появиться в курсоре, определенном с помощью оператора `SELECT *`, если перед этим он не будет удален и переопределен. Это связано с тем, что многие диалекты SQL интерпретируют и расширяют краткую форму записи типа “звездочки” в момент создания курсора. В Transact-SQL, например, расширение краткой формы записи типа “звездочка” — единственный вид интерпретации, выполняемой в момент создания курсора (а не в момент запроса или модификации). Подробности, связанные с работой вашей системы, можно найти в соответствующих справочных руководствах.

Переопределение курсоров

Поскольку курсоры можно определять в терминах других курсоров, можно, в конце концов, создать целую цепочку зависящих друг от друга курсоров. Точно так же, как настоящая цепочка может оборваться на любой связи, это же может произойти и с цепочкой курсоров. Любой из курсоров в цепочке можно переопределить таким образом, что зависящие от него курсоры потеряют всякий смысл.

В качестве примера рассмотрим три поколения курсоров, полученные из таблицы `authors`:

SQL:

```
create view number1
as select au_lname, phone
from authors
where zip like '94%'
```

SQL:

```
select * from number1
```

Результат:

au_lname	phone
Bennet	415 658-9932
Green	415 986-7020
Carson	415 548-7723
Stringer	415 843-2991
Straight	415 834-2919
Karsen	415 534-9219
MacFeather	415 354-7128
Dull	415 836-7128
Yokomoto	415 935-4228
White	408 496-7223
Hunter	415 836-7128
Locksley	415 585-4620

```
SQL:
create view number2
as select au_lname, phone
from number1
where au_lname like '[M-Z]%'
```

```
SQL:
select * from number2
```

Результат:

au_lname	phone
Stringer	415 843-2991
Straight	415 834-2919
MacFeather	415 354-7128
Yokomoto	415 935-4228
White	408 496-7223

```
SQL:
create view number3
as select au_lname, phone
from number2
where au_lname like 'MacFearther'
```

```
SQL:
select * from number3
```

Результат:

au_lname	phone
MacFeather	415 354-7128

Что произойдет, если переопределить курсор *number2* на основе других критериев выбора, например почтового кода, начинающегося с *947*? Будет ли по-прежнему доступным курсор *number3*, который зависит от курсора *number2*? Ответ заключается в том, что все здесь зависит от конкретной реализации. В некоторых системах курсором *number3* можно будет с успехом пользоваться, хотя данные, видимые посредством этого курсора, будут отличаться. При использовании запроса, содержащего ссылку на *number2* или *number3*, курсор будет интерпретироваться как обычно.

Можно, конечно, переопределить курсор *number2* таким образом, чтобы использование курсора *number3* стало невозможным. Если бы, например, новая версия *number2* включала только столбец *au_lname* (без столбца *phone*), то курсор *number3* уже нельзя было бы использовать в запросе, поскольку он не может получить столбец *phone* из объекта, от которого он зависит. Однако курсор *number3* по-прежнему существовал бы, а в некоторых системах им можно было бы снова пользоваться путем удаления и повторного создания курсора *number2*, а также повторной вставки в него столбца *phone*.

Другими словами, некоторые системы позволяют вам изменять определение промежуточного курсора, не оказывая при этом влияния на зависимые курсоры, если *целевой список* зависимых курсоров при этом остается допустимым. Если вы нарушите это правило, запрос, содержащий ссылку на неправильный курсор, приведет к появлению сообщения об ошибке.

МОДИФИКАЦИЯ ДАННЫХ ПОСРЕДСТВОМ КУРСОРОВ

Изменение данных посредством курсоров — непростая задача. Важнейшая проблема, как будет здесь продемонстрировано, состоит в том, что SQL иногда не удается однозначно понимать команды, предназначенные для изменения данных в курсоре. Такие изменения не допускаются любой версией SQL. Другими словами, некоторые курсоры являются необновляемыми по своей сути.

Есть и другие курсоры, которые по логике являются обновляемыми, но во многих версиях SQL трактуются как необновляемые. Запрет использования широкого диапазона операторов модификации данных налагает более жесткие ограничения, но в то же время упрощает правила относительно того, что можно и чего нельзя делать. В других реализациях SQL поддерживаются максимальные возможности по изменениям, несмотря на сложности, неизбежно появляющиеся в правилах, касающихся модификации данных.

Перечни операторов модификации данных, которыми разрешено пользоваться в курсорах, отличаются в разных реализациях SQL. Поэтому правила, которые встретятся вам в этой книге, следует воспринимать лишь как общие ориентиры, а за подробностями лучше обратиться к справочным руководствам по своей системе.

Правила в соответствии с ANSI

Стандарт ANSI устанавливает, что курсоры являются только читаемыми (т.е. не модифицируемыми), если оператор CREATE VIEW содержит один из следующих элементов.

- DISTINCT в списке выбора.
- Выражения (вычисляемые столбцы, функции и т.п.) в списке выбора.
- Ссылки на несколько таблиц либо в предложении FROM, либо в подзапросе, либо в предложении UNION.
- Ссылки на курсор, который сам по себе не является обновляемым, либо в предложении FROM, либо в подзапросе.
- Предложение GROUP BY или HAVING.

Некоторые диалекты SQL налагают менее жесткие ограничения, чем стандарт ANSI, другие же, наоборот, — более жесткие. Если вас интересуют подробности, обратитесь к своему справочному руководству или поэкспериментируйте. Когда вы пытаетесь модифицировать данные посредством курсора, SQL проверяет, не нарушаются ли при этом имеющиеся ограничения. Если обнаруживается такое нарушение, SQL отвергает соответствующий оператор модификации данных, выдавая сообщение об ошибке.

Чтобы лучше понять смысл этих ограничений, рассмотрим ряд примеров необновляемых курсоров. Начнем с ограничения, которое запрещает обновление курсоров со столбцами, полученными в результате вычислений из других столбцов.

Столбец *gross_sales* в курсоре *accounts* вычисляется на основе столбцов *price* и *ytd_sales* из таблицы *titles*:

SQL:

```
create view accounts (title, advance, gross_sales)
as
select title_id, advance, price*ytd_sales
from titles
where price > $15
      and advance > $5000
```

В курсоре *accounts* находятся следующие строки:

SQL:

```
select *
from accounts
```

Результат:

title	advance	gross_sales
PC8888	8000.0000	81900.0000
PC1035	7000.0000	201501.0000
TC3218	7000.0000	7856.2500
PS1372	7000.0000	8096.2500

Представьте себе, что означало бы обновление столбца *gross_sales*. Как могла бы система получить значения цены и объема продаж за последний год до текущей даты, исходя из произвольного значения, которое вы могли бы ввести? Таким образом, обновления для этого курсора объявляются недопустимыми.

Теперь обратимся к курсору, определение которого включает предложение GROUP BY. Такие курсоры (и любой курсор, полученный из них) называются **сгруппированными курсорами (grouped views)**. На сгруппированные курсоры может налагаться множество ограничений (это зависит от конкретной версии SQL, которой вы пользуетесь). Ниже приведен оператор, определяющий такой курсор:

SQL:

```
create view categories (Category, Average_Price)
as select type, avg(price)
from titles
group by type
```

А вот как выглядит сам курсор:

SQL:

```
select *
from categories
```

Результат:

Category	Average_Price
UNDECIDED	NULL
business	13.730
mod_cook	11.490
popular_comp	21.475
psychology	13.504
trad_cook	15.963

Нет никакого смысла вставлять в курсор *categories* новые строки. Какой группе должна принадлежать вставляемая строка? Недопустимо также выполнять какие-либо обновления по отношению к столбцу *Average_Price*, поскольку из любого значения, которое вы могли бы ввести в него, невозможно узнать, как должны измениться исходные цены. Теоретически, конечно, можно допустить обновления в столбце *Category* и удаления строк, но такие действия не поддерживаются многими версиями SQL.

Некоторые диалекты SQL налагают ограничения не только на обновление сгруппированных курсоров, но и на запросы к ним. Например, в таких реализациях SQL следующий запрос был бы недопустимым:

SQL (возможно, недопустимый):

```
select *
from categories
where Average_Price > $12.00
```

Проблемы с этим запросом возникают при его интерпретации: результирующий оператор SELECT является недопустимым, поскольку в операторе WHERE не

допускается использование агрегирующих функций. Вот как мог бы выглядеть такой (недопустимый) оператор:

```
SQL (недопустимый):
select type, avg(price)
from titles
where avg(price) > $12.00
group by type
```

Еще одно ограничение на модификации курсоров запрещает выполнение обновлений и вставок данных в курсор, если соответствующий оператор модифицирует столбцы, получаемые из более чем одного объекта. Это связано с тем, что в одном операторе не допускается выполнение обновления более чем одного объекта.

Некоторые системы позволяют модифицировать курсор, если при этом изменятся столбцы только одной из таблиц, на которых он основан. Если, например, курсор содержит три столбца из таблицы *titles* и два столбца из таблицы *publishers*, эти системы разрешили бы выполнить операцию обновления для курсора, если бы она изменяла только столбцы из таблицы *titles*. Но нельзя выполнить оператор UPDATE, который изменял бы один столбец из таблицы *titles* и один столбец из таблицы *publishers*.

СОЗДАНИЕ КОПИЙ ДАННЫХ

Мы обращали ваше внимание в этой главе на то обстоятельство, что курсоры являются не копиями данных, а виртуальными объектами, с которыми не ассоциируются какие-либо физические данные. Если все, что вам требуется, это независимая копия данных, выясните, какие возможности в этом отношении предоставляет ваша система. Transact-SQL реализует эту функцию посредством предложения INTO в операторе SELECT. Он позволяет вам определить таблицу и поместить в нее данные (основываясь на существующих определениях и данных), не проходя при этом через обычный процесс определения данных. Новая таблица (созданная в операторе INTO) основывается на столбцах, указанных вами в списке выбора, таблице (таблицах), имя которой вы указали в предложении FROM, и строках, выбранных вами в предложении WHERE. Однако это использование оператора INTO не является широко распространенным, и следующие примеры относятся только к Transact-SQL.

Представим, например, что вам требуется новая таблица, называемая *newbooks*, которая состоит из двух столбцов, взятых из таблицы *titles*, и некоторого подмножества ее строк. Вот как можно ее создать:

```
SQL:
select title_id, type
into newbooks
from titles
where price >$20
```

А вот как будет выглядеть таблица *newbooks*:

```
SQL:
select *
from newbooks
```

Результат:

title_id	type
PC1035	popular_comp
TC3218	trad_cook
PS1372	psychology

Эта новая таблица с реальной (физической) копией данных из “родительской” таблицы становится частью базы данных. На данные в “родительской” таблице это никоим образом не влияет.

Оператор SELECT INTO может оказаться полезным при создании тестовых таблиц, новых таблиц, которые напоминают уже существующие таблицы, и таблиц, содержащих все или некоторые из столбцов других таблиц.

Оператор SELECT INTO можно также использовать для создания таблицы-шаблона, не содержащей никаких данных. Для этого достаточно поместить в предложение WHERE какое-либо ложное условие. Например:

SQL:

```
select *  
into newpubs  
from publishers  
where 1 = 2
```

SQL:

```
select *  
from newpubs
```

Результат:

pub_id	pub_name	address	city	state
--------	----------	---------	------	-------

ВОПРОСЫ АДМИНИСТРИРОВАНИЯ БАЗ ДАННЫХ

Следующая глава представляет собой обзор ряда оставшихся вопросов управления базами данных: безопасность, транзакции, производительность и целостность.

Безопасность, транзакции, производительность и целостность

УПРАВЛЕНИЕ БАЗАМИ ДАННЫХ В РЕАЛЬНОМ МИРЕ

Эта глава посвящена четырем проблемам, имеющим особую важность в практических применениях баз данных.

Первая из этих проблем, безопасность, решается практически в любой системе управления реляционными базами данных с помощью двух механизмов: назначения **полномочий** (называемых также **привилегиями**) с помощью команд SQL GRANT и REVOKE и создания курсоров (благодаря которым пользователи могут иметь избирательный доступ к базе данных). В некоторых системах также используются специальные процедуры (совокупности расширенных операторов SQL с параметрами), позволяющие управлять действиями пользователя. Еще три проблемы — транзакции, производительность и целостность — решаются в различных системах управления базами самыми разнообразными способами. По этой причине в данной главе акцент в обсуждении указанных проблем делается скорее на концепциях, а не на подробностях синтаксиса.

Безопасность целесообразно рассматривать лишь в многопользовательской среде: наделение полномочиями других пользователей имеет смысл только тогда, когда они работают в системе совместно. Управление транзакциями позволяет системе баз данных решить проблему устранения коллизий при одновременном поступлении нескольких запросов от разных пользователей на получение одних и тех же данных (иногда эту функцию называют **контролем совпадений (concurrency control)**). Кроме того, управление транзакциями требуется системам управления базами данных для целей **восстановления (recovery)** — на случай сбоя программного обеспечения или носителя информации. Восстановление имеет большое значение как для однопользовательских, так и для многопользовательских систем.

Целостность данных также важна, независимо от того, в какой системе вы работаете, — однопользовательской или многопользовательской. Производительность особенно важна в крупных многопользовательских окружениях баз данных, хотя она может превращаться в проблему и в однопользовательских системах.

В прошлом производительность считалась уязвимым местом реляционной модели. Сейчас некоторые реляционные продукты функционируют не хуже, а иногда и лучше систем, основанных на других моделях данных.

Реляционные модели также часто критикуют за то, что они не обеспечивают надлежащих гарантий целостности данных. Однако в отличие от большинства критиков производительности, те, кто горячее всех обсуждает вопросы целостности, являются последовательными приверженцами реляционной модели. Как вы, наверное, помните из главы 1, д-р И.Ф. Кодд, автор реляционной модели, включил в свои двенадцать правил для систем реляционных баз данных весьма жесткие требования к целостности данных. Поэтому стандарт ANSI SQL-92 включает язык для определения некоторых ограничений на целостность, а большинство разработчиков усилили свои SQL с целью реализации повышенных требований к целостности.

БЕЗОПАСНОСТЬ ДАННЫХ

Безопасность является важным аспектом управления базами данных, поскольку информация представляет собой чрезвычайно ценный ресурс. Большая часть данных, которые стоит верить системе управления базами данных, должна быть защищена от несанкционированного доступа — т.е. просмотра, модификации или удаления со стороны тех, кто не уполномочен это делать.

Если вы являетесь единственным пользователем какой-то базы данных, то проблемы обеспечения безопасности сводятся к определению того, как защитить эту базу данных в целом, — но этот вопрос уже не относится к сфере компетенции SQL. Если вы используете персональный компьютер, например, все ваши меры обеспечения безопасности могут заключаться лишь в том, чтобы не забывать закрыть на замок комнату, в которой установлен ваш компьютер.

С другой стороны, даже если вы являетесь единственным санкционированным пользователем базы данных, система управления базой данных может функционировать в совместно используемой среде (например, на многопользовательском компьютере или на компьютере, работающем в составе компьютерной сети). В большинстве таких компьютеров средства обеспечения безопасности реализованы на уровне операционной системы. Как правило, в таких случаях для входа в систему требуется прежде всего задать пароль; кроме того, каждый файл в таких системах часто помечается специальными атрибутами (в соответствии с тем, какие пользователи имеют право читать, модифицировать или выполнять эти файлы).

Когда вы работаете с какой-либо базой данных совместно с другими пользователями, потребности обеспечения безопасности реализуются на уровне SQL и системы управления базой данных, для чего создано два механизма. Первый механизм основан на выделении полномочий (которые иногда называются разрешениями): с помощью команд SQL GRANT и REVOKE указывается, каким пользователям предоставляется право выполнять определенные команды в отношении определенных таблиц, курсоров и столбцов. Второй механизм обеспечения безопасности заключается в использовании курсоров в сочетании с командами GRANT и REVOKE для предоставления избирательного доступа к различным подмножествам данных.

Оба механизма предполагают, что в системе базы данных предусмотрен способ идентификации (распознавания) каждого конкретного пользователя.

Идентификация пользователя и особые пользователи

Системы управления базами данных весьма отличаются друг от друга по своим подходам к идентификации пользователей — как в базовой концепции, так и в деталях. Вряд ли удастся сказать многое о каких-то общих принципах; пожалуй, лучшее, что можно сделать, это описать в самых общих чертах основные различия между ними.

В некоторых системах основной упор делается на идентификации пользователей на уровне операционной системы. Другими словами, они распознают как санкционированного пользователя любого, кто может открыть сеанс работы на данном компьютере. Это, однако, не означает, что всякий, кто может получить доступ к системе базы данных, сможет делать в ней все, что ему заблагорассудится.

Более сложные системы управления базами данных включают свои собственные механизмы идентификации и санкционирования, которые требуют, чтобы у каждого пользователя было свое “узаконенное” имя (иногда его называют точкой доступа или точкой входа) и пароль для системы в целом и/или для каждой базы данных в этой системе (разумеется, если система поддерживает концепцию нескольких взаимодействующих баз данных). Прикладные программы могут предоставлять дополнительные уровни защиты.

Что же произойдет после того, как система управления базой данных распознает ваш идентификатор и перепроверит его, запросив и приняв ваш пароль? Начиная с этого момента, ваши возможности будут зависеть от того, к какому классу пользователей вы принадлежите, и от того, какими полномочиями вы были наделены со стороны других пользователей.

Большинство многопользовательских систем управления базами данных распознают по крайней мере два вида пользователей, наделенных особыми привилегиями: суперпользователя, которого зачастую называют DBA (database administrator — администратор базы данных), или системным администратором, и владельцев объектов базы данных. Некоторые системы баз данных распознают дополнительные типы пользователей и устанавливают иерархию, в которой привилегии на выполнение различных команд назначаются пользователям в зависимости от их места в этой иерархии.

Администратор базы данных назначается в момент инсталляции системы. DBA — это особая должность, которая может быть (временно) занята любым пользователем, знающим соответствующее имя и пароль. Во многих инсталляциях баз данных роль DBA совместно выполняется несколькими людьми. Каждый из этих пользователей может войти в систему как DBA для выполнения административных задач, но для выполнения других функций в этой базе данных он должен пользоваться своим собственным именем. Как альтернативный вариант, система может поддерживать ряд ролей. В этом случае у пользователей могут быть индивидуальные идентификаторы для базы данных, такие как “Stacy” и “Tom”; кроме того, им могут быть присвоены функциональные роли (DBA, оператор и т.п.). В такой системе пользователи имеют полномочия, которые требуются им для выполнения каких-то специальных работ, а ролевая деятельность может быть расписана для конкретных лиц.

DBA, как правило, наделяется рядом особых привилегий; кроме того, на него возлагается ответственность за нормальное функционирование прикладного программного обеспечения базы данных. С точки зрения нашего обсуждения вопросов безопасности, DBA (в большинстве систем) располагает всеми полномочиями по всем объектам базы данных и для всех команд. Какие из этих разрешений DBA может передавать другим пользователям, зависит от каждой конкретной системы.

Другой особой категорией пользователей, которая представляет интерес с точки зрения безопасности, является владелец каждой конкретной таблицы или курсора. Во многих системах пользователь, который создает какую-нибудь таблицу или курсор, является ее владельцем. (Некоторые системы поддерживают дополнительную концепцию владельца базы данных.) Владелец таблицы или курсора может автоматически наделяться правом выполнять любые операции с этой таблицей или курсором, в том числе (как правило) выдавать разрешение на это другим пользователям.

Пользователи, не являющиеся владельцами таблицы или курсора (а также DBA), должны получить в явном виде разрешение на выполнение каких-либо операций с этой таблицей или курсором. Такие разрешения реализуются с помощью команд SQL GRANT и REVOKE, которые поддерживаются практически любой многопользовательской реализацией SQL.

Информация, касающаяся идентификации пользователей и разрешений, может храниться в словаре данных.

Команды GRANT и REVOKE

Фразы “наделение полномочиями” или “назначение разрешений” часто ассоциируются с обсуждением команд GRANT и REVOKE. В большинстве реализаций SQL вы не имеете права делать того, на что у вас нет явно выраженной санкции.

Команды GRANT и REVOKE указывают, каким пользователям можно выполнять определенные операции в отношении определенных таблиц, курсоров и столбцов. В некоторых версиях SQL разрешение на выполнение таких команд, как CREATE TABLE и DROP INDEX, также должно выдаваться явным образом (как правило, администратором базы данных).

Операции, выполнение которых контролируется с помощью GRANT и REVOKE, включают SELECT, UPDATE, INSERT, DELETE и REFERENCES. Объектами базы данных, к которым применимо разрешение на выполнение этих

операций, являются таблицы и курсоры. Санкция на выдачу этих разрешений исходит от владельца соответствующего объекта и может передаваться вместе с самим разрешением. Существует два варианта синтаксиса команды GRANT, отличающихся по местоположению списка столбцов.

```
GRANT {ALL | список_полномочий}
ON {имя_таблицы [(список_столбцов)] | [список_столбцов]}
TO {PUBLIC | }
[WITH GRANT OPTION]
```

```
GRANT {ALL | список_полномочий [(список_столбцов)]}
ON { имя_таблицы | имя_курсора }
TO {PUBLIC | список_пользователей }
[WITH GRANT OPTION]
```

Команда REVOKE похожа на команду GRANT:

```
REVOKE {ALL | список_полномочий }
ON {имя_таблицы [(список_столбцов)] | имя_курсора [(список_столбцов )]}
FROM {PUBLIC | список_пользователей }
```

```
REVOKE {ALL | список_полномочий [(список_столбцов)]}
ON { имя_таблицы | имя_курсора }
FROM {PUBLIC | список_пользователей }
```

Как это часто бывает в SQL, синтаксис выглядит намного сложнее, чем большинство операторов, с которыми вам придется иметь дело в действительности. Прежде чем мы объясним это подробнее, приведем простой пример, в котором Mary получает разрешение вставлять данные в таблицу *titles* и обновлять ее:

```
SQL:
grant insert, update
on titles
to mary
```

На рис. 10.1 представлено небольшое отличие в операторах из двух диалектов SQL, которые аннулируют разрешение Mary на обновление столбцов *advance* и *price* в таблице *titles*. Другие варианты синтаксиса приведены в Приложении Б.

```
SQL:
revoke update (advance, price)
on titles
from mary
```

```
SQL:
revoke update
on titles (advance, price)
from mary
```

Рис. 10.1. Сравнение двух операторов REVOKE

Рассмотрим подробнее синтаксис команд GRANT и REVOKE. Первая строка включает ключевое слово GRANT или REVOKE и либо ключевое слово ALL, либо список полномочий, которыми наделяется пользователь (или которые снимаются с него). Если в списке необходимо указать несколько полномочий, они разделяются запятыми. Если используется ключевое слово ALL, то предоставляются или аннулируются все полномочия, применимые к данному объекту (т.е. все полномочия, которыми обладает наделяющий или аннулирующий пользователь).

В операторе ON указывается таблица или курсор, на которые выдаются или аннулируются полномочия (по одной таблице или одному курсору на оператор).

В разных версиях SQL допускаются разные наборы полномочий. В этих наборах всегда присутствуют SELECT, UPDATE, INSERT и DELETE, а в SQL-92 добавлено REFERENCES (право ссылаться из оператора CREATE TABLE на другие таблицы). Реже включаемыми полномочиями являются ALTER, INDEX и EXECUTE. Разные SQL отличаются друг от друга возможностью управлять выдачей разрешений на некоторые, все или на одну из этих операций на уровне столб-

цов, таблиц или курсоров. Когда полномочия выдаются по отдельным столбцам, имя (имена) столбца (столбцов) перечисляются в круглых скобках.

Полномочия могут выдаваться на несколько столбцов одновременно, но все эти столбцы должны быть в одной и той же таблице или в одном и том же курсоре. Если вы не включаете список имен столбцов, полномочия выдаются или аннулируются для всей таблицы или курсора. (В некоторых системах список столбцов указывается в первой строке.)

Следующая строка в синтаксисе начинается с ключевого слова `TO` (для команды `GRANT`) или `FROM` (для команды `REVOKE`). Ключевое слово `PUBLIC` означает всех пользователей в системе.

Альтернативой `PUBLIC` является список имен пользователей, которых вы хотите наделить полномочиями (или наоборот, лишить их). Как обычно, имена в этом списке разделяются запятыми. Некоторые диалекты SQL поддерживают концепцию пользовательских групп и ролей (или того и другого); в этих SQL в список пользователей можно включить название какой-либо группы или роли.

Необязательный оператор `WITH GRANT OPTION` определяет, может ли пользователь, наделяемый полномочиями, в свою очередь передавать их другим пользователям. Например, ниже приведен оператор, который наделяет `Mary` правом выполнять операцию `SELECT` в таблице *authors* и позволяет ей передавать это право другим пользователям.

SQL:

```
grant select
on authors
to mary
with grant option
```

Когда вы устанавливаете разрешения для какой-либо таблицы или курсора, владельцем которых являетесь именно вы, лучше всего использовать команды `GRANT` и `REVOKE` в соответствии с каким-то определенным и хорошо продуманным планом, а не выполнять их хаотически, под влиянием сиюминутных настроений.

Ниже приведено несколько примеров, которые показывают, как пользоваться операторами `GRANT` и `REVOKE` для выделения полномочий на определенную таблицу. Допустим, вы являетесь владельцем таблицы *titles* и решаете, что большинству пользователей системы следует разрешить доступ к этой таблице — за исключением столбцов, в которых речь идет о деньгах и продажах. Вы хотите разрешить модифицировать эти столбцы только трем конкретным пользователям (*Sara*, *John* и *Leslie*).

Такой вариант можно реализовать, воспользовавшись одним из двух подходов. Самый простой из них — назначить конкретные полномочия конкретным пользователям. В данном случае это будет означать выдачу нескольких команд `GRANT`. Вам потребуется один оператор, чтобы наделить всеми полномочиями пользователей с именами *Sara*, *John* и *Leslie*:

SQL:

```
grant select, insert, delete, update
on titles
to sara, john, leslie
```

Теперь вам потребуется оператор, чтобы наделить полномочиями на выборку всех остальных пользователей:

SQL:

```
grant select
on titles
to linda, pat, steve, chris, cathy, peter, lee, carl, karen
```

Список пользователей может быть весьма пространным; кроме того, вам понадобится еще один оператор, предоставляющий всем им разрешение модифицировать все столбцы в таблице *titles*, за исключением *advance*, *price* и *ytd_sales*. В зависимости от синтаксических правил, действующих в вашей системе, вам придется воспользоваться одним из двух следующих операторов:

```
SQL:
grant update
on titles (title_id, title, type, pub_id, contract, notes, pubdate)
to linda, pat, steve, chris, cathy, peter, lee, carl, karen

grant update (title_id, title, type, pub_id, contract, notes, pubdate)
on titles
to linda, pat, steve, chris, cathy, peter, lee, carl, karen
```

Поскольку вы собираетесь предоставить большинству пользователей большинство полномочий, проще было бы назначить все полномочия всем пользователям, а затем аннулировать конкретные полномочия у некоторых пользователей. Вот как это делается. Начните с оператора, который наделяет всех пользователей правами выбирать данные из таблицы *titles*, вставлять данные в эту таблицу, удалять из нее данные и обновлять ее.

```
SQL:
grant all
on titles
to public
```

Вы проявили потрясающую щедрость: теперь любой пользователь может модифицировать таблицу *titles* по своему усмотрению. Чтобы защитить самые важные столбцы, выполните оператор, изменяющий ситуацию, созданную предшествующим оператором. Приведенные ниже операторы аннулируют права всех пользователей на обновление столбцов *advance*, *price* и *ytd_sales* в таблице *titles*. Другие права, дарованные в предшествующем операторе, остаются неизменными.

```
SQL:
revoke update
on titles (price, advance, ytd_sales)
from public
```

Теперь, чтобы предоставить пользователям с именами Sara, John и Leslie право обновлять эти столбцы, можно задать следующую команду:

```
SQL:
grant update
on titles (price, advance, ytd_sales)
to sara, john, leslie
```

Вам наверняка потребуется включить в этот список пользователей и себя, поскольку исключение FROM PUBLIC относится и к вам.

Конфликты операторов GRANT и REVOKE. Как явствует из предыдущих примеров, для операторов GRANT и REVOKE имеет большое значение порядок их следования (в большинстве реализаций SQL). В случае какого-либо конфликта оператор, выполненный самым последним, отменяет действие всех предыдущих операторов. Так, например, если все пользователи были наделены правом на выборку данных из таблицы *titles*, а затем пользователю с именем Joe было запрещено просматривать столбец *advance*, то он сможет просматривать все столбцы, за исключением столбца *advance*, в то время как другие пользователи смогут по-прежнему видеть все столбцы. Аналогично, оператор GRANT или REVOKE изменит все установленные ранее полномочия.

Результатом действия этого правила является то, что одни и те же операторы GRANT и REVOKE, выполненные в разной последовательности, могут создавать

совершенно разные ситуации. Например, приведенная ниже совокупность операторов запрещает пользователю с именем Joe выполнять операции SELECT в отношении таблицы *titles*:

```
SQL:
grant select
on titles
to joe

revoke select
on titles
from public
```

А теперь рассмотрим те же операторы, но заданные в обратном порядке:

```
SQL:
revoke select
on titles
from public

grant select
on titles
to joe
```

Теперь только Joe имеет право выполнять операцию SELECT.

Помните, когда вы пользуетесь ключевым словом PUBLIC, вы включаете в этот круг пользователей и себя. Вы можете таким образом даже запретить себе модифицировать свою собственную таблицу, одновременно предоставив себе право обращаться к курсору, сформированному на ее основе, или к процедуре, в которой есть ссылки на эту таблицу. (Разумеется, ничто не мешает вам в любой момент изменить свое решение и восстановить свои права с помощью оператора GRANT.) Вы, вероятно, часто будете пользоваться ключевым словом PUBLIC в качестве быстрого способа аннулирования полномочий с последующим определением каких-то исключений, как было показано в предыдущем примере.

Курсоры как механизм обеспечения безопасности

Второй механизм обеспечения безопасности в большинстве систем управления реляционными базами данных связан с использованием курсоров в сочетании с командами GRANT и REVOKE. Разрешение на доступ к подмножеству данных в курсоре должно быть предоставлено или аннулировано явным образом, независимо от действующей совокупности полномочий, относящихся к таблице (или таблицам), на которой основан данный курсор.

С помощью курсора пользователи могут запрашивать и модифицировать только те данные, которые они могут видеть. Остальная часть базы данных является для них и невидимой, и недоступной. Например, для вас может быть нежелательно, чтобы какие-то пользователи имели возможность доступа к столбцам в таблице *titles*, в которых содержатся сведения о деньгах и продажах. Вы могли бы создать курсор на основе таблицы *titles*, который “отсекает” эти столбцы (назовем его *bookview*), а затем предоставить всем пользователям права на этот курсор. Вот как это сделать.

```
SQL:
revoke all
on titles
from public

grant all
on bookview
to public
```

```
grant all
on titles
to mccann, himmelmwright, brady, mandelbaum
```

Определяя различные курсоры и выборочно предоставляя права на них, можно ограничить доступ пользователя (или любой группы пользователей) к различным подмножествам данных.

- Доступ может быть ограничен некоторым подмножеством строк таблицы базы. Можно, например, определить курсор, который будет содержать только строки для книг по бизнесу и психологии, упрятав тем самым от некоторых пользователей информацию о других видах книг.
- Доступ может быть ограничен некоторым подмножеством столбцов таблицы базы. Можно, например, определить курсор, который будет содержать все строки таблицы *titles*, отсекая при этом столбцы *ytd_sales* и *advance*, ввиду особой важности представленной в них информации.
- Доступ может быть ограничен некоторым подмножеством строк и столбцов таблицы базы. Можно, например, определить курсор, который будет содержать информацию только о книгах по бизнесу и психологии, и не включать финансовую информацию о них.
- Доступ может быть ограничен строками, которые удовлетворяют условию объединения нескольких таблиц базы. Можно, например, определить курсор, который будет объединять таблицы *titles*, *authors* и *titlesauthors*, позволяя отображать фамилии авторов и названия написанных ими книг. Этот курсор будет “скрывать” персональные данные об авторах и финансовую информацию о книгах.
- Доступ может быть ограничен статистическими итоговыми данными. Можно, например, определить курсор, который будет показывать среднюю цену книг каждого вида.
- Доступ может быть ограничен некоторым подмножеством другого курсора или определенного сочетания курсоров и таблиц базы. Можно, например, определить новый курсор на основе курсора, описанного в предыдущем примере, который отображает средние цены книг по кулинарии и компьютерам.

Создание курсоров и реализация с их помощью (а также с помощью таблиц базы) системы доступа нередко является частью процесса определения данных. Однако ничто не мешает вам в любой момент изменить эту схему санкционирования, определив новые курсоры и написав новые операторы GRANT и REVOKE. Более подробная информация о курсорах приведена в главе 9.

ТРАНЗАКЦИИ

Транзакция (transaction) представляет собой логическую единицу работы СУБД. **Управление транзакциями (transaction management)** обеспечивает интерпретацию некоторой совокупности операторов SQL как единого блока (т.е. как неделимого объекта). Другими словами, управление транзакциями гарантирует, что либо будут выполнены все операции из некоторой группы (т.е. транзакция), либо не будет выполнена ни одна из них: транзакция — это альтернатива типа “все или ничего”.

Транзакции необходимы для контроля совпадений (устранения конфликтов пользователей, пытающихся одновременно обратиться к базе данных) и восстановления (предоставления возможности системе базы данных вернуть эту базу данных в нормальное состояние после сбоя программного или аппаратного обеспечения).

SQL автоматически управляет всеми командами, включая простые запросы на изменение данных, как транзакциями. Пользователь также может с помощью специальных команд сгруппировать некоторую совокупность операторов SQL в **транзакцию, определенную пользователем** (синтаксис этих команд меняется от системы к системе, но обычно они содержат такие ключевые слова, как BEGIN, COMMIT и ROLLBACK).

Транзакции и совпадения

В многопользовательских системах несколько пользователей и/или транзакций могут одновременно обращаться к одним и тем же данным. Предотвращение создания взаимных помех при одновременных транзакциях (контроль совпадений) гарантирует, что до тех пор, пока процесс изменения данных не будет завершен, никакой другой пользователь не сможет работать с этими данными или даже просматривать их.

Классическим примером ситуации, когда такие совпадения могут представлять серьезную проблему, является покупка авиабилетов.

Допустим, вы звоните в авиакасса и заказываете билет до Таити. Кассир в авиакассе направляет в базу данных запрос, который выбирает все свободные места, и сообщает вам, что на самолет, отправляющийся сегодня вечером, есть только одно свободное место. Пока вы размышляете о том, как быть, другой отчаявшийся турист звонит в бюро путешествий, пытается выяснить, нельзя ли сегодня вечером вылететь на Таити. Агент бюро путешествий также направляет запрос в базу данных — ту самую, в которую уже обращался ваш кассир, — и получает тот же результат (одно свободное место на сегодня) и сообщает об этом вашему “конкуренту”. Тем временем вы “созреваете”, и кассир авиакассы вносит изменения в базу данных, отражая тот факт, что последний билет уже продан. Вы начинаете паковать, даже не подозревая о том, что агент бюро путешествий (которому ничего не известно об изменениях, внесенных кассиром в базу данных) только что продал билет на то же место. Фактически, обновление, выполненное агентом бюро путешествий, “затирает” обновление, выполненное вашим кассиром. Когда вы прибываете в аэропорт, вас постигает жестокий удар.

Этот пример демонстрирует одну из опасностей, таящихся в одновременных запросах, посылаемых в большие, совместно используемые системы баз данных. Чтобы справиться с проблемами “накладок”, в большинстве систем управления реляционными базами данных применяется механизм, называемый **блокировкой (locking)**. Подробное описание механизма блокировки не входит в задачи этой книги, тем более что в большинстве систем баз данных этот механизм реализуется автоматически, избавляя пользователей даже от мыслей на эту тему. Поэтому есть смысл познакомиться с этим механизмом лишь в самых общих чертах.

Блокировка. В сущности, принцип работы блокировки заключается в том, что когда пользователь выбирает какие-то данные из базы данных, система управления базой данных автоматически закрывает их “на замок”, с тем чтобы никакой другой пользователь не мог обновлять эти данные до тех пор, пока блокировка не будет снята.

Существует много видов блокировки, но для нас важнее всего понять разницу между **исключительной (exclusive)** блокировкой и **совместной (shared)** блокировкой. Исключительная блокировка применяется в системах баз данных в ходе операций модификации данных (так называемых операций записи): UPDATE, INSERT, DELETE. Когда исключительная блокировка применяется к некоторой совокупности данных, никакая другая транзакция не может установить по отношению к этим данным никакой вид блокировки до тех пор, пока в конце транзакции по модификации данных не будет снята исходная блокировка.

Совместные блокировки применяются при выполнении операций, не связанных с обновлением данных (по-другому они называются операциями чтения), таких как SELECT. Применение совместной блокировки к некоторой совокупности данных не позволяет операции записи выполнить исключительную блокировку этих данных, но позволяет другим операциям чтения выполнять свои собственные совместные блокировки даже в том случае, если первая транзакция еще не завершилась.

В основном система базы данных выполняет операции блокирования и разблокирования автоматически — механизм в целом остается невидимым для пользователей. В некоторых системах, однако, пользователи имеют возможность управлять определенными аспектами блокировки. С помощью команд SQL они могут выбирать уровень автоматического применения блокировки (например, уровень таблицы или уровень строки) или продолжительность действия определенных видов блокировки.

Транзакции и восстановление

Транзакция является не только единицей работы, но и единицей восстановления. Восстановлением называется способность системы базы данных “приводить базу данных в чувство” после системного сбоя — т.е. в самое последнее по времени текущее состояние, которое можно с уверенностью считать работоспособным и достоверным.

Системными сбоями называются сбои, которые влияют на все выполняемые в данный момент транзакции, но не наносят физического ущерба самой базе данных. Физический ущерб базе данных может быть нанесен сбоем носителя информации; защитой от этого вида сбоя является регулярное получение резервных копий базы данных и ведение **системного журнала транзакций (transaction log)**. Команды SQL, используемые для этих целей, обсуждаются ниже в этой главе.

В случае системного сбоя механизм восстановления системы базы данных должен выяснить два вопроса.

- Какие транзакции не успели завершиться к моменту сбоя и, следовательно, должны быть отменены.
- Какие транзакции успели завершиться к моменту сбоя, но соответствующая информация еще не переписалась из внутренних буферов системы в саму базу данных (физическую) и, следовательно, должны быть выполнены повторно.

Механизм восстановления в основном скрыт от пользователей, хотя в некоторых системах в распоряжении пользователей имеется такая команда, как CHECKPOINT, которая заставляет систему переписать на диск из своих буферов результаты всех завершившихся транзакций.

Транзакции, определяемые пользователем

Транзакции, определяемые пользователем, позволяют задать системе управления базой данных команду на выполнение любого количества операторов SQL как единого блока. Вид соответствующей команды SQL зависит от конкретной системы. *Начало транзакции может отмечаться определенным оператором (например, BEGIN TRANSACTION или BEGIN WORK) или может быть неявным.* Конец транзакции может включать команды с ключевыми словами COMMIT (для успешно выполненной команды) или ROLLBACK (для отката).

Классической иллюстрацией, когда необходимы транзакции, определяемые пользователем, является ситуация, с которой сталкивается банк, когда его клиент переводит деньги со сберегательного счета на текущий счет. С точки зрения баз данных эта банковская транзакция состоит из двух операций: сначала — обновление сберегательного баланса для отражения дебета, а затем — обновление текущего баланса на основе кредита. Если эти две операции обновления не трактовать как единую транзакцию по отношению к рассматриваемой базе данных, то возникает опасность, что какой-то другой пользователь выполнит запрос в промежутке между ними. Если кто-то из банковских служащих задаст запрос, касающийся баланса данного клиента, после того как деньги будут сняты со сберегательного счета, но до того как они поступят на текущий счет, то результаты этого запроса не будут соответствовать действительности. Объединение же этих двух операций в транзакцию, определяемую пользователем, гарантирует, что перевод денег будет либо завершен полностью, либо не состоится вообще.

Помимо предоставления пользователю контроля над управлением транзакциями, транзакции, определяемые пользователем, повышают общую производительность системы, поскольку избыточные (но неизбежные) действия в системе имеют место не для каждой отдельной команды, а лишь однократно — для транзакции в целом.

Некоторые команды SQL нельзя включать в транзакции, определяемые пользователем. Более подробно об этом вы узнаете из справочных руководств по своей системе.

Работа с транзакциями. Между командами SQL, отмечающими начало и конец транзакций, может помещаться любое количество операторов SQL. В зависимости от синтаксиса, который поддерживает ваша система, это может выглядеть примерно так:

```
[оператор начала транзакции]
  SQL оператор
  SQL оператор
  SQL оператор
оператор конца транзакции
```

Отмена транзакции. Если какую-то транзакцию необходимо отменить до того, как она будет выполнена — либо по причине какого-то сбоя, либо просто потому, что пользователю что-то не понравилось, — результаты выполнения всех ее операторов, которые успели завершиться, должны быть отменены. Транзакцию можно отменить (или вернуть в исходное положение) с помощью команды транзакции ROLLBACK в любой момент до выполнения команды транзакции COMMIT. Можно отменить или всю транзакцию или ее часть (если вы располагаете тем или иным механизмом “точки сохранения”). Однако вы не имеете возможности отменить транзакцию после того, как она будет выполнена.

В большинстве систем синтаксис команды транзакции ROLLBACK выглядит примерно так:

```
[оператор начала транзакции]
  SQL оператор
  SQL оператор
  SQL оператор
оператор отката транзакции
```

Получение резервной копии и восстановление

Процедуры получения резервной копии и восстановления существенно зависят от конкретной системы. Ниже приведен краткий обзор того, как это делается в одной из систем (Sybase). В вашей системе эти вопросы могут решаться совершенно по-другому.

Каждое изменение в базе данных, является ли оно результатом выполнения одного оператора SQL UPDATE (системно-определяемая транзакция) или сгруппированной совокупности операторов SQL (транзакция, определяемая пользователем), автоматически фиксируется в журнале транзакций, который (в системе Sybase) представляет собой таблицу в словаре данных.

Запросы модификации данных (операторы UPDATE, INSERT или DELETE) фиксируются в журнале транзакций по принципу “от момента к моменту”. Когда транзакция начинается, в журнале фиксируется событие начала (BEGIN) транзакции. В момент приема каждого очередного оператора модификации данных он также фиксируется в журнале транзакций.

В системе Sybase изменение фиксируется в журнале транзакций прежде, чем оно произойдет в самой базе данных. Этот тип журнала, называемый журналом с упреждающей записью, гарантирует возможность полного восстановления базы данных в случае сбоя.

После сбоя все транзакции, зафиксированные в журнале, можно выполнить повторно. Команды SQL, предназначенные для получения резервных копий и восстановления баз данных с помощью журналов транзакций, обычно называются DUMP DATABASE, DUMP TRANSACTION, LOAD DATABASE, LOAD TRANSACTION или что-то вроде того. После того как будут заданы соответствующие команды LOAD, система баз данных выполнит процесс восстановления.

ПРОИЗВОДИТЕЛЬНОСТЬ

Производительность является важным вопросом в системах управления реляционными базами данных. Многопользовательские приложения особенно чувствительны к производительности системы: базы данных, функционирующие в этих системах, как правило, отличаются внушительными размерами и сложностью выполняемых в них операций; количество пользователей и транзакций в таких системах предъявляет к ним повышенные требования, а задачи, выполняемые с помощью этих приложений, зачастую выполняются в жестких временных рамках. Однако вопросы производительности привлекают к себе повышенное внимание даже в системах баз данных с единственным пользователем. Никому не нравится подолгу ожидать результатов запроса или выполнения того или иного оператора модификации данных.

К сожалению, многие аспекты производительности неподвластны пользователям. Более того, пользователи зависят от средств и возможностей, встроенных в систему ее проектировщиком. Если вас в данный момент волнует вопрос, покупки системы управления реляционными базами данных, тогда вас особенно должен заинтересовать следующий раздел, в котором кратко обсуждается метод **сравнения с эталоном (benchmarking)**, позволяющий сравнивать производительность различных систем баз данных. Методом сравнения с эталоном можно также пользоваться для оценки влияния на производительность различных проектных решений и индексации.

После обсуждения метода сравнения с эталоном мы коснемся в этом разделе некоторых способов, с помощью которых пользователи могут влиять на производительность.

- Качество проектирования базы данных может существенно сказаться на производительности системы.
- Способ структурирования ваших запросов может повлиять на производительность системы.
- В некоторых системах баз данных предусмотрены средства мониторинга и настройки производительности, включая определенный контроль физической структуры (*где и как* физически хранятся данные).

Производительность — необъятная тема. В этой книге у нас есть возможность лишь слегка коснуться ее, главным образом, чтобы подвести вас к проблемам, которые вам потребуется изучить глубже, когда вопрос производительности станет перед вами со всей остротой.

Сравнение с эталоном

Если вы озабочены вопросом приобретения емкой, производственно-ориентированной системы управления реляционными базами данных, то вам, наверное, уже доводилось слышать заявления разработчиков о числе транзакций в секунду, обрабатываемом их системами. Величины, которые они упоминают, основываются на результатах сравнения с эталоном, или тестах, которые измеряют производительность системы в контролируемой среде на основе той или иной стандартной методологии.

Интерпретация и анализ заявлений, касающихся измерений производительности по методу сравнения с эталоном, как известно, чрезвычайно затруднены. Метод сравнения с эталоном технически сложен, а результаты, объявляемые разработчиками, неизбежно отражают стремление заинтересованной стороны представлять свою продукцию в лучшем свете. Технические подробности метода сравнения с эталоном не входят в задачу данной книги, однако список соответствующей литературы можно найти в конце книги.

Вот несколько “не технических” вопросов, касающихся оценок производительности, полученных по методу сравнения с эталоном, которые наверняка вас заинтересуют.

- Используется ли один из общепринятых стандартных тестов? Или, может быть, проектировщик выбрал данный тест только потому, что его система показывает на этом тесте наилучшие результаты?

- Подтверждаются ли результаты, полученные по методу сравнения с эталоном, какой-либо независимой и авторитетной организацией?
- Похожа ли аппаратная конфигурация, которая использовалась при тестировании по методу сравнения с эталоном, на ту, в которой вы собираетесь эксплуатировать оцениваемую систему? При этом надо учитывать не только тип компьютера, но и конфигурацию его памяти, работал ли он в составе сети и т.п.
- Является ли программное обеспечение, которое использовалось при тестировании по методу сравнения с эталоном, коммерческой версией, или это какая-то специальная/предварительная его версия?
- Являются ли результаты тестирования по методу сравнения с эталоном непротиворечивыми и повторяющимися?

Испытания по методу сравнения с эталоном обычно ассоциируются с системами баз данных, которые будут использоваться для жизненно важных приложений, когда речь идет о десятках и даже сотнях пользователей и очень крупных базах данных (сотни и тысячи мегабайт). Они используются не только для сравнения производительности разных версий, но и для планирования конфигурации технических средств, которые потребуются вам для данного уровня производительности и для принятия решений относительно структуры и индексации базы данных. Можно, например, создать два варианта структуры базы данных и сравнить их производительность с эталоном, определив, какая из них лучше соответствует вашим задачам.

Структура и индексация

Производительность можно существенно улучшить, уделив особое внимание логической структуре базы данных. Важность структуры базы данных для производительности (и не только производительности) опровергает широко распространенное мнение о том, что в случае системы реляционной базы данных от вас требуется указать только то, *что* вы хотите, а не то, *как* это сделать. Эксперт по системам реляционных баз данных Роберт Эпштейн (Robert Epstein) называет это “мифом” и утверждает, что “качество реляционной базы данных целиком и полностью определяется выбранной вами структурой этой базы”.

В главах 2 и 3 поясняются принципы анализа данных и методы нормализации, позволяющие, в конце концов, прийти к качественной, “чистой” структуре базы данных. Из материала этих глав вы, наверное, помните, что индексы существенно ускоряют процесс поиска данных, но в какой-то мере замедляют модификацию данных. Вы должны также помнить, что разбиение таблиц в соответствии с правилами нормализации может привести к замедлению поиска данных, когда запрос, на который можно было бы получить ответ путем просмотра одной таблицы, теперь требует от системы просмотра нескольких таблиц. Многотабличные запросы могут отрицательно сказываться на производительности по нескольким причинам. Поскольку данные оказываются “рассеянными” по нескольким таблицам, для их поиска могут потребоваться дополнительные операции чтения с диска. После того как данные будут найдены, система должна объединить их. Наконец, необходимость доступа к большому числу таблиц влечет за собой необходимость установления дополнительных блокировок.

Таким образом, принимая во внимание фактор производительности, следует в первую очередь рассмотреть, как главным образом будет использоваться проектируемая база данных. Если число запросов к базе данных намного превосходит число операторов модификации данных, постарайтесь создать как можно больше индексов и минимизировать необходимое количество объединений, создавая более крупные таблицы. Точнее, выясните, какие запросы будут выполняться часто, и поместите все столбцы, используемые данными запросами, в одну и ту же таблицу. Если требуется обеспечить минимальное время реакции на операторы модификации данных, используйте как можно меньше индексов и очень внимательно относитесь к нормализации данных.

Запросы

Когда вы запускаете какой-либо оператор SQL в системе управления базой данных, выполняется скрытый от глаз пользователя, но весьма внушительный объем работы. В сущности, SQL считается **непроцедурным языком (non-procedural language)** именно потому, что он не требует от пользователя указывать шаги, которые должна предпринимать система для выполнения данной команды. Вместо этого пользователь просто указывает, что именно требуется найти.

Сила непроцедурного подхода одновременно является и его слабостью. Положительный момент состоит в том, что система делает указанную работу за вас. Плохо, однако, то, что, после того как вы остановили свой выбор на какой-то конкретной структуре базы данных, вам обычно не остается ничего иного, как полагаться на интеллект системы, а не на свой собственный.

Часть системы управления базой данных, отвечающая за анализ операторов SQL, называется **оптимизатором запросов (query optimizer)**. Оптимизатор запросов расчленяет каждый запрос на составные части и реорганизует его с целью оптимизации его выполнения. Он оценивает несколько стратегий поиска или путей доступа, принимая решение относительно использования самых полезных индексов и относительно пути, который требует наименьшего числа обращений к логическим страницам. Другими словами, оптимизатор запросов вырабатывает некоторый план, определяющий наиболее эффективный путь между SQL-оператором пользователя и данными, необходимыми для выполнения поставленной задачи.

Оптимизаторы запросов в различных системах управления реляционными базами данных имеют в своем распоряжении библиотеку стандартных стратегий обработки запросов (например, для использования кластерного индекса столбца), которые применяются в различных ситуациях, возникающих с запросами. Если ни одну из стратегий, известных системе, невозможно применить к анализируемому оператору SQL, система использует стратегию, предусмотренную по умолчанию: читает каждую строку таблицы (или таблиц), указанной в запросе. Такой просмотр всей таблицы обязательно приводит к желаемому результату, но в случае очень крупных таблиц этот процесс может тянуться ужасно долго.

В ранних реализациях не предусматривалось никакой защиты от неэффективного структурирования запросов пользователями. Теперь же, как правило, предусматриваются диагностические средства, помогающие увидеть, как обрабатываются команды. Это позволяет пользователю выполнить необходимые настройки в индексах, структуре таблицы или распределении данных. Оптимизатору запросов зачастую вполне хватает “интеллекта”, чтобы “перефразировать” операторы SQL в формы, выполняющиеся самым эффективным образом.

Некоторые коммерческие системы поддерживают специальные “процедуры”: именованные совокупности операторов SQL, включающие расширения для процедурного кода и входные параметры. (В Transact-SQL эти объекты называются хранимыми процедурами или системными процедурами; они обычно заранее откомпилированы. Такие процедуры обеспечивают реальное повышение производительности и функциональной гибкости.)

Другие инструменты для мониторинга и повышения производительности

В системах управления реляционными базами данных, разработанных для производственно-ориентированных, многопользовательских приложений, часто предусматривается определенный набор средств мониторинга и настройки производительности. Многие из этих средств не имеют никакого отношения к самому SQL; в некоторых системах предполагается, что вы будете выполнять эти задачи с помощью средств операционной системы, а не с помощью средств, предусмотренных в системе базы данных. Ниже приводится краткий перечень средств мониторинга и настройки производительности, который позволит почувствовать возможности, имеющиеся в настоящее время.

Кэш. Конфигурация внутренней памяти вашего компьютера, особенно объем памяти, выделенный для **кэша данных (data cache)** и **буферного кэша (buffer cache)**,

может очень существенно влиять на производительность. В некоторых системах кэш является областью памяти, где хранятся страницы данных и страницы индексов, использовавшиеся самыми последними по времени. Эти системы баз данных всегда сначала именно в кэше пытаются найти нужную им страницу; если нужная страница обнаруживается в кэше, это позволяет экономить время на доступе к диску. Другими словами, хранение часто используемых страниц в кэше позволяет повысить производительность за счет минимизации числа обращений к диску.

Управление кэшем данных обычно осуществляется по принципу “первый пришел — первый ушел”: страницы, использовавшиеся самыми последними, помещаются в кэш, заменяя там страницы, использовавшиеся самыми первыми. Обычно имеет смысл поэкспериментировать с кэшем данных: увеличение его размера может ускорить работу вашего приложения. Некоторые системы, кроме того, позволяют вам настраивать размер страницы памяти для вашего приложения. Чем меньше страницы, тем больше их может поместиться в кэше.

Регистрация. Большинство систем управления реляционными базами данных позволяет вам отключать средство регистрации транзакций базы данных, что ускоряет операции обновления за счет экономии времени на регистрации этих изменений в журнале транзакций. Конечно, в случае сбоя информационного носителя вы не сможете рассчитывать на то, что система базы данных восстановит изменения, внесенные в базу данных, если регистрация транзакций не проводилась.

Имеет смысл временно приостановить регистрацию перед загрузкой в базу данных большого объема информации — если, разумеется, эту операцию нетрудно будет повторить в случае сбоя информационного носителя. Однако, прежде чем приступить к работе, не забудьте получить резервную копию базы данных, с тем чтобы можно было “в случае чего” восстановить ее.

Мониторинг планов выполнения. Проверьте, предусмотрено ли в вашей системе средство мониторинга плана выполнения запроса. Этим средством можно воспользоваться для проверки работы SQL; кроме того, можно увидеть, как изменяется его план, в зависимости от того, куда были помещены индексы и как вы записали соответствующий оператор.

Другие средства мониторинга могут отображать информацию о том, как обрабатывался тот или иной оператор SQL: сколько просмотров таблиц, логических доступов (чтений кэша) и физических доступов (чтений диска) выполняла система для каждой таблицы или курсора, которые были указаны в запросе; сколько страниц было записано по каждой команде модификации данных; сколько процессорного времени занял синтаксический анализ и компиляция каждой команды или выполнение каждого шага команды.

Статистика по индексам. Во многих системах предусмотрен инструмент или утилита, которая позволяет вам более подробно проанализировать функционирование индексов. В ходе такого анализа могут определяться их минимальные и максимальные значения, а также количество различных значений. В Transact-SQL команда UPDATE STATISTICS вычисляет распределение значений ключей индекса в указанном индексе.

Вычисляет ли SQL простую статистику или статистику распределения для определенного индекса, эта информация запоминается для того, чтобы оценить длительность поиска при использовании такого индекса. Сравнивая разные индексы, для которых система накапливает статистику, она может “сознательно” выбрать самый лучший из них.

Статистика, основывающаяся на распределении значений ключей, дает лучшую оценку времени поиска, чем простая статистика, которая предполагает, что значения ключей индекса распределены равномерно. Если вы используете версию SQL, которая использует статистику распределения, то важно не забывать о необходимости повторного выполнения команды UPDATE STATISTICS после того, как вы добавите, удалите или модифицируете совокупность новых данных, поскольку эти действия обычно влияют на распределение ключей индекса. Если информацию о распределении ключей вовремя не обновлять, система будет пользоваться старой информацией и может сделать неудачный выбор в отношении того, как выполнять тот или иной запрос.

У вас могут быть и другие команды, позволяющие вам влиять на установку индексов. Например, опции индексации, предусмотренные в Transact-SQL (FILLFACTOR, MAX_ROWS_PER_PAGE), позволяют вам управлять величиной заполнения каждой страницы нового индекса. Это значение влияет на производительность из-за времени, которое требуется системе на разбивку индексных страниц, когда они заполняются на 100 процентов.

Другой причиной использования средств типа FILLFACTOR является необходимость физического распределения данных по небольшим, но часто используемым таблицам, что приводит к уменьшению числа коллизий. Допустим, какая-то важная таблица занимает лишь пару страниц данных, а блокировка выполняется на уровне страниц. Определенные кластерного индекса с несколькими строками на каждой странице распределяет данные по большому числу страниц и снижает конкуренцию за блокировки.

Недостатком малых значений коэффициента заполнения (FILLFACTOR) является то, что каждый индекс занимает больше места в памяти, что снижает общую производительность системы.

ЦЕЛОСТНОСТЬ ДАННЫХ

В широком смысле, целостность данных означает точность и непротиворечивость данных в базе данных. В идеале, в программном обеспечении базы данных должен быть предусмотрен ряд механизмов для проверки целостности данных; к сожалению, несколько важных типов целостности не поддерживаются большинством реляционных систем.

На практике многие требования к целостности часто удовлетворяются с помощью специализированных прикладных программ. К недостаткам переложения задачи контроля целостности на прикладные программы относится необходимость выполнения дополнительной работы, связанной с написанием и реализацией кода для проверки целостности, вероятность дублирования работы и появление несовместимости, когда несколько приложений пользуются одной и той же базой данных, и та легкость, с которой пользователи, имеющие доступ к соответствующей базе данных, могут обойти ограничения, запрограммированные в приложениях.

Существует несколько видов целостности данных. На самом базовом уровне все системы баз данных (не только реляционные) должны гарантировать, что тип данных вводимого значения является правильным и что это значение входит в диапазон значений, поддерживаемых системой. В разных реляционных системах предусмотрены разные наборы типов данных, но во всех них вводимые значения проверяются, а оператор модификации данных отвергается, если введенное значение не соответствует указанному типу данных. Нулевой статус столбца также проверяется при вводе данных. Наконец, определенные типы данных — обычно символьные типы — могут (или должны) соответствовать длинам, указанным пользователем. Некоторые системы отвергают ввод данных, который превышает максимальную длину, предусмотренную для этого типа данных, другие “обрезают” введенное значение так, чтобы оно соответствовало максимально допустимой длине.

Тремя другими видами целостности, которые обсуждаются в этом разделе, являются:

- ограничения на домен
- целостность объекта
- ссылочная целостность

Ограничения на домен

Домен (domain) представляет собой совокупность логически связанных значений, из которой может быть получено значение в определенном столбце. Вот несколько примеров доменов в базе данных *bookbiz*.

- Доменом столбца *authors.au_id* являются все коды социального страхования, выдаваемые правительством США.

- Доменом столбцов *authors.city* и *publishers.city* являются все города США, а для столбцов *authors.state* и *publishers.state* — все штаты США. (Мы предполагаем, что все авторы проживают в США.)
- Доменом *titles.type* является следующая совокупность значений: *business*, *popular_comp*, *psychology*, *mod_cook* и *trad_cook*.
- Доменом *titles.title_id* является совокупность значений со следующим форматом: первые два символа — прописные буквы алфавита из набора *BU*, *PC*, *PS*, *MC*, *TC*; следующие четыре символа — цифры от 0 до 9 включительно.
- Доменом *titleauthors.royaltyshare* являются все значения в интервале от 0 до 1 включительно.

Обратите внимание на разные виды логических взаимосвязей между значениями в этих доменах. Некоторые из доменов представляют ограничения, определенные в приложениях, — т.е. правила и нормы бизнеса. Например, ограничения на формат чисел идентификатора названия были определены кем-то в издательской компании. Там же, наверное, решили, что цена на книги не может быть меньше \$1.99 и больше \$99.99; таким образом, цены в долларах и центах, укладываемые между этими двумя значениями, и будут представлять собой домен для *titles.price*.

Другие домены основываются не на бизнес-правилах, а на физических или математических ограничениях. Значения в *titleauthors.royaltyshare*, например, представляют собой доли, поэтому они должны укладываться в диапазон от 0 до 1. Другой пример: допустим, издателю потребовалось фиксировать пол каждого автора в базе данных. Домен для этого столбца был бы ограничен человеческой биологией (ее широко распространенными интерпретациями), т.е. значениями *мужчина*, *женщина* и *неизвестно*.

Описания доменов в приведенном выше списке были получены на основе анализа значений в базе данных *bookbiz*. Для выражения многих из них можно воспользоваться ограничением CHECK в операторе CREATE TABLE. Transact-SQL поддерживает дополнительный механизм для указания доменов — команду CREATE RULE. Правилom называется именованный объект базы данных, который может быть ассоциирован с любым числом столбцов или со всеми столбцами указанного типа данных. Подобно ограничению CHECK, определение правила может включать любое выражение, которое допускается в предложении WHERE (арифметические операторы, операторы сравнения, LIKE, IN, BETWEEN и т.д.). Эта гибкость в определении правил позволяет вам в качестве основы доменов указывать списки значений (подобно домену для *titles.type*), диапазоны (подобно домену для *titleauthors.royaltyshare*) или формат (подобно домену для *titles.title_id*).

Механизм правил Transact-SQL ограничен, однако, в том отношении, что определение правил не может содержать ссылку на другой столбец в базе данных. Однако определение правил вне оператора CREATE TABLE удобно для часто меняющихся правил.

И, наконец, последнее замечание. Вспомните материал главы 7, в котором утверждалось, что если значения в двух столбцах имеют одни и те же домены, объединения между этими столбцами обычно являются логически оправданными. Например, *authors.city* и *publishers.city* имеют один и тот же домен (все города США); следовательно, имело бы смысл воспользоваться их объединением.

Целостность объекта

Целостность объекта требует, чтобы ни один из компонентов первичного ключа не мог иметь нулевого значения, т.е. одностолбцовый первичный ключ не может содержать нули (то же можно сказать и о любом из столбцов в составном первичном ключе).

Ограничение целостности объекта является следствием самой реляционной модели, а не требований какого-либо конкретного приложения. Кроме того, проблема целостности объекта не присуща другим моделям управления базами данных (чего не скажешь о доменных ограничениях). Требование, чтобы ни один из первичных ключей не содержал нулевого значения, основывается на том, что объекты

реального мира отличаются друг от друга первичными ключами, которые выполняют роль уникальных идентификаторов.

Когда вы проектируете свою базу данных, то должны гарантировать целостность объекта назначением такого первичного ключа, который не будет принимать нулевых значений. Этого можно добиться, используя ключевое слово `NOT NULL` в операторе `CREATE TABLE` и воспользовавшись ограничением `PRIMARY KEY` или создав для соответствующего столбца уникальный индекс. (Обзор этих понятий приведен в главах 2 и 3.)

Ссылочная целостность

Неформально говоря, ссылочная целостность относится к взаимосвязи между значениями в логически связанных таблицах. В реляционной модели это означает гарантирование логической непротиворечивости базы данных за счет обеспечения постоянного соответствия значений первичного ключа и связанных с ним внешних ключей.

Вот определение, данное Коддом: “Для каждого отдельного ненулевого значения внешнего ключа в реляционной базе данных должно существовать соответствующее значение первичного ключа из того же домена”.

В главе 2 поясняется, что взаимосвязи между внешними и первичными ключами устанавливаются в ходе проектирования базы данных; они отражают логические взаимосвязи между данными (хотя их наличие никоим образом не ограничивает возможные пути доступа к данным). При рассмотрении ссылочной целостности вопрос заключается в том, что именно может сделать система базы данных, чтобы гарантировать поддержание соответствия между значениями внешних ключей и значениями первичного ключа, на который они указывают (т.е. обеспечить соблюдение ссылочных ограничений). В главе 3 рассматривались ограничения `REFERENCES` и `FOREIGN KEY` в операторе `CREATE TABLE`. Эти предложения гарантируют определенную ссылочную целостность — они позволяют вам обеспечить проверки, которые предотвращают добавление внешнего ключа, если он не соответствует первичному ключу. Но это лишь один аспект ссылочной целостности.

Например, изменение идентификатора автора может представлять определенную проблему, поскольку такое изменение нарушило бы связь между таблицами *authors*, *titles* и *titleauthors*. Предложение `REFERENCES` предотвращает этот вид изменения во внешнем ключе (*titleauthors.au_id*), но не в первичном ключе (*authors.au_id*). Как быть с первичным ключом при изменении идентификатора в таблице *authors* либо при удалении или изменении идентификатора издателя в таблице *publishers*, когда книги в таблице *titles* по-прежнему ссылаются на старый идентификатор? Один из подходов заключается в принятии решения о том, что первичные ключи никогда не должны изменяться, и в предотвращении их изменения.

Еще один возможный ответ — автоматически “каскадировать” операцию обновления или удаления в отношении соответствующих внешних ключей. Если, например, изменяется идентификационный номер издателя, система должна точно таким же образом изменить соответствующие идентификаторы в *titles.pub_id* без вмешательства со стороны пользователя.

Третья возможность состоит в том, чтобы согласиться с операциями модификации данных в отношении первичного ключа, даже если они нарушают ссылочную целостность, но перед этим изменить значения соответствующих внешних ключей на `NULL`. (Разумеется, если внешний ключ был определен таким образом, чтобы не принимать нулевых значений, этот подход не сработает.) Воспользуемся приведенным выше примером, слегка видоизменив его: в ответ на команду `DELETE`, которая удаляет строку, соответствующую *Algodata Infosystems*, из таблицы *publishers*, значения *titles.pub_id* для всех книг, опубликованных *Algodata*, будут установлены в `NULL`.

Кратко подведем итоги. Вообще говоря, существует три возможные реакции на попытку удалить или модифицировать первичный ключ, на который указывает внешний ключ.

- **Ограничить** — операция удаления или модификации в отношении первичного ключа отвергается, если имеются соответствующие ему значения внешних ключей.

- **Каскадировать** — операция удаления или модификации автоматически применяется к внешним ключам, значения которых соответствовали “старому” значению удаленного или модифицированного первичного ключа.
- **Обнулить** — до выполнения операции удаления или модификации в отношении первичного ключа значения (значение) соответствующих внешних ключей (ключа) устанавливаются в NULL.

К сожалению, многие диалекты SQL не предусматривают механизмов управления ссылочной целостностью первичных ключей. В этих случаях максимум, что вы можете сделать для обеспечения ссылочной целостности, — это аннулировать все разрешения на удаление и обновление столбца первичных ключей.

Однако по мере все более широкого признания важности понятия “ссылочная целостность” разработчики пытаются обеспечивать ее путем ввода специальных процедур или триггеров. В качестве одного из примеров можно назвать Transact-SQL. Изложение подробностей триггерного механизма, предусмотренного в Transact-SQL, не входит в задачи настоящей книги, но описание, приведенное в следующем разделе, даст вам некоторое представление о возможности использования триггеров для обеспечения ссылочной целостности.

Триггеры. Триггер в Transact-SQL представляет собой именованную совокупность операторов SQL, описывающую действие, которое необходимо выполнить при попытке выполнения указанной операции модификации данных в отношении определенного столбца или таблицы.

Триггеры действуют автоматически. Они работают независимо от того, что именно является причиной модификации данных, — ввод данных, выполненный человеком-оператором, действие той или иной прикладной программы или получение отчета. Каждый триггер предназначен для одной или нескольких операций модификации данных: UPDATE, INSERT или DELETE.

Создание триггера включает указание команды модификации данных, которая “запускает” этот триггер, таблицы, которая является его “целью”, и действия (или действий), которое должен выполнить триггер. Ниже приведен синтаксис оператора CREATE TRIGGER:

```
CREATE TRIGGER имя_триггера
ON имя_таблицы
FOR {INSERT | UPDATE | DELETE}
    [, {INSERT | UPDATE | DELETE}]...
AS SQL_оператор
    [IF UPDATE (имя_столбца)
    [{AND | OR} UPDATE (имя_столбца)]...]
```

Предложение ON задает имя таблицы, которая активизирует этот триггер (так называемая триггерная таблица). Предложение FOR указывает, какая команда (команды) модификации данных активизирует триггер.

Операторы SQL в предложении AS определяют **триггерные действия** и **триггерные условия**. Триггерные действия могут включать любое количество операторов SELECT.

Триггерные условия могут указывать дополнительные критерии, которые определяют, приведут ли к выполнению триггерного действия (действий) попытки выполнения команд INSERT, DELETE или UPDATE. Триггерные условия часто включают подзапрос, которому предшествует ключевое слово IF. Предложение IF UPDATE (имя_столбца) проверяет, был ли модифицирован указанный столбец; при этом триггерные действия ассоциируются с выполнением изменений в конкретных столбцах.

В операторах CREATE TRIGGER используются имена двух логических (или концептуальных) таблиц — *deleted* и *inserted*. Когда на ту или иную триггерную таблицу выдается команда DELETE (например, делается попытка удалить значения первичных ключей), удаляемые строки изымаются из триггерной таблицы и переносятся в таблицу *deleted*. Затем триггер может проанализировать строки в таблице *deleted*, чтобы определить, должны ли (а если должны, то как) выполняться соответствующие триггерные действия (действие).

Когда делается попытка выполнения команды INSERT или UPDATE, строки, представляющие новые значения, одновременно добавляются в триггерную таблицу и в таблицу *inserted*. Строки в таблице *inserted* могут быть затем проанализированы триггерным механизмом.

Ниже приведен пример триггера, который выполняет каскадирование, когда удаляется значение первичного ключа в таблице *titles*. Когда в отношении таблицы *titles* выполняется оператор DELETE, триггер удаляет соответствующую строку из таблицы *titles* и добавляет ее в таблицу *deleted*. Затем он проверяет таблицы *titleauthors*, *salesdetails* и *roysched*, выясняя, нет ли в них строк с внешним ключом *title_id*, который соответствует значению *title_id*, удаленному из таблицы *titles* (теперь это значение хранится в таблице *deleted*). Если триггер обнаружит подобные строки, то удалит их.

SQL (расширение):

```
create trigger delcascadetrig
on titles
for delete
as
delete titleauthors
from titleauthors, deleted
where titleauthors.title_id = deleted.title_id
      /* Удаление строк из таблицы titleauthors,
      соответствующих удаленным строкам из таблицы titles */

delete salesdetails
from salesdetails, deleted
where salesdetails.title_id = deleted.title_id
      /* Удаление строк из таблицы salesdetails,
      соответствующих удаленным строкам из таблицы titles */

delete roysched
from roysched, deleted
where roysched.title_id = deleted.title_id
      /* Удаление строк из таблицы roysched,
      соответствующих удаленным строкам из таблицы titles */
```

Ниже приведен еще один пример триггера контролирующего ссылочную целостность: он предотвращает модификации столбца первичных ключей таблицы *titles* в определенные дни недели. (Такой триггер может понадобиться для любого типа столбца, а не только для столбца первичных ключей.)

Предложение IF UPDATE “фокусирует” этот триггер на определенном столбце — *titles.title_id*. Попытки модификации этого столбца заставляют триггер переходить к действиям, отменяя операцию модификации и распечатывая соответствующее сообщение.

SQL (расширение):

```
create trigger stopupdatetring
on titles
for update
as
if update (title_id)
  and datename (dw, getdate ())
  in ("Saturday", "Sunday")
begin
  rollback transaction
  print "В выходные изменять значения первичного ключа запрещено!"
end
      /* Если titles.title_id изменяется в субботу
      или в воскресенье, то предотвратить его обновление*/
```

Как только что было показано, триггеры Transact-SQL часто используются для контроля ссылочной целостности, которой угрожает выполнение операций модификации и удаления первичного ключа. Однако, в сущности, триггеры представляют собой более обобщенный метод решения проблем целостности.

- Триггеры могут налагать ограничения, более сложные, чем ограничения, определяемые правилами, CHECK-ограничениями или REFERENCES-ограничениями. Триггер, например, может отвергать операции модификации, связанные с увеличением цены книги более чем на 1 процент от авансового гонорара за нее, или препятствовать любым повышениям цены более чем на 100 процентов.
- Триггеры можно использовать для пересчета текущих значений. Например, у вас может быть триггер, который обновляет столбец *ytd_sales* в таблице *titles* всякий раз, когда в таблице *salesdetails* добавляется новая строка.

ОТ АБСТРАКЦИЙ SQL К РЕАЛЬНОМУ МИРУ

Эта глава приблизила обсуждение SQL к понятиям реального мира, затронув такие темы, как производительность, управление транзакциями и средства обеспечения ссылочной целостности. Мы не будем вдаваться в подробное обсуждение этих вопросов, поскольку системы управления реляционными базами данных весьма существенно отличаются друг от друга тем, как в них реализованы средства получения резервных копий и восстановления, управление транзакциями и стратегии доступа.

В следующих двух главах мы вернемся ко многим из тех операторов SQL, с которыми вы уже знакомы, представив более сложные фрагменты кода.

Разрешение проблем

КАК ИСПОЛЬЗОВАТЬ SQL В СВОЕЙ РАБОТЕ

По большому счету, все, что вы знаете об SQL, полезно только в той мере, в какой эти знания помогают вам в выполнении вашей конкретной работы. SQL — это инструмент бизнеса.

Основная задача предыдущих глав заключалась в том, чтобы ознакомить вас с возможностями SQL. Теперь же настало время узнать, как SQL используется на практике — для выполнения конкретной работы. Большая часть материала для этой главы была собрана на основе дискуссий (иногда горячих!) в Internet. Пользователи, у которых возникали те или иные вопросы по SQL, “забрасывали” их в Internet, часто сопровождая свои вопросы примерами кодов и какими-то дополнительными описаниями, в которых содержались просьбы о помощи. Те, кто мог хоть чем-то помочь, охотно откликнулись на такие просьбы, дополняли предыдущие ответы (или категорически не соглашались с ними) и время от времени высмеивали отправителей этих вопросов за то, что те даже не потрудились как следует обдумать возникшую проблему.

Вот из этих-то вопросов и ответов мы составили перечень самых распространенных тем. Определенные вопросы (например, как форматировать результаты) возникали снова и снова. Стала очевидной настоятельная потребность в своего рода “книге кулинарных рецептов”, которая содержала бы примеры кодов и давала, таким образом, ответы на самые распространенные вопросы.

В настоящей книге описываются обобщенные или “промышленные” версии SQL, а не какие-то конкретные их реализации. Однако если уж мы решили давать “рецепты кодов”, от которых будет хоть какая-то практическая польза, нам волею-неволею придется вдаваться в конкретные детали, не говоря уж о том, что любые такие “рецепты” желательно протестировать. Transact-SQL (Sybase) — это диалект SQL, с которым мы знакомы лучше всего. Именно поэтому мы решили воспользоваться им в своих примерах, проверив их как на Sybase SQL Server, так и на SQL Anywhere. Если же вы используете какую-то другую систему баз данных, вам может понадобиться модифицировать кое-какие “рецепты” из нашей “кулинарной книги”. Тем не менее мы старались избегать вопросов, слишком уж “привязанных” к Sybase (храняемая процедура или код триггера).

Вы не найдете здесь и обсуждения вопросов индексации и производительности. Они слишком “привязаны” к конкретным диалектам SQL и конкретному окружению, чтобы имело смысл их иллюстрировать изолированными фрагментами кода. Кроме того, было решено оставить без внимания различные “религиозные” аргументы (относящиеся к “правильным” и “неправильным” реализациям всевозможных функций), поскольку они совершенно не соответствуют практической ориентации этой книги.

Другими словами, эта глава представляет собой собрание небольших фрагментов максимально обобщенного кода SQL. Каждый из них основывается на реальном вопросе, который встречался в Internet. Проблемы и решения были адаптированы для работы с уже известным вам примером базы данных *bookbiz*. Ничего “ужасно трудного” вы здесь не встретите. Тем не менее вы встретите здесь много такого, обо что “сломали зубы” многие “типичные” пользователи. В конце концов, мы руководствовались допущением: если у кого-то возникла такая проблема, то она вполне может возникнуть и у вас.

Итак, пользуйтесь этой главой как книгой кулинарных рецептов по SQL, но постарайтесь внести в этот процесс хоть какой-то элемент собственного творчест-

ва: находя интересные рецепты, не забудьте добавить в свое блюдо какие-нибудь специи, которые помогут улучшить его вкус.

Вот общий перечень обсуждаемых тем.

- Форматирование и отображение данных.
- Поиск данных (в основном по шаблонам).
- Использование сложных объединений и подзапросов.
- Работа с предложением GROUP BY.
- Создание последовательных номеров.

ФОРМАТИРОВАНИЕ И ОТОБРАЖЕНИЕ ДАННЫХ

Вид, в котором вы сохраняете данные, не всегда наилучшим образом подходит для их отображения. Если совместно с вашей базой данных работают какие-либо прикладные программы (например, генераторы отчетов), они могут выполнять все необходимое вам форматирование. Иногда, однако, имеет смысл воспользоваться строковыми, числовыми и функциями даты, которые реализуют в своих продуктах большинство разработчиков баз данных. Многие из них стали частью стандарта ANSI SQL в 1992 г.

Но не забывайте: SQL — это не средство отображения! SQL предназначен для поиска данных, а вовсе не для того, чтобы представить их на экране в каком-нибудь вычурном формате. Если некоторые рецепты, приведенные в этом разделе, покажутся вам несколько сложными, так это потому, что они такие и есть.

В этом разделе вы встретите коды, которые позволяют выполнять следующее.

- Отображать одно поле в виде двух.
- Выравнивать строку символов по правому краю.
- Задавать число разрядов после десятичной точки.

Отображение одного поля в виде двух

Иногда данные, хранящиеся в базе данных в виде одного поля, представляются пользователям более осмысленными, если их представить в виде двух и более отдельных полей. Это утверждение особенно справедливо, когда такое поле состоит из разнотипных данных, как иногда бывает в некоторых составных полях идентификации (например, код страхования здоровья может состоять из хорошо знакомого девятизначного номера социального страхования плюс алфавитный код компании или географический код).

В таблице *titles* поле *title_id* очень напоминает описанную выше ситуацию. Оно содержит шесть символов: две буквы, за которыми следуют четыре цифры. Вот пример соответствующих данных (предложение WHERE служит для ограничения количества возвращаемых данных):

SQL:

```
select title_id
from titles
where price > $19.99
```

Результат:

```
title_id
-----
PC8888
PC1035
TC3218
PS1372
```

Чтобы отобразить буквенную часть поля отдельно от его численной части, в операторе SELECT можно воспользоваться строковой функцией. После того как данные примут вид, который вам требуется, создайте курсор. Это даст возможность пользователям видеть разделенные поля вместо составных.

Прежде всего с помощью строковой функции SUBSTRING, которая возвращает часть символьной или двоичной строки, надо определить две части поля. Эта функция имеет следующий синтаксис:

```
SUBSTRING(выражение, начало, конец)
```

Вы можете познакомиться с тем, как работает функция SUBSTRING, выполнив оператор SELECT для поля *title_id* и присваивая каждому фрагменту этого поля новое имя:

```
SQL:
select substring(title_id, 1, 2) as alpha,
       substring(title_id, 3, 4) as num
from titles
where price > $19.99
```

Результат:

alpha	num
PC	8888
PC	1035
TC	3218
PS	1372

А теперь скройте эту работу от пользователей, создав соответствующий курсор. При выполнении запросов они не будут даже подозревать о том, что *alpha* и *num* фактически являются частями поля *title_id*. Оператор CREATE VIEW выглядит следующим образом (как и в предыдущем случае, предложение WHERE включается лишь для того, чтобы ограничить число возвращаемых строк):

```
SQL:
create view split
as
select substring(title_id, 1, 2) as alpha,
       substring(title_id, 3, 4) as num
from titles
where price > $19.99
```

После того как вы аннулируете разрешение пользоваться таблицей *titles* и предоставите права на использование данного курсора, пользователи смогут увидеть указанные два поля, использовав следующий запрос:

```
SQL:
select *
from split
```

Результат:

alpha	num
PC	8888
PC	1035
TC	3218
PS	1372

Конечно, *num* по-прежнему является символьным полем, поэтому вы не сможете выполнять с ним математические операции. Запрос, подобный приведенному ниже, возвращает ошибку (если, разумеется, ваша система не умеет находить среднеарифметическое символьных значений):

SQL:

```
select avg(num)
from split
```

У вас есть возможность разрешить эту проблему, создав курсор, в котором *num* будет интерпретироваться как целочисленный тип данных. Воспользуйтесь курсором *split* с одним важным дополнением — функцией CONVERT, которая преобразует один тип данных в другой. Синтаксис этой функции следующий:

```
CONVERT (тип_данных [длина], выражение)
```

Чтобы применить эту функцию к столбцу *num*, заменяя его тип данных на целочисленный (integer), нужно записать:

```
convert(int, num)
```

Но при этом надо помнить, что *num* является, в сущности, подстрокой (SUBSTRING) поля *title_id*. Чтобы завершить работу, подставьте в функции CONVERT вместо *num* соответствующее выражение SUBSTRING.

```
convert(int, substring(title_id, 3, 4))
```

Полностью запрос теперь примет следующий вид:

SQL:

```
select substring(title_id, 1, 2) as alpha,
convert(int, substring(title_id, 3, 4)) as num
from titles
where price > $19.99
```

Если вы создадите другой курсор с помощью этого варианта запроса, то сможете воспользоваться для поля *num* числовыми функциями, такими как AVG:

SQL:

```
select avg(num)
from split2
```

Результат:

```
-----
3628
```

В зависимости от того, как ваша система обрабатывает среднеарифметические и целочисленные значения, вы можете получить другой результат.

Выравнивание строки символов по правому краю

Теперь перейдем к более сложной задаче. Как выполняется выравнивание строк символов по правому краю? Простейший (и самый разумный) способ заключается в использовании программы генерации отчетов или какой-либо специальной программы отображения. Если ни того, ни другого у вас нет, или указанный способ не подходит вам по какой-то другой причине (например, вы хотите воспользоваться SQL “из принципа”), попытайтесь воспользоваться следующим рецептом.

Ниже приведен пример отображения по умолчанию (с выравниванием по левому краю) имени и фамилии в таблице *authors*. Предложение WHERE ограничивает выдаваемые на печать результаты только теми авторами, первой буквой имени которых является *A*.

SQL:

```
select au_fname, au_lname
from authors
where au_fname like 'A%'
```

Результат:

au_fname	au_lname
Abraham	Bennet
Albert	Ringer
Anne	Ringer
Ann	Dull
Akiko	Yokomoto

Чтобы выровнять эти данные по правому краю, вам придется выполнить следующую процедуру для каждого поля имени.

1. Найти определенный размер столбца (в операторе CREATE TABLE) и фактический размер данных, хранящихся в этом столбце.
2. Для каждого столбца вычислить разницу между определенным размером столбца и фактическим размером данных.
3. Поместить слева от каждого значения количество пробелов, которое соответствует вычисленной разнице.
4. Добавить данные справа от пробелов; сумма пробелов слева и фактические данные справа дадут в итоге определенный размер столбца.

Большинство разработчиков реализуют в своих системах тот или иной способ получения информации об определенной длине столбца и фактической длине данных. В Transact-SQL можно воспользоваться двумя системными функциями: COL_LENGTH и DATALENGTH. Вот их синтаксис:

```
COL_LENGTH('имя_объекта', 'имя_столбца')
```

```
DATALENGTH(выражение)
```

Подробнее об этих функциях вы можете узнать из справочного руководства по своей системе.

Ниже приведен пример использования этих функций:

SQL:

```
select au_fname,
       col_length('authors', 'au_fname') as col,
       datalength(au_fname) as data,
       col_length('authors', 'au_fname')
       datalength(au_fname) as diff
from authors
where au_fname like 'A%'
```

Результат:

au_fname	col	data	diff
Abraham	20	7	13
Albert	20	3	17
Anne	20	5	15
Ann	20	4	16
Akiko	20	6	14

В таблице результатов представлены длина столбца, фактический размер данных и разница между длиной столбца и длиной данных. Первая функция возвращает размер столбца, тогда как вторая находит количество символов в каждом столбце. Разница между этими двумя числами равна количеству пробелов, которые следует поместить с левой стороны поля, чтобы “протолкнуть” имена в конец поля и выровнять их, таким образом, по правому краю.

Теперь, когда вам известен размер данных и разница между ним и размером столбца, следующим вашим шагом должно стать заполнение поля нужным числом ведущих пробелов. REPLICATE представляет собой функцию Transact-SQL, предназначенную для вставки повторяющихся символов. Вот ее синтаксис:

```
REPLICATE(char_expr, integer_expr)
```

В этом примере *char_expr* соответствует пробелу (“ ”), а *integer_expr* представляет собой количество повторений *char_expr*. В качестве *integer_expr* следует использовать разницу между *datalength* и *col_length*. Вот как все это вместе взятое будет выглядеть на данный момент:

```
SQL:
select replicate(' ', col_length ('authors', 'au_fname')
- datalength(au_fname)) as Personal_name,
      replicate(' ', col_length('authors', 'au_lname')
- datalength(au_lname)) as Surname
from authors
where au_fname like 'A%'
```

Результат:

```
Personal_name      Surname
-----
```

Такой запрос выполняется без проблем, но все, что вы получите, — это заголовки столбцов и... пустой экран! Все, что вы сделали, — это было лишь заполнить пробелами слева; чтобы увидеть на экране нужную информацию, надо просто добавить данные.

Присоедините их, воспользовавшись функцией конкатенации. В Transact-SQL для этого надо воспользоваться знаком “плюс” (+). Синтаксис чрезвычайно прост:

выражение + выражение

Теперь получается следующий код:

```
SQL:
select replicate(' ', col_length('authors', 'au_fname')
- datalength(au_fname)) + au_fname as Personal_name,
      replicate (' ',
col_length('authors', 'au_lname')
- datalength(au_lname)) + au_lname as Surname
from authors
where au_fname like 'A%'
```

Эта версия обеспечивает вам заголовки столбцов и данные, выровненные по правому краю. Однако эти поля слишком велики. Чтобы оставить лишь то, что действительно нужно, добавьте функцию CONVERT для управления размером отображения:

```
SQL:
select convert(varchar (20),
replicate (' ',
col_length('authors', 'au_fname')
```

```

- datalength(au_fname))
+ au_fname) as Personal_name,
      convert(varchar (40),
      replicate(' ',
col_length('authors', 'au_lname')
- datalength(au_lname))
+ au_lname) as Surname
from authors
where au_fname like 'A%'

```

Personal_name	Surname
Abraham	Bennet
Ann	Dull
Akiko	Yokomoto
Anne	Ringer
Albert	Ringer

(Из-за проблем со шрифтами и выравниванием в некоторых системах столбцы не получаются столь же превосходно выровненными, как в приведенном выше примере. Если у вас возникнут подобные проблемы, отправьте результаты в файл формата ASCII и проанализируйте их в этом формате.)

Как указать число разрядов после десятичной точки

Отображение в формате с плавающей точкой, предусмотренное по умолчанию, не всегда устраивает пользователей. Требуемая точность (общее количество разрядов) и масштаб (число разрядов после десятичной точки) могут привести к очень широкому формату отображения. Кроме того, приблизительная природа чисел с плавающей точкой может служить причиной отображения малопонятных результатов. Если ваша цель — представить числа в удобном формате (даже если при этом придется пожертвовать точностью результатов), можно воспользоваться функциями, управляющими отображением.

Вот как данные были введены в таблицу, содержащую столбец с типом данных с плавающей точкой:

one
22.1
1.1
4444.20
22.22
333.1

А вот как выполняется отображение на одной из систем, в которой SQL убирает все нули справа:

```

SQL:
select *
from testfloat

```

Результат:

one
22.1
1.1
4444.2

22.22

333.1

В другой системе можно увидеть примерно следующее (а поскольку речь идет о типе данных с плавающей точкой, фактические значения на разных машинах будут несколько различаться):

Результат:

one

22.1000003814697

1.10000002384186

4444.2001953125

22.2199993133545

333.100006103516

Во многих системах имеется функция ROUND (применяемая только для числовых типов данных). Она позволяет указать количество разрядов справа или слева от десятичного знака. При этом используется следующий синтаксис:

ROUND (числовой_столбец, целое_число)

Значение *целое_число* указывает количество разрядов справа (положительное значение) или слева (отрицательное значение) от десятичной точки. Код для ограничения отображения двумя разрядами после десятичной точки мог бы иметь следующий вид:

SQL:

```
select round(one, 2)
from testfloat
```

Результат:

rounded

22.1

1.1

4444.2

22.22

333.1

Единственная проблема в этом случае состоит в том, что убираются нули справа. Другой подход заключается в использовании совокупности строковых функций. Функция SUBSTRING (совместно с CONVERT, поскольку *one* является числовым типом данных) не может выполнить эту задачу:

SQL:

```
select short = substring(
                    convert(varchar (18), one),
                    1,5), one
from testfloat
```

Результат:

short

one

22.10

22.1000003814697

1.100

1.10000002384186

4444.

4444.2001953125

22.21
333.1

22.2199993133545
333.100006103516

Результаты в поле *short* показывают, что вы получаете первые пять разрядов данных безотносительно того, что именно они собой представляют.

Чтобы контролировать число разрядов после десятичной точки, вы должны согласовать третий аргумент в функции SUBSTRING (количество разрядов, подлежащих отображению) с местоположением десятичной точки. Вам не требуется в данном случае какое-то абсолютное число (как, например, 5 в предыдущем запросе), а что-то вроде выражения “все разряды по левую сторону от десятичной точки и два разряда по правую сторону от десятичной точки”.

Большинство систем баз данных располагают функцией, которая находит положение одной строки в другой. В Transact-SQL такой функцией является CHARINDEX. Вот ее синтаксис:

```
CHARINDEX(expression1, expression2)
```

Здесь *expression1* представляет собой подмножество (строка, которую вы ищете), а *expression2* представляет собой полное множество (строка, в которую “встроено” указанное подмножество).

В нашем случае *expression1* представляет собой десятичную точку (.), а *expression2* представляет собой поле *one*. Поскольку поле *one* будет использоваться внутри функции SUBSTRING, оно должно быть преобразовано в символьный тип данных. Если все эти элементы соединить вместе, то функция CHARINDEX примет следующий вид:

```
charindex(".", convert(varchar (18), one))
```

Добавьте 2 и подставьте все это в качестве третьего аргумента функции SUBSTRING. Вот как теперь будет выглядеть запрос и результаты его выполнения:

```
SQL:
select short = substring(
    convert(varchar (18), one),
    1, charindex
(".", convert(varchar (18), one)) +2), one
from testfloat
```

Результат:

short	one
22.10	22.1000003814697
1.10	1.10000002384186
4444.20	4444.2001953125
22.21	22.2199993133545
333.10	333.100006103516

Если теперь заменить +2 на +1, после десятичной точки будет отображаться только один разряд. В качестве примера попытайтесь выровнять эти числа по правому краю.

РАБОТА С ШАБЛОНАМИ

Приведенные выше рецепты связаны с поиском данных, когда в предложении WHERE выполняется множество действий, но объединения при этом были достаточно простыми. В следующем разделе речь пойдет о более сложных объединениях.

Первые три запроса выполняют поиск данных по принципу сопоставления с соответствующими шаблонами. В каждом из этих случаев используются разные подходы. Четвертый запрос весьма своеобразен. Он обнаруживает противоречи-

вые сочетания NULL и пустых строк. После этого он очищает их с помощью оператора UPDATE.

Знакомясь с последующим материалом, вы узнаете, как найти следующее.

- Символьные данные, когда вы не уверены в том, как они представлены (прописными, строчными или разнотипными буквами).
- Символьные строки заданной длины.
- Данные типа дат.
- Нулевые данные, маскирующиеся под пробелы.

Сопоставление прописных и строчных букв

Иногда данные хранятся в виде прописных букв (“COMPUTER”), иногда — в виде строчных букв (“computer”), а иногда — в виде разнотипных букв (“Computer”). Это может отражать отсутствие стандартов на момент ввода данных и, следовательно, отсутствие проверок на целостность.

Допустим, к примеру, что вы хотите определить, как именно называется книга, заглавие которой звучит то ли как *Life Without Fear*, то ли как *Life without Fear*. Коротче говоря, вы не помните точно, как в данном случае пишется слово “without” — с прописной или строчной буквы. Существует ряд способов, помогающих решить данную проблему.

Можно воспользоваться ключевым словом LIKE, как в следующем примере:

SQL:

```
select title
from titles
where title
      like '% [Ww] [Ii] [Tt] [Hh] [Oo] [Uu] [Tt]%'
```

Результат:

```
title
-----
Life Without Fear
```

Каждая пара букв, заключенная в квадратные скобки, допускает сопоставления для прописных и строчных букв.

Другим вариантом является использование функции UPPER. Она преобразует строчные буквы поля в прописные буквы. Вот ее синтаксис:

UPPER(символьное_выражение)

После того как данные будут преобразованы в прописные буквы, LIKE сравнивает их с соответствующим шаблоном:

SQL:

```
select title
from titles
where upper(title) like '%WITHOUT%'
```

Такую запись легче читать и понимать, чем сам по себе синтаксис LIKE. В противоположность функции UPPER, функция LOWER преобразует прописные буквы поля в строчные буквы. Выясните, как в вашей системе с помощью этих функций обрабатываются слова с разнотипными буквами. В некоторых системах требуется двукратное использование функции UPPER (или LOWER) для учета любой комбинации прописных и строчных букв:

SQL:

```
select title
from titles
where upper(title) like upper ("%wiTHout%")
```

Может показаться, что простейшим ответом на проблему строчных и прописных букв является ввод информации буквами какого-то одного типа, который позволил бы избежать всей описанной выше работы. Однако не обольщайтесь раньше времени! Вы можете потерять важную информацию, которая потребуется вам в дальнейшем. Внутри многих фамилий используются как строчные, так и прописные буквы. Рассмотрим, например, фамилии Blotchett-Halls, DeFrance, O'Leary, MacFeather и del Castillo, которые встречаются в базе данных *bookbiz*. Вы, вероятно, видели их записанными с необычными местоположениями прописных букв. Если вы сохранили их в виде только строчных (или только прописных) букв, то вам может впоследствии понадобиться восстановить их в первоизданном виде, который вы совершенно сознательно нарушили.

Поиск символьных данных заданного размера

По-видимому, найти символьные данные какой-то определенной длины не представляет большой проблемы. Однако даже этот относительно простой случай имеет несколько интересных нюансов. Например, запрос на поиск всех авторов, имеющих четырехбуквенные имена, с использованием шаблонов диапазона в Transact-SQL (квадратных скобок, заключающих в себе символы диапазона строчных или прописных букв, разделенные дефисом) мог бы выглядеть следующим образом:

SQL:

```
select au_fname
from authors
where au_fname like '[A-Z] [a-z] [a-z] [a-z]'
```

Результат:

```
au_fname
-----
Dick
Burk
Dirk
Anne
```

Проверьте, есть ли в вашей системе подобные функции.

Такой запрос будет работать удовлетворительно до тех пор, пока имена авторов будут начинаться с прописной буквы и содержать только алфавитные символы. Чтобы в базе данных можно было отыскивать имена вроде "A-Po" и "L'to", вы должны будете применить следующий запрос:

SQL:

```
select au_fname
from authors
where au_fname like ' _ _ _ _ '
```

Результат:

```
au_fname
-----
Dick
Burk
Dirk
Anne
```

Символ подчеркивания (_) означает совпадение с *любым* символом. Однако результаты запроса будут зависеть от конкретной реализации вашей базы данных; могут, например, быть найдены имена, содержащие менее четырех букв, поэтому результаты могут выглядеть примерно так:

Результат:

```
au_fname
-----
Anne
Dirk
Dick
Ann
Burt
```

Это происходит в тех случаях, когда сравниваются две строки, и более короткая строка дополняется пробелами для выравнивания с более длинной строкой. Чтобы избежать возможной неоднозначности, воспользуйтесь кодом, подобным приведенному ниже. Здесь символ “^” означает отрицание, поэтому [^] означает “не пробел”.

SQL:

```
select au_fname
from authors
where au_fname like '[^ ][^ ][^ ][^ ]'
```

Результат:

```
au_fname
-----
Anne
Dirk
Dick
Burt
```

Теперь у вас есть список, каждое имя в котором состоит из четырех символов (за исключением пробела). Он не включает трехбуквенные имена или имена из трех букв и пробела.

А как найти авторов, имена которых *не* являются четырехбуквенными? Одним из вариантов решения этой задачи является вариант запроса, обратный приведенному выше:

SQL:

```
select au_fname
from authors
where au_fname not like '[^ ][^ ][^ ][^ ]'
```

Из базы данных *bookbiz* будет возвращено девятнадцать строк.

Как найти данные типа дат

Информацию о дате бывает нелегко найти из-за множества возможных способов ее хранения. В Transact-SQL поля даты включают месяц, день, год и время дня (первые три из этих полей можно отобразить несколькими способами).

Это создает определенные проблемы при поиске временной части поля по методу совпадения с шаблоном. В качестве примера можно привести следующий запрос:

SQL:

```
select price, pubdate
from titles
where pubdate = 'Oct 21 1985'
```

Результат:

```
price          pubdate
-----
21.59          Oct 21 1985 12 : 00AM
20.95          Oct 21 1985 12 : 00AM
```

Этот запрос предназначен для поиска записей, датированных 21-м октября 1985 г. (время 12:00). Чтобы получить все записи за это число с *любым* временем, надо внести дополнительный код. Для проверки этих решений изменим сначала одну из дат:

```
SQL:
update titles
set pubdate = 'Oct 21 1985 2:30PM'
where pubdate = 'Oct 21 1985' and price = $20.95
update titles
```

Теперь запрос возвращает только одну строку:

```
SQL:
select price, pubdate
from titles
where pubdate = 'Oct 21 1985'
```

Результат:

price	pubdate
21.59	Oct 21 1985 12 : 00AM

Довольно простой метод поиска всех строк, введенных в один день, заключается в предоставлении исчерпывающего описания: вы указываете минимальное и максимальное значения времени для интересующего вас дня и используете ключевое слово BETWEEN для поиска всех записей, удовлетворяющих этим критериям:

```
SQL:
select price, pubdate
from titles
where pubdate
    between 'Oct 21 1985 00:00'
        and 'Oct 21 1985 23:59'
```

Результат:

price	pubdate
21.59	Oct 21 1985 12:00AM
20.95	Oct 21 1985 2:30PM

Другая идея состоит в использовании ключевого слова LIKE для поиска всего, что совпадает с известной частью даты. В данном случае шаблон “знак процента” (%) обозначает все, что следует за той частью даты, которая соответствует месяцу, дню и году:

```
SQL:
select price, pubdate
from titles
where pubdate like 'Oct 21 1985%'
```

Кроме того, можно воспользоваться функцией CONVERT и преобразовать дату в более короткую символьную строку, а затем выподнить поиск этой строки:

```
SQL:
select price, pubdate
from titles
where convert(char (11), pubdate) = 'Oct 21 1985'
```

Другой подход связан с использованием функций даты. Выясните, имеются ли подобные функции в вашей системе. В приведенном ниже примере используется функция DATEPART, которая сравнивает каждую известную часть (месяц, день и год) полного значения даты с ее численным представлением (октябрь соответствует 10). Временная часть даты значения уже не имеет. Синтаксис функции DATEPART:

DATEPART (часть_даты, дата)

Вот как выглядит такой запрос:

SQL:

```
select price, pubdate
from titles
where datepart(mm, pubdate) = 10 and
datepart(dd, pubdate) = 21 and
datepart(yy, pubdate) = 1985
```

DATEADD — еще одна функция подобного типа. Используйте ее, чтобы добавить один день к известной дате. Затем вы можете отыскать все данные, удовлетворяющие заданным параметрам, не заботясь о времени, как показано в следующем примере.

SQL:

```
select price, pubdate
from titles
where pubdate between 'Oct 21 1985' and
dateadd(day, 1, 'Oct 21 1985')
```

Этот запрос позволяет найти все записи, помеченные датами от 10/21/85 и до 10/22/85 (т.е. на один день позже).

Замена пробелов на нули

Пользователь может создать таблицу, допускающую нули, когда какое-то значение пропущено или неизвестно. Другие пользователи, менее искушенные во всевозможных “реляционных хитростях”, могут не понять ваших намерений, старательно вводя пробелы, вместо того чтобы дать возможность базе данных самой вставить, там, где это необходимо, значение NULL. Результатом всего этого будут “грязные” данные: в каких-то местах — строки пробелов, в других местах — нули. Если единственными правильными вариантами являются либо значимые данные, либо NULL, вам будет необходимо преобразовать эти строки пробелов в нули.

В более длительной перспективе имеет смысл продумать способ введения определенных жестких правил работы. Это может означать использование специальных правил, триггеров или предварительных анализаторов, которые предотвращали бы появление пробелов там, где необходимы нули. Если же говорить о ближайших задачах, то вам надо каким-то образом избавиться от пробелов.

Например, кто-то добавил двух новых авторов в таблицу *authors*. Их имена и идентификационные номера введены правильно, но информация о телефонах и адресах неполна — это просто строки пробелов. (Все адресные поля — *city, state, zip* и *phone* — допускают нулевые значения.)

Если вы не имеете ничего против примера с известной вам базой данных, вставьте следующие две строки в таблицу *authors*:

SQL:

```
insert authors
values ('123-45-6789', 'Wu', 'Amelia', ' ',
      ' ', ' ', ' ', ' ', ' ')
insert authors
values ('123-54-6789', 'Khandasamy', ' J.', ' ',
      ' ', ' ', ' ', ' ', ' ')
```

Если выполнить оператор SELECT по отношению к полям имени, телефонного номера и адреса (укороченным с помощью функции SUBSTRING для удобочитаемости), то будут получены следующие результаты:

Результат:

au_fname	phone	address	city	state	zip
Amelia					
J.					

Как сделать, чтобы эти строки были совместимы с остальной частью таблицы и отобразить NULL там, где нужная информация пропущена или неизвестна? Функция RTRIM в Sybase Transact-SQL удаляет “хвостовые” пробелы (те, которые находятся с правой стороны). Когда в поле нет ничего, кроме пробелов, и в этом поле допускается значение NULL, функция RTRIM удаляет пробелы, а система (которая явно испытывает отвращение к вакууму) вставляет NULL. Синтаксис функции RTRIM следующий:

RTRIM(символьное_выражение)

Вясните, есть ли в вашей системе подобная функция и поступает ли она с нулевыми значениями так же, как было описано выше (следует отметить, что не во всех системах это делается одинаково).

Вот как выглядит оператор UPDATE с (таким вариантом) функции RTRIM:

SQL:

```
update authors
set phone = rtrim(phone),
    address = rtrim(address)
city = rtrim(city),
state = rtrim(state),
zip = rtrim(zip)
where au_id like '123%'
```

Этот запрос позволяет получить следующие результаты:

SQL:

```
select au_fname, phone, address, city, state, zip
from authors
where au_id like '123%'
```

Результат:

au_fname	phone	address	city	state	zip
Amelia	NULL	NULL	NULL	NULL	NULL
J.	NULL	NULL	NULL	NULL	NULL

Если же поля, которые вы “подстригаете” подобным образом, не допускают значений NULL, система не будет вставлять NULL и выдаст сообщение об ошибке — но это уже совсем другая проблема. В нашем примере мы предполагаем, что со значением NULL у вас не возникнет проблем.

Чтобы привести в порядок все свои данные в процессе преобразования пустых полей, воспользуйтесь функцией LTRIM для удаления ведущих (левосторонних) пробелов. (Напоминаем, что функция RTRIM удаляет “хвостовые” пробелы.) Синтаксис функции LTRIM аналогичен синтаксису RTRIM. Например, имя г-на Khandasamy (“J.”) было вставлено в таблицу с двумя ведущими пробелами. Их можно удалить, добавив следующие строки кода:

SQL:

```
update authors
set au_fname = ltrim(au_fname)
```

Полученные результаты показывают, что оба новых имени выровнены по левому краю (ведущие пробелы отсутствуют):

```
SQL:
select au_fname, phone, address, city, state, zip
from authors
where au_id like '123%'
```

Результат:

au_fname	phone	address	city	state	zip
Amelia	NULL	NULL	NULL	NULL	NULL
J.	NULL	NULL	NULL	NULL	NULL

А вот как можно было бы скомбинировать операции избавления значимых данных от излишних пробелов и преобразования пустых строк в нули:

```
SQL:
update authors
set au_fname = rtrim(ltrim (au_fname)),
    phone = rtrim(ltrim (phone)),
    address = rtrim(ltrim (address)),
    city = rtrim(ltrim (city)),
    state = rtrim(ltrim (state)),
    zip = rtrim(ltrim (zip))
```

Прежде чем продолжить работу, удалите две строки, которые вы добавили в таблицу *authors*, с помощью оператора DELETE:

```
SQL:
delete authors
where au_id like '123%'
```

ПОИСК ДАННЫХ С ПОМОЩЬЮ СЛОЖНЫХ ОБЪЕДИНЕНИЙ И ПОДЗАПРОСОВ

В этом разделе вы найдете другие рекомендации по составлению запросов, общей темой которых является использование более сложных объединений и подзапросов. Здесь вы найдете коды для решения следующих задач.

- Сопоставление пар полей в разных таблицах (с помощью внешнего объединения или подзапроса).
- Нахождение строк в определенном диапазоне, когда вам неизвестно ни верхнее, ни нижнее значение (с подзапросами и без них).
- Отображение данных в формате электронной таблицы (с помощью коррелированных подзапросов в предложении SELECT).

Сопоставление пар столбцов в разных таблицах

Если у вас есть две таблицы с подобными данными (например, поля *city* и *state* в таблицах *authors* и *publishers*), то как выяснить, встречаются ли одинаковые пары значений в обеих таблицах?

Этим кодом удобно пользоваться при выполнении проверок на целостность данных после их копирования из одной таблицы в другую и в тех случаях, когда вы храните подмножества большой таблицы в отдельных, более мелких таблицах с целью повышения эффективности операций поиска.

Ниже изложен метод, в котором используется внешнее объединение. Как вы, наверное, помните, внешнее объединение может быть левым (*=) или правым (=*).

Левое внешнее объединение отображает все выделенные поля в первой таблице объединения и только совпадающие поля — во второй. Правое внешнее объединение выполняет обратную операцию: оно отображает совпадающие поля в первой таблице и все выделенные поля — во второй. Проще всего запомнить это по тому, что символ “звездочка” (*) обозначает “все”. Поэтому таблица со стороны “звездочки” в объединении (правом или левом) является таблицей, дающей полные результаты. Результаты внешнего объединения очень удобно анализировать визуально.

SQL:

```
select authors.city acity, authors.state astate, publishers.city pcity,
publishers.state pstate
from authors, publishers
where authors.city != publishers.city
      and authors.state != publishers.state
```

Результат:

acity	astate	pcity	pstate
Menlo Parc	CA	NULL	NULL
Berkeley	CA	Berkeley	CA
Berkeley	CA	Berkeley	CA
Oakland	CA	NULL	NULL
Oakland	CA	NULL	NULL
Oakland	CA	NULL	NULL
Salt Lake City	CA	NULL	NULL
Salt Lake City	CA	NULL	NULL
Gary	IN	NULL	NULL
Rockville	MD	NULL	NULL
Vacaville	CA	NULL	NULL
Oakland	CA	NULL	NULL
Palo Alto	CA	NULL	NULL
Palo Alto	CA	NULL	NULL
Walnut Creek	CA	NULL	NULL
San Jose	CA	NULL	NULL
Covelo	CA	NULL	NULL
Nashville	TN	NULL	NULL
Ann Arbor	MI	NULL	NULL
San Francisco	CA	NULL	NULL
Corvallis	OR	NULL	NULL
Lawrence	KS	NULL	NULL

Другой метод, основанный на использовании ключевого слова EXISTS, позволяет найти ту же информацию: какие пары city/state из таблицы *authors* встречаются и в таблице *publishers*?

SQL:

```
select authors.city, authors.state
from authors
where exists
(select *
 from publishers
 where publishers.city = authors.city
       and publishers.state = authors.state)
```

Результат:

city	state
Berkeley	CA
Berkeley	CA

Если вы хотите увидеть пары из таблицы *authors*, которые не встречаются в таблице *publishers*, используйте конструкцию NOT EXISTS.

Поиск данных в определенном диапазоне, если вам не известны точные значения

Чтобы найти все книги с ценами между \$7.00 и \$10.95, можно выполнить запрос, подобный следующему:

SQL:

```
select title, title_id, price
from titles
where price between $7.00 and $10.95
```

Результат:

title	title_id	price
Emotional Security: A New Algorithm	PS7777	7.99
Is Anger the Enemy?	PS2091	10.95
Life Without Fear	PS2106	7

Если вы хотите увидеть все книги с ценами, находящимися в диапазоне между ценами каких-то двух конкретных книг, но вам не известны цены этих книг, попробуйте выполнить следующий запрос:

SQL:

```
select title, title_id, price
from titles
where price between
    (select price
     from titles
     where title like 'Life Without%')
and
    (select price
     from titles
     where title like 'Is Anger%')
```

Пока вы помещаете книгу с более низкой ценой сначала в предложение BETWEEN, все будет нормально. Но если вы разместите их в убывающей последовательности (поскольку не знаете их настоящих цен), то получите в результате нулевые строки или сообщение об ошибке. Одним из способов избежать этой ситуации является использование оператора OR и включение обеих комбинаций:

SQL:

```
select title, title_id, price
from titles
where price between
    (select price
     from titles
     where title like 'Life Without%')
and
    (select price
```

```

        from titles
        where title like 'Is Anger%')
or price between
    (select price
     from titles
     where title like 'Is Anger%')
and
    (select price
     from titles
     where title like 'Life Without%')

```

Еще один способ, который охватывает оба эти случая, заключается в нахождении минимальной и максимальной цен. Как вариант, можно воспользоваться идентификационными номерами книг вместо их названий:

```

SQL:
select title, title_id, price
from titles
where price between
    (select min(price)
     from titles
     where title_id in ('PS2106', 'PS2091'))
and
    (select max(price)
     from titles
     where title_id in ('PS2106', 'PS2091'))

```

Без подзапросов этот код мог бы выглядеть примерно так:

```

SQL:
select t1.title, t1.title_id, t1.price
from titles t1, titles life, titles anger
where life.title like 'Life%'
    and anger.title like 'Is Anger%'
    and (t1.price between life.price and
        anger.price or t1.price between
        anger.price and life.price)

```

В запрос можно было бы включить и оператор OR, поскольку вы не знаете, какая из цен выше. Скобки гарантируют правильную работу OR (они нужны, чтобы ограничить количество возвращаемых строк, поскольку в этом трехтабличном запросе отсутствуют объединения).

Отображение данных в формате электронной таблицы

Таблица *titleauthors* соединяет авторов и названия книг. Поскольку у каждой книги может быть один или несколько авторов, эта таблица содержит двадцать пять строк для семнадцати книг: по одной строке для каждой комбинации автор-название. Десять книг имеют одного автора, шесть книг имеют двух авторов и одна книга имеет трех авторов.

Для отображения значений *title_id* с первым и вторым авторами в одной строке можно воспользоваться следующим коррелированным подзапросом:

```

SQL:
select distinct title_id,
(select au_id
 from titleauthors

```

```

        where au_ord = 1 and title_id = t.title_id) as au1,
(select au_id
    from titleauthors
    where au_ord = 2 and title_id = t.title_id) as au2,
from titleauthors t

```

Этот запрос дает следующие результаты:

Результат:

title_id	au1	au2
BU1032	409-56-7008	213-46-8915
PS7777	486-29-1786	NULL
PC9999	486-29-1786	NULL
MC2222	712-45-1867	NULL
PS3333	172-32-1176	NULL
PC1035	238-95-7766	NULL
BU2075	213-46-8915	NULL
PS2091	998-72-3567	899-46-2035
PS2106	998-72-3567	NULL
MC3021	722-51-5454	899-46-2035
TC3218	807-91-6654	NULL
BU7832	274-80-9391	NULL
PC8888	427-17-2319	846-92-7186
PS1372	756-30-7391	724-80-9391
BU1111	724-80-9391	267-41-2394
TC7777	672-71-3249	267-41-2394
TC4203	648-92-1872	NULL

В зависимости от того, как этот вид подзапроса обрабатывается в вашей системе, может оказаться, что ключевое слово DISTINCT вам не потребуется.

Единственную строку с тремя авторами можно отобразить, добавив в предложение SELECT третий коррелированный подзапрос:

SQL:

```

select distinct title_id
    (select au_id
        from titleauthors
        where au_ord = 1 and title_id = t.title_id) as au1,
    (select au_id
        from titleauthors
        where au_ord = 2 and title_id = t.title_id) as au2,
    (select au_id
        from titleauthors
        where au_ord = 3 and title_id = t.title_id) as au3,
from titleauthors t

```

Результат:

title_id	au1	au2	au3
BU1032	409-56-7008	213-46-8915	NULL
PS7777	486-29-1786	NULL	NULL
PC9999	486-29-1786	NULL	NULL

MC2222	712-45-1867	NULL	NULL
PS3333	172-32-1176	NULL	NULL
PC1035	238-95-7766	NULL	NULL
BU2075	213-46-8915	NULL	NULL
PS2091	998-72-3567	899-46-2035	NULL
PS2106	998-72-3567	NULL	NULL NULLNULL
MC3021	722-51-5454	899-46-2035	NULL
TC3218	807-91-6654	NULL	NULL
BU7832	274-80-9391	NULL	NULL
PC8888	427-17-2319	846-92-7186	NULL NULNULL
PS1372	756-30-7391	724-80-9391	NULL
BU1111	724-80-9391	267-41-2394	NULL
TC7777	672-71-3249	267-41-2394	472-27-2349
TC4203	648-92-1872	NULL	NULL

Еще один способ получения того же результата заключается в использовании новой таблицы и трех запросов, каждый из которых соответствует первому, второму и третьему авторам. Если соответствующая ячейка пуста, можно добавить комментарий типа "none". В приведенном ниже примере используется временная таблица (хотя она может быть и постоянной):

SQL:

```
create table #au_order
(title_id char (6),
au1 char (11),
au2 char (11),
au3 char (11))
```

Во-первых, воспользуемся оператором INSERT для добавления всех строк с первыми (или единственными) авторами:

SQL:

```
insert #au_order
select title_id, au_id, 'none', 'none'
from titleauthors
where au_ord = 1
```

Вот какой результат позволяет получить выполнение оператора SELECT для данной таблицы:

Результат:

title_id	au1	au2	au3
BU1032	409-56-7008	none	none
PS7777	486-29-1786	none	none
PC9999	486-29-1786	none	none
MC2222	712-45-1867	none	none
PS3333	172-32-1176	none	none
PC1035	238-95-7766	none	none
BU2075	213-46-8915	none	none
PS2091	998-72-3567	none	none
PS2106	998-72-3567	none	none
MC3021	722-51-5454	none	none

TC3218	807-91-6654	none	none
BU7832	274-80-9391	none	none
PC8888	427-17-2319	none	none
PS1372	756-30-7391	none	none
BU1111	724-80-9391	none	none
TC7777	672-71-3249	none	none
TC4203	648-92-1872	none	none

Всего имеется семнадцать строк, каждая с *au_id* в столбце первого автора и с *none* — в столбцах второго и третьего авторов.

Теперь обновим таблицу *#au_order* значениями для столбца второго автора:

```
SQL:
update #au_order
set au2 = au_id
from titleauthors
where au_ord = 2
      and titleauthors.title_id = #au_order.title_id
```

Изменились семь строк:

Результат:

title_id	au1	au2	au3
BU1032	409-56-7008	213-46-8915	none
PS7777	486-29-1786	none	none
none	486-29-1786	none	none
MC2222	712-45-1867	none	none
none	172-32-1176	none	none
none	238-95-7766	none	none
BU2075	213-46-8915	none	none
PS2091	998-72-3567	899-46-2035	none
PS2106	998-72-3567	none	none
MC3021	722-51-5454	899-46-2035	none
TC3218	807-91-6654	none	none
BU7832	274-80-9391	none	none
PC8888	427-17-2319	846-92-7186	none
PS1372	756-30-7391	724-80-9391	none
BU1111	724-80-9391	267-41-2394	none
TC7777	672-71-3249	267-41-2394	none
TC4203	648-92-1872	none	none

Наконец, чтобы заполнить столбец третьего автора, воспользуемся оператором UPDATE:

```
SQL:
update #au_order
set au3 = au_id
from titleauthors
where au_ord = 3
      and titleauthors.title_id = #au_order.title_id
```

Запрос к *#au_order* позволяет получить окончательный результат:

```
SQL:
select *
from #au_order
```

Результат:

title_id	au1	au2	au3
BU1032	409-56-7008	213-46-8915	none
PS7777	486-29-1786	none	none
none	486-29-1786	none	none
MC2222	712-45-1867	none	none
none	172-32-1176	none	none
none	238-95-7766	none	none
BU2075	213-46-8915	none	none
PS2091	998-72-3567	899-46-2035	none
PS2106	998-72-3567	none	none
MC3021	722-51-5454	899-46-2035	none
TC3218	807-91-6654	none	none
BU7832	274-80-9391	none	none
PC8888	427-17-2319	846-92-7186	none
PS1372	756-30-7391	724-80-9391	none
BU1111	724-80-9391	267-41-2394	none
TC7777	672-71-3249	267-41-2394	472-27-2349
TC4203	648-92-1872	none	none

ПРЕДЛОЖЕНИЕ GROUP BY

Предложение GROUP BY порождает столько вопросов и сомнений, что большинство примеров, найденных в Internet, приведены в следующей главе. Там вы найдете полное описание GROUP BY и его “напарника”, предложения HAVING. В этом разделе вы встретите только один фрагмент кода. Он связан с использованием GROUP BY для поиска данных по времени.

Отображение данных по времени

Предложение GROUP BY является ответом на следующую интересную проблему: как получить временную статистику? Например, каково распределение по месяцам книг, опубликованных в течение года? Проверьте, есть ли в вашей системе функция наподобие DATEPART, которая отображает указанную часть даты (например, месяц или год) на основании полной даты.

```
SQL:
select datepart(month, pubdate), count(title_id)
from titles
group by datepart(month, pubdate)
```

Результаты отображают группу книг, опубликованных в течение шестого месяца:

Результат:

NULL	2
6	13
10	3

Для удобочитаемости добавим заголовки столбцов:

SQL:

```
select datepart(month, pubdate) as month#,
count(title_id) as books
from titles
group by datepart(month, pubdate)
```

Результат:

month#	books
NULL	2
6	13
10	3

ПОСЛЕДОВАТЕЛЬНЫЕ НОМЕРА

Последовательные номера выполняют несколько важных функций: взять хотя бы их значение для отслеживания счета-фактуры, проверки выполнения заказов и идентификации служащих. Но поскольку последовательные номера относятся к таким сложным вопросам, как индексы и блокировка, оптимальный способ их генерации и использования не столь очевиден.

Некоторые разработчики баз данных реализуют эту возможность. В Oracle предусмотрено средство Sequencer, в Informix — тип данных SERIAL, а Sybase и Microsoft обеспечивают свойство IDENTITY.

Если в вашей системе базы данных не предусмотрена возможность автоматической нумерации (или это средство не подходит для вашего приложения), в вашем распоряжении имеется ряд других методов.

В этом разделе вы узнаете о нескольких возможных способах добавления последовательных номеров в поле *sonum*. Они включают нахождение текущего максимального значения *sonum*, добавление 1 и сохранение этого максимального значения в отдельной таблице. Кроме того, мы рассмотрим одну из альтернатив последовательным номерам — использование уникального произвольного значения.

Для простоты во всех последующих примерах *sonum* используется в первых пяти строках таблицы *sales*. Если вы хотите самостоятельно проверить все приведенные здесь примеры, сначала удалите все строки *sales* со значением *sonum* выше 5. (Эти строки можно сохранить в другой таблице.)

SQL:

```
delete
from sales
where sonum > 5
```

Вот как выглядят эти данные:

SQL:

```
select * from sales
```

Результат:

sonum	stor_id	ponum	sdate
1	7066	QA7442.3	Sep 13 1985 12 : 00AM
2	7067	D4482	Sep 14 1985 12 : 00AM
3	7131	N914008	Sep 14 1985 12 : 00AM
4	7131	N914014	Sep 14 1985 12 : 00AM
5	8042	423LL922	Sep 14 1985 12 : 00AM

Нахождение максимального значения и добавление 1

Самый “интуитивный” подход заключается в нахождении текущего максимального значения, добавлении к нему 1 и вставке результата в таблицу *sales*:

SQL:

```
insert sales
  select max(sonum) + 1, '8042', '66', 'Apr 22 1993'
  from sales
```

Теперь данные примут следующий вид:

SQL:

```
select *
from sales
```

Результат:

sonum	stor_id	ponum	sdate
1	7066	QA7442.3	Sep 13 1985 12 : 00AM
2	7067	D4482	Sep 14 1985 12 : 00AM
3	7131	N914008	Sep 14 1985 12 : 00AM
4	7131	N914014	Sep 14 1985 12 : 00AM
5	8042	423LL922	Sep 14 1985 12 : 00AM
6	8042	66	Apr 22 1993 12 : 00AM

Этот подход годится для простых приложений. Совсем другое дело, если в вашей системе одновременно работает множество пользователей. Как предотвратить одновременный доступ к этой таблице двух пользователей? Уникальный индекс для *sonum*, конечно, помогает, но за это придется расплачиваться снижением производительности.

Один способ увеличить ваши шансы на успех заключается в том, чтобы поместить в транзакцию оператор INSERT:

SQL:

```
begin transaction
insert sales
  select max(sonum) + 1, '8042', '66', 'Apr 22 1993'
  from sales
commit transaction
```

Если в вашей системе базы данных обеспечивается механизм блокировки, контролируемой пользователем (например, HOLDLOCK, как в Transact-SQL), поэкспериментируйте с ним. Это может обеспечить и дополнительную защиту от возможности генерации дублирующихся ключей, когда база данных используется особенно интенсивно.

SQL:

```
begin transaction
insert sales
  select max(sonum) + 1, '8042', '66', 'Apr 22 1993'
  from sales holdlock
commit transaction
```

Конфликт между двумя и более пользователями по-прежнему возможен. В зависимости от того, как реализована ваша система, не исключено, что вам потребуются написать код для обработки ошибок и взаимоблокировок.

Использование отдельной таблицы ключей

Еще один метод заключается в использовании отдельной таблицы для записи максимального значения последовательного поля (в нашем случае — *sonum*). Это уменьшает вероятность “соревнований” за захват таблицы и опасность взаимоблокировки. Сначала создайте таблицу, которая будет содержать текущий максимальный последовательный номер:

SQL:

```
create table maxnum
(sonum int)
```

Поскольку вы будете обновлять таблицу (изменяя существующие значения), то должны инициализировать ее с помощью какого-то значения (например, 1):

SQL:

```
insert maxnum
values(1)
```

```
select *
from maxnum
```

Результат:

```
sonum
-----
1
```

Теперь можно использовать транзакцию для нахождения максимального значения *sonum* в таблице *sales*, сохранить это значение в *maxnum*, а затем обновить таблицу *sales* (запросы включены только для того, чтобы показать содержимое этих двух таблиц в ходе процесса):

SQL:

```
begin transaction
    insert maxnum
        select max(sonum) + 1
        from sales
    insert sales
        select max(maxnum.sonum), 'test', 'test',
        'May 1 1993'
        from maxnum
    select * from sales
commit transaction
```

Результат:

```
sonum
-----
7
```

sonum	stor_id	ponum	sdate
1	7066	QA7442.3	Sep 13 1985 12 : 00AM
2	7067	D4482	Sep 14 1985 12 : 00AM
3	7131	N914008	Sep 14 1985 12 : 00AM
4	7131	N914014	Sep 14 1985 12 : 00AM
5	8042	423LL922	Sep 14 1985 12 : 00AM
6	8042	66	Apr 22 1993 12 : 00AM
7	test	test	May 1 1993 12 : 00AM

Использование произвольного значения

Оба предыдущих метода — поиск с помощью функции *max(sonum)* в целевой таблице и получение этого значения из отдельной таблицы ключей — имеют недостатки. Первый, как отмечалось, может привести к конфликтам между пользователями. Второй может превратиться в “узкое место”: для каждого INSERT нужен INSERT, SELECT и снова INSERT.

Этой проблемы можно избежать, воспользовавшись произвольным значением, таким как дата или время. Если в вашей системе предусмотрено использование временной отметки, это может оказаться именно тем, что вам нужно. Другой подход заключается в генерации такого числа: используйте результат формулы, которая вычисляет какое-либо произвольное или уникальное значение.

Этот “произвольный” подход имеет свои собственные проблемы.

- Если вам действительно необходим последовательный номер, этот метод не удовлетворит ваших потребностей.
- Выполнение сортировки может оказаться весьма накладным, поскольку данные не упорядочены.

Если вы планируете генерировать уникальные идентификационные номера, сравните недостатки, присущие этому методу, с потребностями вашего приложения.

КАК ИЗБЕЖАТЬ ОШИБОК

В следующей главе также используется материал из Internet. В ней перечислены некоторые распространенные ошибки с пояснениями, почему они возникают, и рекомендациями, как их избежать.



Ошибки, и как их избежать

НЕТ, ВЫ НЕ ДУРАК

SQL — это необычный компьютерный язык: он не предназначен для профессионалов. Однако настоящих знатоков SQL по-прежнему совсем немного. Многие пользователи SQL являются на самом деле либо программистами на С, либо бухгалтерами, либо специалистами по маркетингу. Они изучают SQL по собственной инициативе и делают это исключительно практическим путем. Эти пользователи приобретают определенную квалификацию, оставаясь при этом лишь *случайными* пользователями (т.е. работают с SQL лишь от случая к случаю). Когда они сталкиваются с какой-то серьезной трудностью, у них нет возможности обратиться за помощью к специалисту по SQL, сидящему где-нибудь в соседнем отделе. Поэтому им часто приходится обращаться в Internet за помощью к таким же случайным пользователям, как они сами, в надежде, что кто-то из них уже сталкивался с подобной проблемой и нашел способ ее решения.

Давайте признаем тот очевидный факт, что SQL не очень простой язык, и при его использовании у вас может возникнуть немало вопросов. Примеры кодов, приведенные в этой (как и в предыдущей) главе, основаны на проблемах, взятых из реальной жизни, которые нам удалось почерпнуть из Internet. Эти проблемы спроецированы на уже хорошо знакомую вам базу данных *bookbiz*. Только в этом случае они приведены вовсе не для того, чтобы вы скопировали их себе или следовали им “один к одному”, поскольку они представляют собой иллюстрации самых распространенных ошибок и заблуждений.

Интересно отметить, что опять повторяются те же самые темы.

- Использование предложения GROUP BY для подсчета элементов.
- Непонимание того, когда следует пользоваться предложением WHERE, а когда — предложением HAVING.
- Неопределенность по поводу того, как использовать ключевое слово DISTINCT со столбцами и агрегирующими функциями.
- Непонимание истинных возможностей SQL.

Если вы предпочитаете практический подход к изучению SQL, то вам, несомненно, покажется очень полезным изучение примеров ошибок, часто допускаемых другими пользователями, и анализ правильных методов решения указанных проблем. Не забывайте: вы не одиноки. Если судить по количеству вопросов по SQL, встречающихся в Internet, то многие даже весьма квалифицированные программисты на SQL имеют проблемы в указанных областях, но при этом не боятся задавать вопросы.

В этой главе вы не найдете исчерпывающих пояснений и полных синтаксических диаграмм. Поэтому когда вам встретится какая-то тема, по которой вам понадобится более подробная информация, обратитесь к предыдущим главам этой книги или к справочным руководствам по вашей системе.

ПРЕДЛОЖЕНИЕ GROUP BY

GROUP BY — это чрезвычайно эффективный элемент грамматики SQL. Тем не менее многие пользователи и не подозревают о его существовании, прибегая к весьма сложным манипуляциям для вычисления количества элементов, принадлежащих заданным группам. В то время как некоторые пользователи забывают о существовании предложения GROUP BY, другие просто теряются от множества ограничений, связанных с ним, или запутываются в различных диалектах.

Подсчет по единицам

Один страдалец обратился в Internet за помощью в связи со следующей проблемой. Код для поиска количества названий, связанных с определенным издательством, был достаточно прост:

SQL:

```
select count(*)
from titles
where pub_id = '1389'
```

Результат:

```
-----
6
```

Но выполнение подсчета по каждому издательству оказалось более сложным. Примеры кода (неудачного) включали использование переменных, циклов WHILE, транзакций и ключевого слова DISTINCT.

Ряд доброхотов из Internet предложили воспользоваться примерно следующей стратегией:

SQL:

```
select pub_id, count(*)
from titles
group by pub_id
```

Результат:

```
pub_id
-----
0736      6
0877      6
1389      6
```

GROUP BY разделяет данные на наборы. COUNT(*) дает итоговое значение для каждого такого набора.

ПРЕДЛОЖЕНИЯ WHERE И HAVING

Взаимодействия предложений WHERE и HAVING представляют собой классические случаи недоразумений, особенно когда вы используете их вне стандартного контекста списка выбора, содержащего только агрегирующие функции и группируемые столбцы. Вот две наиболее характерные ошибки:

- WHERE, GROUP BY и HAVING без агрегирующих функций,
- WHERE, GROUP BY и HAVING со смесью агрегирующих функций и значений строк.

Почему столько строк?

Вот пример заблуждения, которое периодически появляется в Internet и обычно принимает форму двух печальных вопросов:

- Почему результаты этого запроса (если он будет выполняться на вашей системе) включают цены выше \$10,00?

SQL:

```
select price, type
from titles
where price < $10.00
group by type
```

Результат:

price	type
19.99	business
7.99	psychology
19.99	psychology
11.95	business
19.99	mod_cook
2.99	business
10.95	psychology
7.00	psychology
2.99	mod_cook
19.99	business
21.59	psychology

- Почему повторение условия WHERE в предложении HAVING “исправляет” эти результаты?

SQL:

```
select price, type
from titles
where price < $10.00
group by type
having price < $10.00
```

Результат:

price	type
2.99	business
2.99	mod_cook
7.00	psychology
7.99	psychology

С этим связана базовая проблема: что же на самом деле означает первый запрос (т.е. *bookbiz*-версия примера, который появился в Internet)? По-видимому, второй (“правильный”) ответ дает нам именно то, что и требовалось: список цен и типов для книг стоимостью меньше \$10.00.

Однако те же результаты можно было бы получить, убрав из запроса предложения GROUP BY и HAVING:

SQL:

```
select price, type
from titles
where price < $10.00
```

Результат:

price	type
2.99	business
2.99	mod_cook
7.00	psychology
7.99	psychology

В сущности, предложение GROUP BY ничего не добавляет к первому запросу. Поскольку вас интересуют только значения строк (отдельные цены и типы), нет

никакой потребности формировать группы. GROUP BY — в первом запросе — просто ошибка.

Как вы, наверное, помните из предыдущих глав, GROUP BY имеет смысл пользоваться в основном с агрегирующими функциями (в этом случае вы можете находить итоговое значение для каждой группы). Вот как, например, можно было бы найти среднюю цену книг, стоимость которых не превышает \$10.00, по типам этих книг:

SQL:

```
select avg(price), type
from titles
where price < $10.00
group by type
```

Результат:

price	type
2.99	business
2.99	mod_cook
7.50	psychology

Возможно, тот пользователь, который отправил данный вопрос в Internet, представлял себе что-то вроде этого. GROUP BY в данном случае имеет смысл, поскольку в предложении SELECT есть агрегирующая функция.

Однако вернемся к первому вопросу: почему все-таки в результатах присутствуют строки для книг, цены на которые выше \$10.00, в то время как, судя по всему, предложение WHERE должно было бы убрать их?

Если проанализировать результаты первого и второго запросов, а также результаты запроса без предложений GROUP BY и HAVING, то можно, наверное, понять, в чем тут дело. Предложение WHERE нашло строки со значениями ниже \$10.00. Они включают три типа книг: *business*, *mod_cook* и *psychology*. Предложение GROUP BY затем отобразило все книги в каждой из указанных групп. Если в этот запрос добавить ORDER BY, ситуация прояснится:

SQL:

```
select price, type
from titles
where price < $10.00
group by type
order by type
```

Результат:

price	type
2.99	business
11.95	business
19.99	business
19.99	business
2.99	mod_cook
19.99	mod_cook
7.00	psychology
7.99	psychology
10.95	psychology
19.99	psychology
21.59	psychology

Запрос с предложением GROUP BY можно перефразировать как “отобрази для меня цены и типы всех книг в группах, содержащих одну или несколько книг с ценами ниже \$10.00”. Обратите внимание, что в результатах отсутствуют строки *popular_comp* или *trad_cook*. Нет здесь книг любой из этих групп, которые стоили бы ниже \$10.00.

Поскольку в списке выбора отсутствуют агрегирующие функции, отображается каждая строка, удовлетворяющая указанным критериям (а не одна строка для итоговых значений каждой группы).

Теперь перейдем ко второму вопросу: почему повторение условия WHERE в предложении HAVING “исправляет” результаты? Во-первых, WHERE находит дешевые книги, а затем GROUP BY вставляет все книги в группы, которые содержат одну или несколько дешевых книг. Наконец, HAVING выбирает в этом списке значения ниже \$10.00. Поскольку условия WHERE и HAVING идентичны, вы получаете в конце концов те же четыре строки в результате выполнения запроса, содержащего только WHERE, и запроса, содержащего предложения WHERE, GROUP BY и HAVING. Можно было бы сказать, что HAVING в этом конкретном запросе “отменяет” действие GROUP BY.

Изменение условия в предложении HAVING делает данный процесс более понятным:

SQL:

```
select price, type
from titles
where price < $10.00
group by type
having price > $20.00
```

Результат:

price	type
21.59	psychology

После того как WHERE классифицирует строки, и GROUP BY сформирует группы, HAVING делает “завершающий аккорд”. Как вы уже видели, WHERE и GROUP BY вместе отображают одиннадцать строк. HAVING убирает все эти строки, кроме одной, цена в которой превышает \$20.00. Поскольку условия WHERE и HAVING различны, запрос с HAVING находит не те строки, которые находит запрос, содержащий только WHERE.

В сущности, если рассматривать этот конкретный запрос с его необычным использованием GROUP BY, то запрос, содержащий только HAVING, позволяет получить те же результаты, что и запрос, содержащий только WHERE, и даже запрос, содержащий и WHERE, и HAVING.

SQL:

```
select price, type
from titles
group by type
having price < $10.00
```

Результат:

price	type
2.99	business
2.99	mod_cook
7.00	psychology
7.99	psychology

На самом деле ваша реализация может не допускать использования предложения HAVING без GROUP BY. Подробнее об использовании GROUP BY и HAVING вы можете узнать из справочного руководства по вашей системе.

Наилучший способ избежать загадочных результатов, подобных указанным выше, — это тщательно проанализировать, чего именно вы хотите добиться с помощью того или иного запроса. Может оказаться (как в нашем случае), что в использовании GROUP BY нет никакой необходимости.

Сочетание значений строк и агрегирующих функций

Вот еще одна ситуация, связанная с использованием предложений GROUP BY и HAVING. Как вы, наверное, помните, есть ряд строгих ограничений на то, что разрешено помещать в списке выбора в случае использования предложения GROUP BY. Вообще говоря, вы ограничены столбцами, которые вы используете для группирования, и столбцами, для которых вы вычисляете групповые итоговые (агрегирующие) значения. В выходных результатах отображается группа и итоговое значение для группы.

В некоторых системах предусмотрены расширения, которые допускают наличие в списке выбора дополнительных столбцов и выражений — значений, которые не являются ни группами, ни агрегирующими функциями. В этом разделе рассматриваются вопросы, которые могут возникнуть, когда вы работаете с такими комбинациями значений строк и групп. Если в вашей системе эти возможности не поддерживаются, переходите к разделу “Как избежать проблем с предложением HAVING”.

Вот вопрос, который периодически возникает у пользователей: как найти итоговое значение для конкретного столбца каждой группы, а затем получить столбцы итоговых значений?

В качестве отправной точки рассмотрим *bookbiz*-версию такого запроса в его сокращенном варианте — для нахождения только минимальной цены (итогового значения) по каждому типу книг (группе):

SQL:

```
select type, min(price) as minprice
from titles
where type is not null
group by type
order by type
```

Результат:

type	minprice
popular_comp	20.00
trad_cook	11.95
business	2.99
psychology	7.00
mod_cook	2.99

Распространенным подходом для получения итоговых и строчных значений является добавление в список выбора неагрегируемого столбца, такого как *title_id* (уникального для каждой строки):

SQL:

```
select type, min(price) as minprice, title_id
from titles
where type is not null
group by type
order by type
```

Результат:

type	minprice	title_id
business	2.99	BU1032
business	2.99	BU1111
business	2.99	BU2075
business	2.99	BU7832
mod_cook	2.99	MC2222
mod_cook	2.99	MC3021
popular_comp	20.00	PC1035
popular_comp	20.00	PC8888
popular_comp	20.00	PC9999
psychology	7.00	PS1372
psychology	7.00	PS2091
psychology	7.00	PS2106
psychology	7.00	PS3333
psychology	7.00	PS7777
trad_cook	11.95	TC3218
trad_cook	11.95	TC4203
trad_cook	11.99	TC7777

Но это — не совсем то, что вам нужно. Здесь отображаются группы, минимальная цена по каждой группе и идентификационные номера книг, однако все это сделано в малопривлекательном для использования виде. В этих результатах невозможно найти идентификационные номера книг, которые имеют минимальную цену в каждой группе.

Более полный запрос, отображающий как цену, так и минимальную цену, выводит вас на правильный путь:

SQL:

```
select type, min(price) as minprice, price, title_id
from titles
where type is not null
group by type
order by type
```

Результат:

type	minprice	price	title_id
business	2.99	19.99	BU1032
business	2.99	11.95	BU1111
business	2.99	2.99	BU2075
business	2.99	19.99	BU7832
mod_cook	2.99	19.99	MC2222
mod_cook	2.99	2.99	MC3021
popular_comp	20.00	22.95	PC1035
popular_comp	20.00	20	PC8888
popular_comp	20.00		PC9999
psychology	7.00	21.59	PS1372
psychology	7.00	10.95	PS2091
psychology	7.00	7	PS2106
psychology	7.00	19.99	PS3333

psychology	7.00	7.99	PS7777
trad_cook	11.95	20.95	TC3218
trad_cook	11.95	11.95	TC4203
trad_cook	11.99	14.99	TC7777

Теперь должно быть ясно: если уж что-то вам действительно нужно, так это отобразить строку (или строки) в каждой группе, в которой значения цены и минимальной цены совпадают. В своем условии поиска вы должны использовать агрегирующую функцию. Предложение WHERE не позволяет сделать это, а HAVING — позволяет. Одним словом, похоже, что нам опять придется обратиться за помощью к предложению HAVING.

Чтобы отобразить идентификационные номера для строк, содержащих минимальную цену по каждой группе, попытайтесь сделать следующее:

```
SQL:
select type, min(price) as minprice, price, title_id
from titles
where type is not null
group by type
having price = min(price)
order by type
```

Результат:

type	minprice	title_id
business	2.99	BU2075
mod_cook	2.99	MC3021
popular_comp	20.00	PC8888
psychology	7.00	PS2106
trad_cook	11.95	TC4203

Теперь у вас есть список типов книг, минимальная цена по каждому типу и книги, имеющие такую цену.

Однако не следует забывать, что этот запрос может не подходить для некоторых систем. Многие из них не допускают каких-либо иных значений в операторе SELECT, кроме столбцов из списка GROUP BY и агрегируемых столбцов. Таким образом, может оказаться, что подобное использование столбца *title_id* является недопустимым.

Те же результаты можно получить с помощью подзапроса:

```
SQL:
select type, price, title_id
from titles t
where price =
    (select min(price)
     from titles t2
     where t.type = t2.type)
group by t2.type)
```

Результат:

type	price	title_id
business	2.99	BU2075
mod_cook	2.99	MC3021
popular_comp	20.00	PC8888
psychology	7.00	PS2106
trad_cook	11.95	TC4203

Как избежать проблем с предложением HAVING

Ниже приведено несколько соображений, которые помогут вам избежать проблем с использованием комбинаций предложений WHERE, GROUP BY и HAVING.

В запросах без агрегирующих функций предложение WHERE накладывает ограничения на строки, которые появятся в окончательном результате:

SQL:

```
select pub_id, type
from titles
group by pub_id, type
order by pub_id, type
```

Результат:

pub_id	type
0736	business
0736	psychology
0877	NULL
0877	mod_cook
0877	trad_cook
1389	business
1389	popular_comp

SQL:

```
select pub_id, type
from titles
where pub_id <> '0877'
group by pub_id, type
order by pub_id, type
```

Результат:

pub_id	type
0736	business
0736	psychology
1389	business
1389	popular_comp

В запросах с агрегирующими функциями предложение WHERE накладывает ограничения на строки, используемые при вычислении этих функций и создании групп:

SQL:

```
select pub_id, type, max(advance)
from titles
where pub_id <> '0877'
group by pub_id, type
order by pub_id, type
```

Результат:

pub_id	type	
0736	business	10125.00
0736	psychology	7000.00
1389	business	5000.00
1389	popular_comp	8000.00

Как с агрегирующими функциями, так и без них, предложение HAVING накладывает ограничения на все распечатываемые строки:

SQL:

```
select pub_id, type
from titles
where pub_id <> '0877'
group by pub_id, type
having type in ('business', 'psychology')
order by pub_id, type
```

Результат:

pub_id	type
0736	business
0736	psychology
1389	business

SQL:

```
select pub_id, type, max(advance)
from titles
where pub_id <> '0877'
group by pub_id, type
having type in ('business', 'psychology')
order by pub_id, type
```

Результат:

pub_id	type	max(advance)
0736	business	10125.00
0736	psychology	7000.00
1389	business	5000.00

Если требуется ограничить выводимые значения агрегирующих функций, воспользуйтесь предложением HAVING. Помните: агрегирующие функции нельзя включать в предложение WHERE.

SQL:

```
select pub_id, type, max(advance)
from titles
where pub_id <> '0877'
group by pub_id, type
having type in ('business', 'psychology') and
max(advance) > $5000.00
order by pub_id, type
```

Результат:

pub_id	type	max(advance)
0736	business	10125.00
0736	psychology	7000.00

Если в вашей системе допускается использование в списке выбора столбцов и выражений, которые не являются агрегирующими функциями и не находятся в предложении GROUP BY, то можно воспользоваться предложением HAVING для управления результатами:

SQL:

```
select pub_id, type, maxadv = max(advance), price
from titles
where pub_id <> '0877'
group by pub_id, type
having type in ('business', 'psychology') and
           max(advance) > $5000.00
order by pub_id, type
```

Результат:

pub_id	type	maxadv	price
1389	business	10125.00	2.99
0736	psychology	7000.00	7.00
0736	psychology	7000.00	7.99
0736	psychology	7000.00	10.95
0736	psychology	7000.00	19.99
0736	psychology	7000.00	21.59

SQL:

```
select pub_id, type, maxadv = max(advance), price
from titles
where pub_id <> '0877' group by pub_id, type
having type in ('business', 'psychology') and
           max(advance) > $5000.00 and
           advance = max(advance)
order by pub_id, type
```

Результат:

pub_id	type	maxadv	price
1389	business	10125.00	2.99
0736	psychology	7000.00	21.59

КЛЮЧЕВОЕ СЛОВО DISTINCT

Если у вас есть таблица с двумя столбцами, то как отыскать все уникальные комбинации содержащихся в них значений? Очевидным ответом в этом случае кажется использование ключевого слова **DISTINCT**, однако многие пользователи жалуются в Internet, что при использовании таких, казалось бы, простых **DISTINCT**-запросов им не удается получить ничего иного, кроме синтаксических ошибок и странных результатов.

Это, по-видимому, связано с тем, что **DISTINCT** имеет две различные формы: с именами столбцов и с агрегирующими функциями. Синтаксис этих двух форм отличается в степени вполне достаточной, чтобы окончательно сбить с толку случайного пользователя.

С именами столбцов и выражениями **DISTINCT** используется однократно, и его действие в этом случае распространяется на все, что находится после него. Оно является первым словом в списке выбора и скобки для него не требуются. Когда вы решаете использовать **DISTINCT** именно таким способом, то существенно скрываете тем самым свои действия. Вы уже не можете указать какое-либо “не-**DISTINCT**” значение для чего бы то ни было в списке выбора.

Но бывают и исключения. Когда вы используете **DISTINCT** с агрегирующими функциями (**AVG**, **SUM** и т.п.), его действие распространяется только на элементы этой конкретной агрегирующей функции. В некоторых системах в списке выбора можно использовать несколько таких “агрегирующих” **DISTINCT**.

Эти два варианта настолько похожи друг на друга, что вряд ли стоит удивляться тому страху, неопределенности и сомнениям, которые они сеют в рядах пользователей. Сколько ключевых слов DISTINCT можно использовать в списке выбора? Следует ли заключать их в скобки или нет?

Самый надежный способ избежать ошибок — это помнить, что на самом деле есть два варианта использования ключевого слова DISTINCT.

DISTINCT со столбцами и выражениями

Ниже приведен пример, который наверняка прояснит разницу между двумя формами ключевого слова DISTINCT. Допустим, вы хотите отобразить распределение цен и авансовых выплат в таблице *titles* для книг стоимостью ниже \$15.00. Получить отдельные списки уникальных цен и уникальных авансовых сумм несложно:

SQL:

```
select distinct price
from titles
where price < $15.00
```

Результат:

```
price
-----
2.99
7
7.99
10.95
11.95
14.99
```

SQL:

```
select distinct advance
from titles
where price < $15.00
```

Результат:

```
advance
-----
2275.00
4000.00
5000.00
6000.00
8000.00
10125.00
15000.00
```

Итак, имеется шесть разных цен и семь различных авансовых сумм.

Если вы хотите найти разные комбинации этих значений, не поддавайтесь искушению использовать DISTINCT для каждого столбца: DISTINCT должно быть первым словом в предложении SELECT, а его действие распространяется на все, что находится после него.

Следующий запрос отыскивает уникальные комбинации цены и аванса для книг дешевле \$15.00:

SQL:

```
select distinct price, advance
from titles
where price < $15.00
```

Результат:

price	advance
2.99	10125.00
2.99	15000.00
7	6000.00
7.99	4000.00
10.95	2275.00
11.95	4000.00
11.95	5000.00
14.99	8000.00

Полученные результаты отображают восемь разных комбинаций цены и аванса. Две цены (\$2.99 и \$11.95) появляются дважды; одна авансовая сумма (\$4000.00) также появляется дважды.

DISTINCT с агрегирующими функциями

А теперь посмотрим, как DISTINCT используется с агрегирующими функциями. Многие новички по части SQL задавались вопросом, как подсчитать различные значения в той или иной таблице, — и не находили на него ответа. Код, которым они, как правило, пытались воспользоваться, выглядит примерно так:

SQL:

```
select count(distinct *)
from titles
```

В результате на экране не появляется ничего, кроме синтаксических ошибок.

В приведенном примере DISTINCT нельзя использовать в сочетании с функцией COUNT(*). Если над этим хорошенько подумать, то станет ясно, почему. Поскольку COUNT(*) подсчитывает все строки, нет возможности определить, от каких дубликатов вы пытаетесь избавиться. От копий конкретных значений столбцов? Каких именно? От копий дубликатных строк?

По той же причине от DISTINCT вряд ли можно ожидать чего-то путного и в отношении функций MIN или MAX. Минимальное (или максимальное) значение для какого-либо столбца всегда является единственным.

Ответ можно получить, выполнив следующие два шага:

SQL:

```
select distinct pub_id
into temp_table
from titles

select count(*)
from temp_table
```

Результат:

3

Однако есть более легкий способ. DISTINCT можно использовать для подсчета уникальных значений, когда вы указываете столбец по его имени. Вот пример кода, реализующего эту задачу:

SQL:

```
select count(distinct pub_id)
from titles
```

Результат:

3

С другими агрегирующими функциями DISTINCT используется точно так же. Чтобы найти среднеарифметическое значение цен, не превышающих \$15.00, можно использовать следующий запрос:

SQL:

```
select avg(distinct price)
from titles
where price < $15
```

Результат:

9.31

Если в вашей системе допускается использование нескольких ключевых слов DISTINCT с агрегирующими функциями, можете попробовать следующий маневр:

SQL:

```
select avg(distinct price),
       avg(distinct advance)
from titles
where price < $15
```

Результат:

9.31 7200.00

Обратите внимание, что DISTINCT используется дважды — для каждой агрегирующей функции. В противном случае вы, вероятно, получите весьма сомнительные результаты.

DISTINCT и DISTINCT?

В некоторых системах баз данных допускается сочетание двух видов DISTINCT — но будьте осторожны! Комбинирование имен столбцов и агрегирующих функций может таить в себе немало подводных камней. Сочетание может запрещаться вашей системой, если по указанному столбцу не выполняется группировка. Но даже если это допускается, результаты, на первый взгляд, могут не иметь смысла. Рассмотрим следующий пример:

SQL:

```
select distinct price, avg(distinct price)
from titles
where price < $15
```

Результат:

price

NULL 9.31
2.99 9.31
7.00 9.31
7.99 9.31
10.95 9.31

11.95	9.31
14.99	9.31
19.99	9.31
20.00	9.31
20.95	9.31
21.59	9.31
22.95	9.31

То, что вы рассчитывали получить, т.е. среднеарифметическое значение разных цен (\$9.31), вы и получаете в этом запросе. Но вы получили в распечатке так много цен? Очевидно, они не были ограничены условиями предложения WHERE.

Как вы, наверное, помните, когда в списке выбора присутствуют агрегирующие функции, на них распространяется действие предложения WHERE. Для управления запросом в целом необходимо добавить предложение HAVING:

SQL:

```
select distinct price, avg(distinct price)
from titles
where price < $15
having price < $15
```

Результат:

price	
NULL	9.31
2.99	9.31
7.00	9.31
7.99	9.31
10.95	9.31
11.95	9.31
14.99	9.31

Теперь вы получили более осмысленные результаты: все уникальные цены ниже \$15.00 и среднеарифметическое значение всех этих цен. Однако совмещение этих показателей выглядит не очень впечатляюще: такое отображение больше запутывает картину, чем проясняет ее. Возможно, имеет смысл не пользоваться несколькими DISTINCT в одном запросе, а разнести их по двум отдельным запросам.

ДРУГИЕ НЕДОРАЗУМЕНИЯ

Этот раздел относится к задачам, которые SQL “должен бы” выполнять — но не выполняет. Рассмотрим следующие два примера.

- Оформление результатов в виде отчета.
- Нахождение “первого” значения.

Удаление дубликатов

Что поделаешь — SQL действительно не является инструментом форматирования! Есть кое-что, чего SQL просто не умеет делать, или умеет, но путем невероятных ухищрений. Иногда пользователю не остается ничего иного, как махнуть на SQL рукой и отправить свои результаты на какой-нибудь хороший генератор отчетов. Одним из таких случаев является удаление на выходе дублирующихся элементов.

Но надежда никогда не покидает стойких приверженцев SQL, которые очень часто спрашивают своих единомышленников по Internet, предусмотрен ли в SQL какой-нибудь метод для “подчистки”, например, таких результатов:

```
SQL:
select title_id, authors.au_id
from authors, titleauthors
where authors.au_id = titleauthors.au_id
      and titleauthors.title_id like 'P%'
order by title_id, authors.au_id
```

Результат:

title_id	au_id
PC1035	238-95-7766
PC8888	427-17-2319
PC8888	846-92-7186
PC9999	486-29-1786
PS1372	724-80-9391
PS1372	756-30-7391
PS2091	899-46-2035
PS2091	998-72-3567
PS2106	998-72-3567
PS3333	172-32-1176
PS7777	486-29-1786

Конечно, хорошо бы получить что-нибудь вроде:

title_id	au_id
PC1035	238-95-7766
PC8888	427-17-2319
	846-92-7186
PC9999	486-29-1786
PS1372	724-80-9391
	756-30-7391
PS2091	899-46-2035
	998-72-3567
PS2106	998-72-3567
PS3333	172-32-1176
PS7777	486-29-1786

Придется вас огорчить: SQL не позволяет легко сформировать отображение подобного вида. Можно лишь порекомендовать вам воспользоваться какой-нибудь программой предварительного форматирования или генератором отчетов и переформатировать эти неудобоваримые результаты.

Нахождение “первого” входа

Существует проблема, в основе которой лежит не просто синтаксическая ошибка, — это непонимание возможностей реляционной базы данных.

Этот вопрос, подобно другим вопросам, которые рассматриваются в данной главе, мы “извлекли” из Internet. Если у вас есть таблица с повторяющимся значением в столбцах, то как найти первое вхождение для такого значения.

Такая постановка вопроса вызывает определенные сомнения. Во-первых, она предполагает некий “встроенный порядок”. Во-вторых, первому вхождению для каждого значения как бы присваивается особый статус. Обе эти концепции не свойственны реляционным базам данных.

Если оказывается, что вы должны сформировать подобный запрос, то, вероятно, следует начать с более тщательного анализа структуры своей базы данных. Может быть, таблице, о которой идет речь, просто не хватает столбца, в котором можно было бы хранить информацию о том, когда или в какой последовательности вводились данные. Если добавить указанную информацию, то значения можно будет находить одно за другим. Не располагая такой информацией, вы будете вынуждены сделать ряд сомнительных предположений о том, что же это такое — “первый”.

Краткое описание синтаксиса SQL, используемого в книге

СОГЛАШЕНИЯ ПО СИНТАКСИСУ

BIG	Ключевые слова (команды) записываются прописными буквами.
MIXed	Запись из прописных и строчных букв используется для представления ключевых слов, причем их можно вводить либо целиком, либо в сокращенном виде (часть, набранную прописными буквами).
little	Строчными буквами записываются переменные.
{}	Фигурные скобки означают, что вы должны выбрать как минимум одну заключенную в них опцию.
[]	Квадратные скобки означают, что выбор одной или нескольких заключенных в них опций необязателен.
()	Круглые скобки обычно являются частью команд (в отличие от фигурных и квадратных скобок, использующихся в качестве синтаксических символов).
 	Вертикальная черта означает, что можно выбрать максимум одну опцию.
,	Запятая означает, что можно выбрать любое количество опций, разделяя их в команде запятыми.
...	Многоточие означает, что последнее действие можно повторить любое количество раз.

ФОРМАТИРОВАНИЕ

SQL не налагает никаких ограничений на длину строк и места их переноса. Однако для удобочитаемости все предложения операторов в примерах этого руководства начинаются с новой строки. Длинные предложения переносятся на новые строки и соответственно выравниваются.

Регистр

```
SELECT имя_столбца  
FROM имя_таблицы  
WHERE условия
```

В операторах этой книги ключевые слова (команды) записываются прописными буквами, а идентификаторы и пользовательские переменные — строчными. Вы можете набирать команды и идентификаторы как угодно, не обращая внимания на регистр. (Если ключевое слово записано в смешанном регистре, это означает, что его можно вводить либо целиком, либо использовать сокращенную запись, используя его часть, представленную прописными буквами.)

SELECT — это то же самое, что Select или select

Регистр, однако не безразличен для идентификаторов.

Column_name — это не то же самое, что column_name или COLUMN_NAME

СПИСОК ОПЕРАТОРОВ

В *Практическом руководстве по SQL* используются следующие операторы:

ALTER DATABASE	DROP VIEW
ALTER TABLE	DUMP DATABASE
BEGIN TRANSACTION	DUMP TRANSACTION
COMMIT TRANSACTION	GRANT
CREATE DATABASE	INSERT
CREATE INDEX	LOAD DATABASE
CREATE TABLE	LOAD TRANSACTION
CREATE VIEW	REVOKE
DELETE	ROLLBACK TRANSACTION
DROP DATABASE	SELECT
DROP INDEX	UPDATE
DROP TABLE	UPDATE STATISTICS

Аналогии между ключевыми словами разных диалектов SQL

СРАВНЕНИЕ СИНТАКСИСОВ

В этом приложении представлен синтаксис наиболее общих команд SQL (CREATE, DROP, SELECT, INSERT, UPDATE, DELETE, GRANT, REVOKE) для Sybase SQL, Sybase SQL Anywhere, Microsoft SQL Server, Informix и Oracle. Все версии операторов представлены в соответствии с используемыми в этой книге соглашениями, хотя в ряде случаев допущены некоторые упрощения, чтобы сделать сравнение более простым и удобным. Например, мы используем сокращенные имена для баз данных, таблиц, столбцов и индексов (*база данных, таблица, индекс, столбец* вместо полных имен (*база_данных.владелец.имя_таблицы* в Sybase или *db_имя@имя_сервера;владелец.имя_таблицы* в Informix)). На рис. Б.1 представлен полный синтаксис имен в различных версиях SQL.

Sybase SQL Server	[[db_имя.]владелец.]имя_таблицы
Sybase SQL Anywhere	[создатель.]имя_таблицы
Microsoft SQL Server	[[db_имя.]владелец.]имя_таблицы
Informix	[[db_имя[@имя_сервера];]владелец.]имя_таблицы
Oracle	[пользователь.]таблица

Рис.Б.1. Соглашения по именам

Если синтаксис допускал использование разных форм команд, мы выбирали форму, наиболее близкую к стандартной. Синтаксис команд в разных реализациях SQL достаточно близок, однако нужно всегда помнить и о некоторых важных отличиях.

- Не во всех реализациях поддерживается даже базисный набор команд SQL.
- Даже в сходных командах могут существовать отличия в опциях.
- В разных реализациях одинаковые команды могут иметь разный смысл.
- Разработчики постоянно совершенствуют свои версии, добавляя в них новые команды и опции. Поэтому представленный список может не отражать последних изменений.

Это приложение поможет вам сориентироваться в различных версиях SQL, но за более подробной информацией лучше обращаться к соответствующей документации.

ОПРЕДЕЛЕНИЕ ДАННЫХ

Следующие команды предназначены для создания и удаления баз данных и их объектов (индексы, таблицы и курсоры). Где это было возможно, аналогичные предложения для удобства сравнения выровнены по одной строке.

Операторы базы данных

Большинство систем поддерживают команды по созданию и удалению баз данных, хотя в ряде случаев эти функции возлагаются на операционную систему.

Sybase SQL Server	Sybase SQL Anywhere	Microsoft SQL Server	Informix	Oracle
CREATE DATABASE db [ON {DEFAULT [= size] [, dev [=size]]...} [LOG ON dev [size [, dev [size]]...] [WITH OVERRIDE] [FOR LOAD]	Утилиты инициализации	CREATE DATABASE db [ON {DEFAULT [= size] [, dev [= size]]...} [LOG ON dev [= size] [, dev [=size]]...] [FOR LOAD]	CREATE DATABASE db [IN dspace] [WITH LOG IN 'pathname' [MODE ANSI]] [WITH { (BUFFERED) LOG LOG MODE ANSI)]	CREATE DATABASE [db] [CONTROLFILE REUSE] [LOGFILE (GROUP int) file definition [, { GROUP int } file def]...] [MAXLOGFILES int] [MAXLOGMEMBERS int] [MAXLOGHISTORY integer] [DATAFILE file_def [, file_def]...] [AUTOEXTEND file_def [, file_def] [ON OFF] [NEXT int [K M]]] [MAXSIZE UNLIMITED int [K M]]] [MAXDATAFILES int] [MAXINSTANCES int] [ARCHIVELOG NOARCHIVELOG] [EXCLUSIVE] [CHARACTER SET charset]
USE db	CONNECT {TO engine} [DATABASE db] [AS connection] [USER] userid IDENTIFIED BY password	USE db	DATABASE db [EXCLUSIVE]	CONNECT user[/password] [&db]
DROP DATABASE db [,db]...	утилита erase	DROP DATABASE db [,db]...	DROP DATABASE db	

Sybase SQL Server	Sybase SQL Anywhere	Microsoft SQL Server	Informix	Oracle
<pre>[ON segment] [FOREIGN KEY (column [, column...] REFERENCES table [(column [, col- umn)...]) [CHECK (condition)]]...) [WITH MAX ROWS PER PAGE =x] [ON_segment]] ALTER TABLE table {ADD column datatype [DEFAULT default definition] [identity NULL] [column_constraint],... [, next_column_definition [column_constraint],...]}... [ADD table_constraint [DROP CONSTRAINT con- straint] REPLACE column DEFAULT new default definition PARTITION}</pre>	<pre>umn [, column...]) REFER- ENCES table [(column [, col- umn)...]) {(ON UPDATE DELETE) {CASCADE SET NULL SET DEFAULT RESTRICT} } [CHECK ON COMMIT]]]...) [IN dspace] [ON COMMIT DELETE ROWS ON COMMIT PRE- SERVE ROWS] } ALTER TABLE table {ADD column definition [column_constraint],... ADD table_constraint MODIFY column definition MODIFY column DEFAULT default value MODIFY column [NOT] NULL MODIFY column CHECK (condition) DELETE column DELETE CHECK DELETE UNIQUE (column [, column)...]) DELETE PRIMARY KEY DELETE FOREIGN KEY role RENAME table RENAME column TO column}</pre>	<pre>{FOREIGN KEY (column [, column]...)} REFERENCES table (column [, column]...)} CHECK [NOT FOR REPLICATION] (expr)}]...) [ON segment] } ALTER TABLE table [WITH NOCHECK] {ADD column definition [, column_constraint],... next_column_definition [column_constraint],...]}... ADD table_constraint DROP CONSTRAINT constraint [, constraint]...}</pre>	<pre>column)...} CHECK (condition)} [CONSTRAINT constraint]]...) [storage option] } ALTER TABLE {table syn- onym} {ADD column definition [, column_constraint],... next_column_definition [column_constraint],...]}... DROP column [, col- umn]...} MODIFY column {datatype [NOT NULL] DEFAULT de- fault definition CONSTRAINT constraint_definition} ADD CONSTRAINT constraint_definition CONSTRAINT constraint DROP CONSTRAINT constraint}</pre>	<pre>[TABLESPACE tablespace] [STORAGE storage]} FOREIGN KEY (column [, column]) REFERENCES table [(column [, column]) [ON DELETE CASCADE] [CHECK (condition)]} [EXCEPTIONS INTO table] [DISABLE]) [CLUSTER cluster (column [, column]...)] [INTRANS int] } ALTER TABLE table {ADD (column element table_constraint) [, {column_element table_constraint},...]} MODIFY (column_element [, column_element],...)} DROP drop...} PCTFREE int} PCTUSED int} [INTRANS int] MAXTRANS int} [STORAGE storage] ALLOCATE EXTENT [(size int [K M]) DATAFILE file] INSTANCE int} [ENABLE enable DISABLE disable] [NOCACHE CACHE] PARALLEL PARALLEL DEGREE n} [INSTANCES n]] } DROP TABLE table [CASCADE CONSTRAINTS]</pre>
<pre>DROP TABLE table [, ta- ble]... CREATE [UNIQUE] CLUSTERED NONCLUSTERED] INDEX index</pre>	<pre>DROP TABLE table CREATE [UNIQUE] CLUSTERED NONCLUSTERED] INDEX index</pre>	<pre>DROP TABLE table [, table]... CREATE [UNIQUE] CLUSTERED NONCLUSTERED] INDEX index</pre>	<pre>DROP TABLE {table syno- nym} CREATE [UNIQUE DIS- TINCT] [CLUSTER] INDEX index</pre>	<pre>DROP TABLE table CREATE [UNIQUE INDEX index]</pre>

Sybase SQL Server	Sybase SQL Anywhere	Microsoft SQL Server	Informix	Oracle
ON table (column [, column]...) [WITH {DEFAULT FILLFACTOR PER- CENT MAX ROWS IGNORE DUP KEY, SORTED DATA, [IGNORE_DUP_ROW ALLOW_DUP_ROW] [ON segment]]	ON table (column [ASC DESC] [, column [ASC DESC]]...) [IN dbspace]	ON table (column [, column]...) [WITH FILLFACTOR =x, IGNORE DUP KEY, {SORTED DATA SORTED DATA REORG}, {IGNORE_DUP_ROW ALLOW_DUP_ROW} [ON segment]]	ON (table synonym) (column [ASC DESC] [, column [ASC DESC]]...)	ON table (column [ASC DESC] [, column [ASC DESC]]...) [CLUSTER cluster] INITRANS int) [MAXTRANS int] [PCTFREE int) {STORAGE storage} [TABLESPACE tablespace] [NOSORT] [NOPARALLEL PARALLEL {DEGREE n INSTANCES n}] [UNRECOVERABLE RECOVERABLE]
DROP INDEX index [, in- dex]...	DROP INDEX index	DROP INDEX index [, index]...	DROP INDEX index	DROP INDEX index
CREATE VIEW view [(column [, column]...)] AS select_subset [WITH CHECK OPTION]	CREATE VIEW view [(column [, column]...)] AS select_subset [WITH CHECK OPTION]	CREATE VIEW view [(column [, column]...)] AS select_subset [WITH CHECK OPTION]	CREATE VIEW view [(column [, column]...)] AS select_subset [WITH CHECK OPTION]	CREATE [OR REPLACE] [FORCE NOFORCE] VIEW view [(column [, column]...)] AS select_subset [WITH CHECK OPTION(CON- STRAINT constraint)]
DROP VIEW view [, view]...	DROP VIEW view	DROP VIEW view [, view]...	DROP VIEW view	DROP VIEW view

МАНИПУЛЯЦИИ С ДАННЫМИ

Для манипуляций с данными используются команды SELECT, INSERT и UPDATE.

Sybase SQL Server	Sybase SQL Anywhere	Microsoft SQL Server	Informix	Oracle
SELECT [ALL DISTINCT] select list [INTO new_table] FROM (table [alias] [, table [alias] {CROSS JOIN [NATURAL KEY] JOIN [NATURAL KEY] INNER JOIN [, table [alias]	SELECT [ALL DISTINCT] select list [INTO variable_list] FROM (table [alias] [, table [alias] {CROSS JOIN [NATURAL KEY] JOIN [NATURAL KEY] INNER JOIN [, table [alias]	SELECT [ALL DISTINCT] select list [INTO new_table] FROM table [alias] [(INDEX = {name id} NOLOCK HOLDLOCK UPDLOCK TABLOCK PAGLOCK TABLOCKX FASTFIRSTROW)...]	SELECT [ALL DISTINCT] select list [INTO variable_list] FROM (table synonym) [(AS) alias] [, {OUTER} {table synonym (AS) alias}]...	SELECT [ALL DISTINCT] select list [INTO variable_list] FROM (table [alias]) [, table [alias]]...

Sybase SQL Server	Sybase SQL Anywhere	Microsoft SQL Server	Informix	Oracle
<pre> {INDEX index {PREFIX size} [LRU MRU]]} {HOLDLOCK NOHOLDLOCK} [SHARED]... } [WHERE conditions] } [GROUP BY {ALL} {column expr} [, column expr]... } [HAVING conditions] } [ORDER BY {column position expr label} {ASC DESC} [, {column position label} {ASC DESC}]... } [COMPUTE row_aggregate (col- umn) [, row_aggregate (col- umn)]...] [BY column [, column]...]] } [FOR {READ ONLY UPDATE (OF column_list)}] [AT ISOLATION (READ- UNCOMMITTED READ- COMMITTED SERIALIZ- ABLE)] } [FOR BROWSE] } INSERT INTO {table view} {(column_list)} {VALUES (values list) select_statement} } DELETE {FROM} {table view} [FROM view table (INDEX index {PREFIX size} [LRU MRU])}] [, {view table (INDEX index {PREFIX size} [LRU MRU])}]... } [WHERE conditions] } </pre>	<pre> {NATURAL KEY} LEFT OUTER JOIN {NATURAL KEY} RIGHT OUTER JOIN table {alias} [ON conditions]} } [WHERE conditions] } [GROUP BY {column label function} [, column label function]... } [HAVING conditions] } [ORDER BY {column position label} {ASC DESC} [, {column position label} {ASC DESC}]... } </pre>	<pre> [, table {alias} ((INDEX = {name id} NOLOCK HOLDLOCK UPDLOCK TABLOCK PAGLOCK TABLOCKX FASTFIRSTROW)...)]... } [WHERE conditions] } [GROUP BY {ALL} {column expr} [, column expr]... } [HAVING conditions] } [ORDER BY {column position label} {ASC DESC} [, {column position label} {ASC label} {ASC DESC}]... } [COMPUTE row_aggregate (col- umn) [, row_aggregate (col- umn)]...]] [BY column [, column]...]] } [FOR BROWSE] } INSERT INTO {table view} {(column_list)} {VALUES (values list) VALUES {DEFAULT expr} [, DEFAULT expr]... select_statement} } DELETE {FROM} {table view} [FROM table [, table]... } [WHERE conditions] } </pre>	<pre> [WHERE conditions] } [START WITH condition] } [GROUP BY {column expr} [, {column expr}]... } [HAVING conditions] } { UNION [ALL] INTERSECT MINUS } SELECT ... } [ORDER BY {expr position label} {ASC DESC} [, {expr position label } DESC]]... } [INTO TEMP temptable {WITH NOLOG}] } [FOR UPDATE (OF {table. view.} column [, {table. view.} column]...)] [NOWAIT]] } INSERT INTO table {(column_list)} {VALUES (values list) select_statement} } DELETE {FROM} table {alias} [WHERE conditions] } </pre>	

Sybase SQL Server	Sybase SQL Anywhere	Microsoft SQL Server	Informix	Oracle
UPDATE {table view} SET column = {expr NULL select_statement} [, column = {expr NULL select_statement}]... [FROM {view table view} [, {table view}]]... [(INDEX index {PREFIX size} [LRU MRU])] [, {view table} {(INDEX index {PREFIX size} [LRU MRU])}]...] [WHERE conditions]	UPDATE {table view} [, {table view}]... SET column = expr [, column = expr]... [FROM {table view} [, {table view}]]... [WHERE conditions] [ORDER BY expr {ASC DESC} [, expr {ASC DESC}]]...	UPDATE {table view} SET column = {expr NULL DEFAULT} [, column = {expr NULL DEFAULT}]... [FROM {view table} [,view table]]... [WHERE conditions]	UPDATE {table view synonym} SET column = {expr select_statement} [, column = {expr select_statement}]... [WHERE conditions]	UPDATE table [alias] SET {column = expr [, column = expr]}... [(column [, column]...)] = {sub-query} [WHERE conditions]

АДМИНИСТРИРОВАНИЕ ДАННЫХ

Для администрирования данных используются команды GRANT и REVOKE.

Sybase SQL Server	Sybase SQL Anywhere	Microsoft SQL Server	Informix	Oracle
Полномочия на объекты GRANT {ALL PRIVILEGES} {SELECT INSERT DELETE UPDATE REFER-ENCES EXECUTE} {SELECT INSERT DELETE UPDATE REFERENCES} [, column]...] REFERENCES ALTER	Полномочия на объекты GRANT {ALL PRIVILEGES} {SELECT INSERT DELETE UPDATE {column [, column]...}} REFERENCES ALTER [, SELECT INSERT DELETE UPDATE {column [, column]...}} REFERENCES ALTER [, ...]	Полномочия на объекты GRANT {ALL PRIVILEGES} {SELECT INSERT DELETE UPDATE REFERENCES EXECUTE} {SELECT INSERT DELETE UPDATE REFERENCES EXECUTE}...]	Полномочия на объекты GRANT {ALL PRIVILEGES} {SELECT {column [, column]...}} INSERT DELETE UPDATE {column [, column]...}} REFERENCES {column [, column]...} ALTER INDEX]...]	Полномочия на объекты GRANT {SELECT {column [, column]...}} INSERT DELETE UPDATE {column [, column]...}} REFERENCES {column [, column]...}} ALTER INDEX

Sybase SQL Server	Sybase SQL Anywhere	Microsoft SQL Server	Informix	Oracle
ON {table [, column]...} [, column]...} view {(column [, column]...)} procedure} TO { PUBLIC user [, user]...} role} WITH GRANT OPTION}	ON table TO userid [, userid]... WITH GRANT OPTION}	ON {table [, column [, col- umn]...]} view {(column [, column]...)} procedure extended_procedure} TO { PUBLIC user [, user]...}	ON {table view synonym } TO {PUBLIC user [, user]...} WITH GRANT OPTION} [AS GRANTOR]	ON {table view procedure} TO {PUBLIC user [, user]...} role [, role]...} WITH GRANT OPTION}
Полномочия на базу данных GRANT {ALL [PRIVILEGES] CREATE_commands} TO {PUBLIC user [,user]... role}	Полномочия на базу данных GRANT {DBA RESOURCE GROUP MEMBERSHIP IN GROUP userid } [, DBA RESOURCE GROUP MEMBERSHIP IN GROUP userid]... TO userid	Полномочия на базу данных GRANT{ALL CREATE/DUMP_commands} TO {PUBLIC user [,user]...}	Полномочия на базу данных GRANT {CONNECT RESOURCE DBA} [, CON- NECT RESOURCE DBA]... TO {PUBLIC user [, user]...}	Полномочия на базу данных GRANT {system_privilege [, system_privilege]... role [, role]} ...} TO {PUBLIC user [, user]... role [, role]...} [, PUBLIC user [, user]... role {, role}...] [WITH ADMIN OPTION]
Полномочия на объекты REVOKE {GRANT_OPTION_FOR {ALL [PRIVILEGES] SELECT INSERT DELETE UPDATE REFER- ENCES EXECUTE} [, SELECT INSERT REFER- ENCES EXECUTE] [, SELECT INSERT DELETE UPDATE {(column [, col- umn]...)} REFERENCES ALTER} ON {table {(column [, col- umn]...)} view {(column [, column]]...)} procedure} FROM {PUBLIC user [, user] [, role]...} [CASCADE]}	Полномочия на объектах REVOKE {SELECT INSERT DELETE UPDATE {(column [, column]...)} REFERENCES ALTER} [, SELECT INSERT DELETE UPDATE {(column [, col- umn]...)} REFERENCES ALTER} ON table FROM userid [, userid]...	Полномочия на объектах REVOKE {SELECT INSERT {ALL SELECT INSERT DELETE UPDATE REFER- ENCES EXECUTE} [, SELECT INSERT DELETE UPDATE REFERENCES ALTER INDEX] ON {table view synonym } FROM {PUBLIC user [, user]...}	Полномочия на объектах REVOKE {SELECT {(column [, column]...)} INSERT DELETE UPDATE {(column [, column]...)} REFERENCES [(column [, column]...)] ALTER INDEX} , SELECT {(column [, column]...)} INSERT DELETE UPDATE {(column [, column]...)} REFERENCES [(column [, column]...)] ALTER INDEX]...} ON {table view procedure} FROM {PUBLIC user [, user]... role [, role]...} [CASCADE CONSTRAINTS]}	

Sybase SQL Server	Sybase SQL Anywhere	Microsoft SQL Server	Informix	Oracle
<p>Полномочия на базу данных REVOKE (ALL [PRIVILEGES] CREATE commands) FROM (PUBLIC user [, user]... role [, role]...)</p>	<p>Полномочия на базу данных REVOKE (DBA RESOURCE GROUP MEMBERSHIP IN GROUP us- erid) [, DBA RESOURCE GROUP MEMBERSHIP IN GROUP userid]... FROM userid</p> <p>Полномочия на процедуру REVOKE EXECUTE ON procedure TO userid [, userid]...</p>	<p>Полномочия на базу данных REVOKE (ALL CREATE/DUMP commands) FROM (PUBLIC user [, user]...)</p>	<p>Полномочия на базу данных REVOKE (CONNECT RESOURCE DBA) [, CON- NECT RESOURCE DBA]... FROM (PUBLIC user [, user]...)</p>	<p>Полномочия на базу данных REVOKE (system_privilege [, system_privilege]... role [, role]...) FROM (PUBLIC user [, user]... role [, role]...) {, PUBLIC user [, user]... role [, role]...}...</p>

Словарь терминов

access strategy (стратегия доступа)

Метод, с помощью которого система управления базами данных находит физические данные.

aggregate functions (агрегирующие функции)

Чаще всего используемые в предложениях GROUP BY и HAVING, агрегирующие функции генерируют одно суммарное значение для группы значений в указанном столбце. В их число входят функции AVG, COUNT, COUNT(*), MIN и MAX.

alias (псевдоним)

Временное имя, присвоенное таблице (в предложении FROM). Ниже приведено два соответствующих примера:

```
Select au_id, a.city, p.city, pub_id
from authors a, publishers p
where au_lname like 'P%'
```

```
select a.au_id, b.au_id
from authors a, authors b
where a.zip = b.zip
```

В первом примере благодаря псевдонимам не нужно набирать полные имена таблиц в предложении FROM. Во втором примере псевдонимы применяются для самообъединения — таблица *authors* получает два псевдонима — *a* и *b*.

argument (аргумент)

Значение (также называемое параметром), передаваемое в функцию.

arithmetic operators (арифметические операторы)

Арифметическими операторами являются сложение (+), вычитание (-), умножение (*) и деление (/). Эти операторы могут использоваться с любыми числовыми столбцами. В некоторых системах также поддерживается оператор деления по модулю (%), возвращающий целую часть от деления двух целых чисел.

association (связь)

Отношение многих-ко-многим между объектами.

attribute (атрибут)

Значение, описывающее одну характеристику объекта. Также часто называется полем или столбцом.

base table (базовая таблица)

Постоянная физическая таблица базы данных, на основе которой строятся курсы (также называемые виртуальными таблицами).

benchmarking (сравнение с эталоном)

Процесс тестирования аппаратного и программного обеспечения, в ходе которого определяются его сравнительные характеристики.

binary datatype (двоичный тип данных)

Тип данных для хранения двоичной информации.

bit datatype (битовый тип)

Тип данных для хранения данных вида истина/ложь.

boolean expressions (булево выражение)

Выражение, возвращающее значение “истина” или “ложь”.

boolean operators (булевы операторы)

Логические операторы AND, OR и NOT.

buffer cache (кэш буфера)

Память, выделяемая для хранения данных.

cartesian product (декартово произведение)

Все возможные комбинации строк таблицы.

cascade (каскадирование)

Распространение операций обновления или удаления на связанные таблицы в базе данных.

character datatype (символьный тип данных)

Тип данных для хранения символьной информации, такой как буквы, числа и специальные символы.

character set (набор символов)

Список букв и специальных символов и соответствующих им компьютерных кодов.

clustered index (кластерный индекс)

Индекс, на нижнем уровне которого находятся сами данные. Таблица может иметь только единственный такой индекс.

collating sequence (упорядоченная последовательность)

См. **sort order (порядок сортировки)**

collation

См. **sort order (порядок сортировки)**

column (столбец)

Атрибут или характеристика объекта в таблице. Также называется полем.

command (команда)

Любой оператор SQL, например INSERT или CREATE DATABASE.

comparison operators (операторы сравнения)

Операторы, используемые для сравнения значений выражений в предложениях WHERE или HAVING. В их число входят оператор равенства (=), оператор больше чем (>), оператор меньше чем (<), оператор больше или равно (>=), оператор меньше или равно (<=) и оператор не равно (!= или <>).

composite indexes (составные индексы)

Индексы, основанные больше чем на одном столбце таблицы.

data sublanguage (подязык данных)

Язык для общения с базой данных.

concurrency control (контроль совпадений)

Блокировка, которая предотвращает одновременное изменение несколькими пользователями одной и той же информации.

connecting column (столбцы соединения)

Столбцы, по которым выполняется объединение таблиц. Столбцы соединения, принадлежащие одной или нескольким таблицам, должны содержать совместимые значения.

constraints (ограничения)

Предложения в операторе CREATE TABLE (CHECK, PRIMARY KEY, UNIQUE, REFERENCES, FOREIGN KEY), которые обеспечивают ссылочную целостность и поддерживают бизнес-правила.

correlated subquery (коррелированный подзапрос)

Подзапрос, который не может выполняться независимо от внешнего запроса. Также называется "повторяющимся подзапросом", так как выполняется для каждой строки, выбираемой во внешнем запросе.

data administration (администрирование данных)

Одно из трех основных понятий SQL. Два другие — определение данных и манипуляция с данными. На этапе администрирования данных определяются полномочия пользователей на выполнение тех или иных действий.

data cache (кэш данных)

Область памяти, выделенная под хранение данных.

data control (управление данными)

Другой термин для обозначения процесса администрирования данных.

data definition (определение данных)

Процесс создания (или удаления) базы данных и ее объектов.

data dictionary (словарь данных)

Системная таблица, которая содержит описания объектов базы данных и их структур.

data manipulation (манипуляция данными)

Выборка и изменение данных с помощью операторов SELECT, INSERT, DELETE и UPDATE.

data modification (модификация данных)

Изменение данных с помощью операторов SQL INSERT, DELETE и UPDATE.

data retrieval (выборка данных)

Поиск и отображение данных из базы данных посредством запросов (операторы SELECT).

data structure diagram (структурная диаграмма данных)

Диаграмма, отображающая зависимости между объектами базы данных.

database (база данных)

Набор связанных таблиц, содержащих данные и определения объектов базы данных.

database administrator (администратор базы данных)

См. system administrator (системный администратор)

database design (проектирование базы данных)

Процесс определения объектов базы данных (чаще всего таблиц и их столбцов).

database device (устройство базы данных)

Физическое или логическое устройство, на котором хранится база данных.

database owner (владелец базы данных)

Создатель или владелец базы данных. Это понятие наиболее характерно для многопользовательских систем.

decimal datatype (десятичный тип данных)

Тип данных, используемый для записи десятичных данных.

default (значение по умолчанию)

Значение, которое автоматически вводится системой, если пользователь не указывает конкретное значение столбца.

derived tables (производные таблицы)

Так иногда называют курсоры, которые также известны как “виртуальные таблицы”.

difference (разность)

Операция из теории множеств, результатом которой являются строки первой таблицы, отсутствующие во второй таблице.

distinguished nulls (различаемые нули)

Значения, в точности не известные, но о которых имеется некоторая информация.

domain (домен)

Набор всех допустимых значений столбца.

entity integrity (целостность объекта)

Правило, требующее, чтобы каждая строка имела первичный ключ и чтобы в нем отсутствовали нулевые значения.

entity-relationship diagram (диаграмма взаимосвязей между объектами)

См. data structure diagram (структурная диаграмма данных)

escape character (ключевой символ)

Символ, используемый в предложении LIKE с ключевым словом ESCAPE и представляющий SQL трактовать символ шаблона как обычный литерал.

exclusive lock (исключительная блокировка)

Иногда называется блокировкой на запись. Позволяет пользователю единолично распоряжаться строками, страницами или таблицами в процессе их изменения.

expression (выражение)

Константа, имя столбца, функция или любая их комбинация с арифметическими операторами.

field (поле)

Атрибут объекта, столбец таблицы.

file (файл)

Часто используется как эквивалент таблицы.

fillfactor

Параметр индексации, поддерживаемый в Transact-SQL и позволяющий управлять плотностью заполнения страниц с новыми индексами. Эта величина оказыва-

ет влияние на производительность, так как для разбиения заполненной на 100% страницы индексов системе потребуется дополнительное время.

first normal form (первая нормальная форма)

Первая из пяти нормальных форм. Она требует, чтобы в таблице содержалось конечное число столбцов при отсутствии повторяющихся групп.

fixed length (фиксированная длина)

Некоторые типы данных могут иметь или фиксированную или переменную длину. Правильный выбор может влиять на размеры расходуемой памяти и на производительность.

foreign key (внешний ключ)

Столбец в таблице, соответствующий первичному столбцу в другой таблице.

form-of-use (используемая форма)

См. character set (набор символов)

form system (система с формами)

Тип интерфейса пользователя, определяющий, в каком месте вы можете ввести данные.

free-form language (необусловленный язык)

Компьютерный язык, не накладывающий ограничений на длину строки и места их разрывов.

grouped view (сгруппированный курсор)

Курсор, содержащий в своем определении предложение GROUP BY.

identifier (идентификатор)

Имя базы данных или ее объекта.

inclusive range (включающий диапазон)

Диапазон, определенный с ключевым словом BETWEEN, в котором при поиске учитываются как промежуточные, так и граничные значения.

index (индекс)

Механизм для поиска данных.

instance (экземпляр)

Каждая строка таблицы, представляющая конкретный объект.

integrity (целостность)

Непротиворечивость и правильность данных.

intersection (пересечение)

Операция из теории множеств, с помощью которой находят общие строки двух и более таблиц.

join (объединение)

Выборка из нескольких таблиц на основе сравнения значений в определенных столбцах.

join column (столбец объединения)

Столбец, используемый для объединения.

join-compatible columns (совместимые для объединения столбцы)

Столбцы, содержащие данные аналогичных типов.

key values (ключевые значения)

Первичные ключи однозначно идентифицируют строки, тогда как внешние ключи обеспечивают доступ к ним из других таблиц.

keyword (ключевое слово)

Слово, используемое в синтаксисе SQL. Также называется "зарезервированным словом".

locking (блокировка)

Механизм, предотвращающий одновременное изменение одних и тех же данных несколькими пользователями и просмотр данных в момент выполнения транзакции.

logical independence (логическая независимость)

Понятие, отражающее тот факт, что взаимосвязи между таблицами, столбцами и строками могут изменяться, не влияя при этом на работу приложений и выполнение типичных запросов.

logical operators (логические операторы)

AND (объединяет два и более условий и возвращает истинный результат при выполнении всех условий), OR (соединяет два и более условий и возвра-

щает истинный результат при выполнении одного из условий) и NOT (отрицание условия).

lookup table (поисковая таблица)

Таблица, используемая в основном для ссылочных целей, а не для хранения или модификации данных.

many-to-many (многий-ко-многим)

Зависимость между таблицами (например, между таблицами *authors* и *titles*), при которой автор может иметь несколько книг, а книга может иметь нескольких авторов.

master table (основная таблица)

Таблица в базе данных, содержащая наиболее значимую информацию по определенному предмету (например, таблица *sales* в базе данных *bookbiz*), связанная с одной или несколькими вспомогательными таблицами.

modulo (деление по модулю)

Арифметическая операция, возвращающая целый остаток от деления двух целых чисел. Например, 21 по модулю 9 равно 3, так как при делении 21 на 9 получается 2, а в остатке остается 3.

money datatype (денежный тип)

Тип данных, используемый для записи денежных значений.

nested query (вложенный запрос)

См. *subquery* (подзапрос)

nested sort (вложенная сортировка)

Сортировка “внутри” другой сортировки.

nonrelated subquery (несвязанный подзапрос)

Запрос, который может выполняться независимо от внешнего оператора SQL.

non-loss decomposition (декомпозиция без потери информации)

Процесс разбиения таблицы на несколько меньших таблиц без потери информации.

nonprocedural language (непроцедурный язык)

Язык, позволяющий описать желаемые результаты, без указания способа их получения.

normal forms (нормальные формы)

См. *normalization* (нормализация)

normalization (нормализация)

Набор стандартов (нормальных форм) проектирования структур данных. В настоящее время широко используются пять нормальных форм.

null (нуль)

Нули представляют пропущенные или неприменимые значения в столбцах таблиц базы данных.

one-to-many (один-ко-многим)

Отношение, в котором строка первой таблицы может быть связана с несколькими строками второй таблицы, но любая строка второй таблицы может быть связана только с единственной строкой первой таблицы.

owner (владелец)

Создатель объекта базы данных является его владельцем и обычно имеет на него полный набор полномочий.

permissions (разрешения)

Допуск на выполнение определенных действий на определенных объектах базы данных или на выполнение определенных команд.

physical data independence (физическая независимость данных)

Независимость способа физического хранения данных от логической структуры базы данных.

primary key (первичный ключ)

Столбец или столбцы, значения которых однозначно идентифицируют строки таблицы.

projection (проекция)

Выбор столбцов, значения которых должны быть включены в результаты выполнения запроса.

qualifications (уточнение)

Условия на выбираемые строки, содержащиеся в предложениях WHERE или HAVING.

queries (запросы)

Требования на выборку информации из базы данных.

query optimizer (оптимизатор запроса)

Специальное программное обеспечение СУБД, определяющее наиболее эффективный способ выполнения запроса.

record (запись)

Набор связанных полей, описывающий определенный объект. Также называется кортежем или строкой.

recovery (восстановление)

Приведение базы данных в рабочее состояние после аппаратного или программного сбоя.

referential integrity (ссылочная целостность)

Правила, управляющие целостностью данных. В соответствии с ними внешний ключ должен либо полностью совпадать с соответствующим первичным ключом, либо быть полностью нулевым.

relation (отношение)

Синоним таблицы.

repeating subquery (повторяющийся подзапрос)

См. *correlated subquery* (коррелированный подзапрос)

restrict (ограничивать)

См. *restriction* (ограничение)

restriction (ограничение)

Одна из основных операций в реляционных системах, также называемая выбором. Ограничение определяет, какие строки должны выбираться из таблицы.

rows (строки)

Набор значений данных, описывающий экземпляр объекта.

rule

Спецификация, управляющая тем, какие данные могут вводиться в определенном столбце.

scalar aggregate (скалярная агрегирующая функция)

Агрегирующая функция, возвращающая из оператора SELECT единственное значение.

schema (схема)

В SQL-92 — набор объектов базы данных, принадлежащих одному пользователю. Также используется для описания общей структуры базы данных.

second normal form (вторая нормальная форма)

Требует, чтобы все неключевые столбцы были связаны со всем первичным ключом, а не только с одним из его компонентов.

selection (выбор)

Определение условий на выборку строк из таблицы.

select list (список выбора)

Звездочка (для задания всех столбцов) или список столбцов и выражений, которые будут включены в результаты запроса.

self-join (самообъединение)

Выборка из таблицы на основе сравнения значений из одного или нескольких столбцов этой же таблицы.

serial datatype (последовательный тип)

Набор последовательно возрастающих значений.

sets (наборы)

Группы строк, к которым применяются агрегирующие функции.

shared lock (совместная блокировка)

Блокировка, создаваемая операциями чтения. При этом пользователи могут одновременно просматривать данные, и пока все подобные блокировки не будут сняты, ни одна транзакция не сможет выполнить исключительную блокировку.

sort order (порядок сортировки)

Порядок, определяющий последовательность символов.

SQL

Язык для определения, изменения и управления данными в реляционных базах данных. Сокращение от Structured Query Language — язык структурированных запросов.

statement (оператор)

Команда определения, манипуляции или администрирования данных.

strings (строки)

Набор из одной или более букв, чисел или специальных символов (таких как знаки вопроса и звездочки).

subquery (подзапрос)

Оператор SELECT, вложенный в предложение WHERE или в другой оператор SELECT.

system administrator (системный администратор)

Лицо, ответственное за поддержку базы данных, непротиворечивость и целостность данных.

system catalog (системный каталог)

Системные таблицы, содержащие описания объектов базы данных и их структур.

system tables (системные таблицы)

См. **system catalog (системный каталог)**

table (таблица)

Представление данных в виде строк и столбцов.

table list (список таблиц)

Список таблиц, курсоров или того и другого после ключевого слова FROM в операторе SELECT.

table scan (сканирование таблицы)

Просмотр каждой строки таблицы вместо использования для поиска информации индексов. (Для небольших таблиц сканирование может быть наиболее эффективным методом доступа.)

terminator (терминатор)

Символ, слово или команда меню, используемые для отметки конца оператора SQL.

time datatype (временной тип данных)

Тип данных для записи информации о времени.

transaction (транзакция)

Механизм, обеспечивающий выполнение целого ряда действий в качестве одного неделимого задания.

transaction log (журнал транзакций)

Файл, в который записываются изменения, вносимые в базу данных (обычно используется для восстановления базы данных после сбоя).

transaction management (управление транзакциями)

Управление транзакциями обеспечивает либо успешное завершение транзакции, либо ее отмену.

trigger (триггер)

Значение этого термина может изменяться от системы к системе. В Transact-SQL — это специальная хранимая процедура, которая активизируется при выполнении команды модификации данных в заданной таблице или столбце.

trigger conditions (триггерные условия)

В Transact-SQL — условия выполнения триггера.

tuple (кортеж)

Набор связанных атрибутов, описывающих определенный объект. Также называется строкой или записью.

unique indexes (уникальные индексы)

Индекс с неповторяющимися первичными ключами.

unmodified comparison operator (немодифицированный оператор сравнения)

Оператор сравнения, за которым не следуют ключевые слова ANY или ALL.

user-defined datatypes (типы данных, определяемые пользователем)

В Transact-SQL — типы данных, определенные пользователем на основе системных типов данных с такими характеристиками как нулевой статус и длина.

user-defined transaction (транзакция, определенная пользователем)

Транзакция, определенная с помощью команд BEGIN TRANSACTION и COMMIT TRANSACTION.

user tables (пользовательские таблицы)

Таблицы с информацией, составляющие основу любой базы данных.

validation rules (правила проверки)

Правила, определяющие, какие данные могут быть введены в определенный столбец.

value (значение)

Элемент данных, находящийся, например, на пересечении строки и столбца.

vector aggregate (векторная агрегирующая функция)

Агрегирующая функция, возвращающая массив значений (по одному на каждый набор).

view (курсор)

Альтернативный способ просмотра данных из одной или нескольких таблиц.

virtual table (виртуальная таблица)

См. view (курсор)

wildcards (шаблоны)

Символы, используемые в ключевом слове LIKE, представляющие один символ (подчеркивание, _) или любое количество символов (знак процента, %).

Описание базы данных bookbiz

Ниже описывается база данных *bookbiz*, которая содержит таблицы *authors*, *publishers*, *roysched*, *titleauthors*, *titles*, *editors*, *titleditors*, *sales* и *salesdetails*. Содержащаяся в базе данных информация представлена в форме таблиц, кроме того, приводится соответствующий ей код SQL с операторами CREATE и INSERT.

ТАБЛИЦЫ

В заголовках таблиц указывается тип данных столбцов и их нулевой статус. Также отмечаются индексированные столбцы.

<i>Publishers</i>					
<i>pub_id</i> char(4) not null		<i>pub_name</i> varchar(40) null	<i>address</i> varchar(40) null	<i>city</i> varchar(20) null	<i>state</i> char(2) null
<i>unique index</i>					
<i>pubind</i>					
0736		New Age Books	1 1st St	Boston	MA
0877		Binnet & Hardley	2 2nd Ave.	Washington	DC
1389		Algodata Infosystems	3 3rd Dr.	Berkeley	CA

<i>authors</i>							
<i>au_id</i> char(11) not null	<i>au_lname</i> varchar(40) not null	<i>au_fname</i> varchar(20) not null	<i>phone</i> char(12) null	<i>address</i> varchar(40) null	<i>city</i> varchar(20) null	<i>state</i> char(2) null	<i>zip</i> char(5) null
<i>unique index</i>		<i>composite index</i>					
<i>auaidind</i>		<i>auainmind</i>					
409-56-7008	Bennet	Abraham	415 658-9932	6223 Bateman St.	Berkeley	CA	94705
213-46-8915	Green	Marjorie	415 986-7020	309 63rd St. #411	Oakland	CA	94618
238-95-7766	Carson	Cheryl	415 548-7723	589 Darwin Ln.	Berkeley	CA	94705
998-72-3567	Ringer	Albert	801 826-0752	67 Seventh Av.	Salt Lake City	UT	84152
899-46-2035	Ringer	Anne	801 826-0752	67 Seventh Av.	Salt Lake City	UT	84152
722-51-5454	DeFrance	Michel	219 547-9982	3 Balding Pl.	Gary	IN	46403
807-91-6654	Panteley	Sylvia	301 946-8853	1956 Arlington Pl.	Rockville	MD	20853
893-72-1158	McBadden	Heather	707 448-4982	301 Putnam	Vacaville	CA	95688
724-08-9931	Stringer	Dirk	415 843-2991	5420 Telegraph Av.	Oakland	CA	94609
274-80-9391	Straight	Dick	415 834-2919	5420 College Av.	Oakland	CA	94609
756-30-7391	Karsen	Livia	415 534-9219	5720 McAuley St.	Oakland	CA	94609
724-80-9391	MacFeather	Stearns	415 354-7128	44 Upland Hts.	Oakland	CA	94612
427-17-2319	Dull	Ann	415 836-7128	3410 Blonde St.	Palo Alto	CA	94301
672-71-3249	Yokomoto	Akiko	415 935-4228	3 Silver Ct.	Walnut Creek	CA	94595
267-41-2394	O'Leary	Michael	408 286-2428	22 Cleveland Av. #14	San Jose	CA	95128
472-27-2349	Gringlesby	Burt	707 938-6445	PO Box 792	Covelo	CA	95428
527-72-3246	Greene	Morningstar	615 297-2723	22 Graybar House Rd.	Nashville	TN	37215
172-32-1176	White	Johnson	408 496-7223	10932 Bigge Rd.	Menlo Park	CA	94025
712-45-1867	del Castillo	Innes	615 996-8275	2286 Cram Pl. #86	Ann Arbor	MI	48105
846-92-7186	Hunter	Sheryl	415 836-7128	3410 Blonde St.	Palo Alto	CA	94301
486-29-1786	Locksley	Chastity	415 585-4620	18 Broadway Av.	San Francisco	CA	94130
648-92-1872	Blotchett-Halls	Reginald	503 745-6402	55 Hillsdale Bl.	Corvallis	OR	97330
341-22-1782	Smith	Meander	913 843-0462	10 Misisipi Dr.	Lawrence	KS	66044

titles									
title_id char(6) not null	title varchar(80) not null	type char(12) null	pub_id char(4) null	price money null	advance money null	ytd_sales int null	contract bit not null	notes varchar(200) null	pubdate date null
unique index titleidind	index titleind								
PC8888	Secrets of Silicon Valley	popular_co mp	1389	20	8000	4095	1	Muckraking reporting on the world's largest computer hardware and software manufacturers.	Jun 12 1985 12:00AM
BU1032	The Busy Executive's Database Guide	business	1389	19,99	5000	4095	1	An overview of available database systems with emphasis on common business applications. Illustrated.	Jun 12 1985 12:00AM
PS7777	Emotional Security: A New Algorithm	psychology	0736	7,99	4000	3336	1	Protecting yourself and your loved ones from undue emotional stress in the modern world. Use of computer and nutritional aids emphasized.	Jun 12 1985 12:00AM
PS3333	Prolonged Data Deprivation: Four Case Studies	psychology	0736	19,99	2000	4072	1	What happens when the data runs dry? Searching evaluations of information-shortage effects.	Jun 12 1985 12:00AM
BU1111	Cooking with Computers: Surreptitious Balance Sheets	business	1389	11,95	5000	3876	1	Helpful hints on how to use your electronic resources to the best advantage.	Jun 9 1985 12:00AM
MC2222	Silicon Valley Gastronomic Treats	mod_cook	0877	19,99	0	2032	1	Favorite recipes for quick, easy, and elegant meals tried and tested by people who never have time to eat, let alone cook.	Jun 9 1985 12:00AM
TC7777	Sushi, Anyone?	trad_cook	0877	14,99	8000	4095	1	Detailed instructions on improving your position in life by learning how to make authentic Japanese sushi in your spare time. 5-10% increase in number of friends per recipe reported from beta test.	Jun 12 1985 12:00AM
TC4203	Fifty Years in Buckingham Palace Kitchens	trad_cook	0877	11,95	4000	15096	1	More anecdotes from the Queen's favorite cook describing life among English royalty. Recipes, techniques, tender vignettes.	Jun 12 1985 12:00AM

titles									
title_id char(6) not null	title varchar(80) not null	type char(12) null	pub_id char(4) null	price money null	advance money null	ytd_sales int null	contract bit not null	notes varchar(200) null	pubdate date null
unique index titleidind	index titleind								
PC1035	But Is It User Friendly?	popular_c mp	1389	22,95	7000	8780	1	A survey of software for the naive user, focusing on the 'friendliness' of each.	Jun 30 1985 12:00AM
MC3026	The Psychology of Computer Cooking	NULL	0877	NULL	NULL	NULL	0	NULL	NULL
BU2075	You Can Combat Computer Stress!	business	0736	2,99	10125	18722	1	The latest medical and psychological techniques for living with the electronic office. Easy-to-understand explanations.	Jun 30 1985 12:00AM
PS2091	Is Anger the Enemy?	psychology	0736	10,95	2275	2045	1	Carefully researched study of the effects of strong emotions on the body. Metabolic charts included.	Jun 15 1985 12:00AM
PS2106	Life Without Fear	psychology	0736	7	6000	111	1	New exercise, meditation, and nutritional techniques that can reduce the shock of daily interactions. Popular audience. Sample menus included, exercise video available separately.	Oct 5 1985 12:00AM
MC3021	The Gourmet Microwave	mod_cook	0877	2,99	15000	22246	1	Traditional French gourmet recipes adapted for modern microwave cooking.	Jun 18 1985 12:00AM
TC3218	Onions, Leeks, and Garlic: Cooking Secrets of the Mediterranean	trad_cook	0877	20,95	7000	375	1	Profusely illustrated in color, this makes a wonderful gift book for a cuisine-oriented friend.	Oct 21 1985 12:00AM
BU7832	Straight Talk About Computers	business	1389	19,99	5000	4095	1	Annotated analysis of what computers can do for you: a no-hype guide for the critical user.	Jun 22 1985 12:00AM
PS1372	Computer Phobic and Non-Phobic Individuals: Behavior Variations	psychology	0736	21,59	7000	375	1	A must for the specialist, this book examines the difference between those who hate and fear computers and those who think they are swell.	Oct 21 1985 12:00AM
PC9999	Net Etiquette	popular_c mp	1389	NULL	NULL	NULL	0	A must-read for computer conferencing debutantes!.	NULL

titleauthors			
au_id char(11) not null	title_id char(6) not null	au_ord tinyint null	royaltyshare float null
unique composite index taind			
409-56-7008	BU1032	1	0.600000000
486-29-1786	PS7777	1	1.000000000
486-29-1786	PC9999	1	1.000000000
712-45-1867	MC2222	1	1.000000000
172-32-1176	PS3333	1	1.000000000
213-46-8915	BU1032	2	0.400000000
238-95-7766	PC1035	1	1.000000000
213-46-8915	BU2075	1	1.000000000
998-72-3567	PS2091	1	0.500000000
899-46-2035	PS2091	2	0.500000000
998-72-3567	PS2106	1	1.000000000
722-51-5454	MC3021	1	0.750000000
899-46-2035	MC3021	2	0.250000000
807-91-6654	TC3218	1	1.000000000
274-80-9391	BU7832	1	1.000000000
427-17-2319	PC8888	1	0.500000000
846-92-7186	PC8888	2	0.500000000
756-30-7391	PS1372	1	0.750000000
724-80-9391	PS1372	2	0.250000000
724-80-9391	BU1111	1	0.600000000
267-41-2394	BU1111	2	0.400000000
672-71-3249	TC7777	1	0.400000000
267-41-2394	TC7777	2	0.300000000
472-27-2349	TC7777	3	0.300000000
648-92-1872	TC4203	1	1.000000000

sales			
sonum int not null	stor_id char(4) not null	ponum varchar(20) not null	sdate date null
1	7066	QA7442.3	Sep 13 1985 12:00AM
2	7067	D4482	Sep 14 1985 12:00AM
3	7131	N914008	Sep 14 1985 12:00AM
4	7131	N914014	Sep 14 1985 12:00AM
5	8042	423LL922	Sep 14 1985 12:00AM
6	8042	423LL930	Sep 14 1985 12:00AM
7	6380	722a	Sep 13 1985 12:00AM
8	6380	6871	Sep 14 1985 12:00AM
9	8042	P723	Mar 11 1988 12:00AM
19	7896	X999	Feb 21 1988 12:00AM
10	7896	QQ2299	Oct 28 1987 12:00AM
11	7896	TQ456	Dec 12 1987 12:00AM
12	8042	QA879.1	May 22 1987 12:00AM
13	7066	A2976	May 24 1987 12:00AM
14	7131	P3087a	May 29 1987 12:00AM
15	7067	P2121	Jun 15 1987 12:00AM

salesdetails				
sonum int not null	qty_ordered smallint not null	qty_shipped smallint null	title_id char(6) not null	date_shipped date null
1	75	75	PS2091	Sep 15 1985 12:00AM
2	10	10	PS2091	Sep 15 1985 12:00AM
3	20	720	PS2091	Sep 18 1985 12:00AM
4	25	20	MC3021	Sep 18 1985 12:00AM
5	15	15	MC3021	Sep 14 1985 12:00AM
6	10	3	BU1032	Sep 22 1985 12:00AM

salesdetails				
sonum int not null	qty_ordered smallint not null	qty_shipped smallint null	title_id char(6) not null	date_shipped date null
7	3	3	PS2091	Sep 20 1985 12:00AM
8	5	5	BU1032	Sep 14 1985 12:00AM
9	25	5	BU1111	Mar 28 1988 12:00AM
19	35	35	BU2075	Mar 15 1988 12:00AM
10	15	15	BU7832	Oct 29 1987 12:00AM
11	10	10	MC2222	Jan 12 1988 12:00AM
12	30	30	PC1035	May 24 1987 12:00AM
13	50	50	PC8888	May 24 1987 12:00AM
14	20	20	PS1372	May 29 1987 12:00AM
14	25	25	PS2106	Apr 29 1987 12:00AM
14	15	10	PS3333	May 29 1987 12:00AM
14	25	25	PS7777	Jun 13 1987 12:00AM
15	40	40	TC3218	Jun 15 1987 12:00AM
15	20	20	TC4203	May 30 1987 12:00AM
15	20	10	TC7777	Jun 17 1987 12:00AM

editors								
ed_id char(11) not null	ed_fname varchar(40) not null	ed_lname varchar(20) not null	ed_pos varchar(12) null	phone char(12) null	address varchar(40) null	city varchar(20) null	state char(2) null	zip char(5) null
unique index edind	composite index ednmind							
321-55-8906	DeLongue	Martinella	project	415 843-2222	3000 6th St.	Berkeley	CA	94710
723-48-9010	Sparks	Manfred	copy	303 721-3388	15 Sail	Denver	CO	80237
777-02-9831	Samuelson	Bernard	project	415 843-6990	27 Yosemite	Oakland	CA	94609
777-66-9902	Almond	Alfred	copy	312 699-4177	1010 E. Devon	Chicago	IL	60018
826-11-9034	Himmel	Eleanore	project	617 423-0552	97 Bleaker	Boston	MA	02210
885-23-9140	Rutherford-Hayes	Hannah	project	301 468-3909	32 Rockbill Pike	Rockbill	MD	20852
993-86-0420	McCann	Dennis	acquisition	301 468-3909	32 Rockbill Pike	Rockbill	MD	20852
943-88-7920	Kaspchek	Christof	acquisition	415 549-3909	18 Severe Rd.	Berkeley	CA	94710
234-88-9720	Hunter	Amanda	acquisition	617 432-5586	18 Dowdy Ln.	Boston	MA	02210

titleditors		
ed_id char(11) not null	title_id char(6) not null	ed_ord tinyint null
unique composite index teind		
826-11-9034	BU2075	2
826-11-9034	PS2091	2
826-11-9034	PS2106	2
826-11-9034	PS3333	2
826-11-9034	PS7777	2
826-11-9034	PS1372	2
885-23-9140	MC2222	2
885-23-9140	MC3021	2
885-23-9140	TC3281	2
885-23-9140	TC4203	2
885-23-9140	TC7777	2
321-55-8906	BU1032	2
321-55-8906	BU1111	2
321-55-8906	BU7832	2
321-55-8906	PC1035	2
321-55-8906	PC8888	2
321-55-8906	BU2075	3
777-02-9831	PC1035	3
777-02-9831	PC8888	3
943-88-7920	BU1032	1

titleditors		
ed_id char(11) not null	title_id char(6) not null	ed_ord tinyint null
unique composite index teind		
943-88-7920	BU1111	1
943-88-7920	BU2075	1
943-88-7920	BU7832	1
943-88-7920	PC1035	1
943-88-7920	PC8888	1
993-86-0420	PS1372	1
993-86-0420	PS2091	1
993-86-0420	PS2106	1
993-86-0420	PS3333	1
993-86-0420	PS7777	1
993-86-0420	MC2222	1
993-86-0420	MC3021	1
993-86-0420	TC3218	1
993-86-0420	TC4203	1
993-86-0420	TC7777	1

royshed			
title_id char(6) not null	lorange int null	hirange int null	royalty float null
index rstdind			
BU1032	0	5000	0.100000
BU1032	5001	50000	0.120000
PC1035	0	2000	0.100000
PC1035	2001	4000	0.120000
PC1035	4001	50000	0.160000
BU2075	0	1000	0.100000
BU2075	1001	5000	0.120000
BU2075	5001	7000	0.160000
BU2075	7001	50000	0.180000
PS9999	0	50000	0.100000
PS2091	0	1000	0.100000
PS2091	1001	5000	0.120000
PS2091	5001	50000	0.140000
PS2106	0	2000	0.100000
PS2106	2001	5000	0.120000
PS2106	5001	50000	0.140000
MC3021	0	1000	0.100000
MC3021	1001	2000	0.120000
MC3021	2001	6000	0.140000
MC3021	6001	8000	0.180000
MC3021	8001	50000	0.200000
TC3218	0	2000	0.100000
TC3218	2001	6000	0.120000
TC3218	6001	8000	0.160000
TC3218	8001	50000	0.160000
PC8888	0	5000	0.100000
PC8888	5001	50000	0.120000
PS7777	0	5000	0.100000
PS7777	5001	50000	0.120000
PS3333	0	5000	0.100000
PS3333	5001	50000	0.120000
MC3026	0	1000	0.100000
MC3026	1001	2000	0.120000
MC3026	2001	6000	0.140000
MC3026	6001	8000	0.180000
MC3026	8001	50000	0.200000

royshed			
<i>title_id</i> <i>char(6)</i> <i>not null</i>	<i>lorange</i> <i>int</i> <i>null</i>	<i>hirange</i> <i>int</i> <i>null</i>	<i>royalty</i> <i>float</i> <i>null</i>
<i>index</i> <i>rstidind</i>			
BU1111	0	4000	0.100000
BU1111	4001	8000	0.120000
BU1111	8001	50000	0.140000
MC2222	0	2000	0.100000
MC2222	2001	4000	0.120000
MC2222	4001	8000	0.140000
MC2222	8001	12000	0.160000
TC7777	0	5000	0.100000
TC7777	5001	15000	0.120000
TC4203	0	2000	0.100000
TC4203	2001	8000	0.120000
TC4203	8001	16000	0.140000
BU7832	0	5000	0.100000
BU7832	5001	50000	0.120000
PS1372	0	50000	0.100000

ОПЕРАТОРЫ CREATE И INSERT ДЛЯ БАЗЫ ДАННЫХ BOOKBIZ

Для воссоздания базы данных *bookbiz* вы можете использовать приведенный ниже код (для Sybase Anywhere). Мы не гарантируем его правильную работу в других версиях SQL.

```

/*
Для SQL Anywhere 3/1/96
*/
set option date_format='Mmm dd yyyy hh:mmaa'
go
set option date_order = 'MDY'
go
set option scale = 2
go

drop table authors
go
create table authors
    (au_id char(11) not null,
    au_lname varchar(40) not null,
    au_fname varchar(20) not null,
    phone char(12) null,
    address varchar(40) null,
    city varchar(20) null,
    state char(2) null,
    zip char(5) null)
go
grant select on authors to public
go
drop table publishers
go
create table publishers
    (pub_id char(4) not null,

```

```

pub_name varchar(40) null,
address varchar(40) null,
city varchar(20) null,
state char(2) null)
go
grant select on publishers to public
go
drop table roysched
go
create table roysched
(title_id char(6) not null,
lorange int null,
hirange int null,
royalty float null)
go
grant select on roysched to public
go
drop table titleauthors
go
create table titleauthors
(au_id char(11) not null,
title_id char(6) not null,
au_ord tinyint null,
royaltyshare float null)
go
grant select on titleauthors to public
go
drop table titles
go
create table titles
(title_id char(6) not null,
title varchar(80) not null,
type char(12) null,
pub_id char(4) null,
price money null,
advance money null,
ytd_sales int null,
contract bit not null,
notes varchar(200) null,
pubdate date null)
go
grant select on titles to public
go
drop table editors
go
create table editors
(ed_id char(11) not null,
ed_lname varchar(40) not null,
ed_fname varchar(20) not null,
ed_pos varchar(12) null,
phone char(12) null,
address varchar(40) null,
city varchar(20) null,

```

```

state char(2) null,
zip char(5) null)
go
grant select on editors to public
go
drop table titleditors
go
create table titleditors
(ed_id char(11) not null,
title_id char(6) not null,
ed_ord tinyint null)
go
grant select on titleditors to public
go
drop table sales
go
create table sales
(sonum int not null,
stor_id char(4) not null,
ponum varchar(20) not null,
sdate date null)
go
grant select on sales to public
go
drop table salesdetails
go
create table salesdetails
(sonum int not null,
qty_ordered smallint not null,
qty_shipped smallint null,
title_id char(6) not null,
date_shipped date null)
go
grant select on salesdetails to public
go
create unique index pubind on publishers
(pub_id)
go
create unique index auidind
on authors (au_id)
go
create index aunmind
on authors (au_lname, au_fname)
go
create unique index titleidind
on titles (title_id)
go
create index titleind
on titles (title)
go
create unique index taind
on titleauthors (au_id, title_id)
go

```

```

create unique index edind
on editors (ed_id)
go
create index ednmind
on editors (ed_lname, ed_fname)
go
create unique index teind
on titleditors (ed_id, title_id)
go
create index rstidind
on roysched (title_id)
go
insert into authors
values('409-56-7008', 'Bennet', 'Abraham',
'415 658-9932', '6223 Bateman St.', 'Berkeley', 'CA', '94705')
go
insert into authors
values ('213-46-8915', 'Green', 'Marjorie',
'415 986-7020', '309 63rd St. #411', 'Oakland', 'CA', '94618')
go
insert into authors
values('238-95-7766', 'Carson', 'Cheryl',
'415 548-7723', '589 Darwin Ln.', 'Berkeley', 'CA', '94705')
go
insert into authors
values('998-72-3567', 'Ringer', 'Albert',
'801 826-0752', '67 Seventh Av.', 'Salt Lake City', 'UT', '84152')
go
insert into authors
values('899-46-2035', 'Ringer', 'Anne',
'801 826-0752', '67 Seventh Av.', 'Salt Lake City', 'UT', '84152')
go
insert into authors
values('722-51-5454', 'DeFrance', 'Michel',
'219 547-9982', '3 Balding Pl.', 'Gary', 'IN', '46403')
go
insert into authors
values('807-91-6654', 'Panteley', 'Sylvia',
'301 946-8853', '1956 Arlington Pl.', 'Rockville', 'MD', '20853')
go
insert into authors
values('893-72-1158', 'McBadden', 'Heather',
'707 448-4982', '301 Putnam', 'Vacaville', 'CA', '95688')
go
insert into authors
values('724-08-9931', 'Stringer', 'Dirk',
'415 843-2991', '5420 Telegraph Av.', 'Oakland', 'CA', '94609')
go
insert into authors
values('274-80-9391', 'Straight', 'Dick',
'415 834-2919', '5420 College Av.', 'Oakland', 'CA', '94609')
go
insert into authors

```

```

values('756-30-7391', 'Karsen', 'Livia',
'415 534-9219', '5720 McAuley St.', 'Oakland', 'CA', '94609')
go
insert into authors
values('724-80-9391', 'MacFeather', 'Stearns',
'415 354-7128', '44 Upland Hts.', 'Oakland', 'CA', '94612')
go
insert into authors
values('427-17-2319', 'Dull', 'Ann',
'415 836-7128', '3410 Blonde St.', 'Palo Alto', 'CA', '94301')
go
insert into authors
values('672-71-3249', 'Yokomoto', 'Akiko',
'415 935-4228', '3 Silver Ct.', 'Walnut Creek', 'CA', '94595')
go
insert into authors
values('267-41-2394', 'O'Leary', 'Michael',
'408 286-2428', '22 Cleveland Av. #14', 'San Jose', 'CA', '95128')
go
insert into authors
values('472-27-2349', 'Gringlesby', 'Burt',
'707 938-6445', 'PO Box 792', 'Covelo', 'CA', '95428')
go
insert into authors
values('527-72-3246', 'Greene', 'Morningstar',
'615 297-2723', '22 Graybar House Rd.', 'Nashville', 'TN', '37215')
go
insert into authors
values('172-32-1176', 'White', 'Johnson',
'408 496-7223', '10932 Bigge Rd.', 'Menlo Park', 'CA', '94025')
go
insert into authors
values('712-45-1867', 'del Castillo', 'Innes',
'615 996-8275', '2286 Cram Pl. #86', 'Ann Arbor', 'MI', '48105')
go
insert into authors
values('846-92-7186', 'Hunter', 'Sheryl',
'415 836-7128', '3410 Blonde St.', 'Palo Alto', 'CA', '94301')
go
insert into authors
values('486-29-1786', 'Locksley', 'Chastity',
'415 585-4620', '18 Broadway Av.', 'San Francisco', 'CA', '94130')
go
insert into authors
values('648-92-1872', 'Blotchet-Halls', 'Reginald',
'503 745-6402', '55 Hillsdale Bl.', 'Corvallis', 'OR', '97330')
go
insert into authors
values('341-22-1782', 'Smith', 'Meander',
'913 843-0462', '10 Misisipi Dr.', 'Lawrence', 'KS', '66044')
go
insert into publishers
values('0736', 'New Age Books', '1 1st St', 'Boston', 'MA')

```

```
go
insert into publishers
values('0877', 'Binnet & Hardley', '2 2nd Ave.', 'Washington', 'DC')
go
insert into publishers
values('1389', 'Algodata Infosystems', '3 3rd Dr.', 'Berkeley', 'CA')
go
insert into roysched
values('BU1032', 0, 5000, .10)
go
insert into roysched
values('BU1032', 5001, 50000, .12)
go
insert into roysched
values('PC1035', 0, 2000, .10)
go
insert into roysched
values('PC1035', 2001, 4000, .12)
go
insert into roysched
values('PC1035', 4001, 50000, .16)
go
insert into roysched
values('BU2075', 0, 1000, .10)
go
insert into roysched
values('BU2075', 1001, 5000, .12)
go
insert into roysched
values('BU2075', 5001, 7000, .16)
go
insert into roysched
values('BU2075', 7001, 50000, .18)
go
insert into roysched
values('PS9999', 0, 50000, .10)
go
insert into roysched
values('PS2091', 0, 1000, .10)
go
insert into roysched
values('PS2091', 1001, 5000, .12)
go
insert into roysched
values('PS2091', 5001, 50000, .14)
go
insert into roysched
values('PS2106', 0, 2000, .10)
go
insert into roysched
values('PS2106', 2001, 5000, .12)
go
insert into roysched
```

```
values('PS2106', 5001, 50000, .14)
go
insert into roysched
values('MC3021', 0, 1000, .10)
go
insert into roysched
values('MC3021', 1001, 2000, .12)
go
insert into roysched
values('MC3021', 2001, 6000, .14)
go
insert into roysched
values('MC3021', 6001, 8000, .18)
go
insert into roysched
values('MC3021', 8001, 50000, .20)
go
insert into roysched
values('TC3218', 0, 2000, .10)
go
insert into roysched
values('TC3218', 2001, 6000, .12)
go
insert into roysched
values('TC3218', 6001, 8000, .16)
go
insert into roysched
values('TC3218', 8001, 50000, .16)
go
insert into roysched
values('PC8888', 0, 5000, .10)
go
insert into roysched
values('PC8888', 5001, 50000, .12)
go
insert into roysched
values('PS7777', 0, 5000, .10)
go
insert into roysched
values('PS7777', 5001, 50000, .12)
go
insert into roysched
values('PS3333', 0, 5000, .10)
go
insert into roysched
values('PS3333', 5001, 50000, .12)
go
insert into roysched
values('MC3026', 0, 1000, .10)
go
insert into roysched
values('MC3026',1001, 2000, .12)
go
```

```
insert into roysched
values('MC3026', 2001, 6000, .14)
go
insert into roysched
values('MC3026', 6001, 8000, .18)
go
insert into roysched
values('MC3026', 8001, 50000, .20)
go
insert into roysched
values('BU1111', 0, 4000, .10)
go
insert into roysched
values('BU1111', 4001, 8000, .12)
go
insert into roysched
values('BU1111', 8001, 50000, .14)
go
insert into roysched
values('MC2222', 0, 2000, .10)
go
insert into roysched
values('MC2222', 2001, 4000, .12)
go
insert into roysched
values('MC2222', 4001, 8000, .14)
go
insert into roysched
values('MC2222', 8001, 12000, .16)
go
insert into roysched
values('TC7777', 0, 5000, .10)
go
insert into roysched
values('TC7777', 5001, 15000, .12)
go
insert into roysched
values('TC4203', 0, 2000, .10)
go
insert into roysched
values('TC4203', 2001, 8000, .12)
go
insert into roysched
values('TC4203', 8001, 16000, .14)
go
insert into roysched
values('BU7832', 0, 5000, .10)
go
insert into roysched
values('BU7832', 5001, 50000, .12)
go
insert into roysched
values('PS1372', 0, 50000, .10)
```

```
go
insert into titleauthors
values('409-56-7008', 'BU1032', 1, .60)
go
insert into titleauthors
values('486-29-1786', 'PS7777', 1, 1.00)
go
insert into titleauthors
values('486-29-1786', 'PC9999', 1, 1.00)
go
insert into titleauthors
values('712-45-1867', 'MC2222', 1, 1.00)
go
insert into titleauthors
values('172-32-1176', 'PS3333', 1, 1.00)
go
insert into titleauthors
values('213-46-8915', 'BU1032', 2, .40)
go

insert into titleauthors
values('238-95-7766', 'PC1035', 1, 1.00)
go
insert into titleauthors
values('213-46-8915', 'BU2075', 1, 1.00)
go

insert into titleauthors
values('998-72-3567', 'PS2091', 1, .50)
go
insert into titleauthors
values('899-46-2035', 'PS2091', 2, .50)
go
insert into titleauthors
values('998-72-3567', 'PS2106', 1, 1.00)
go
insert into titleauthors
values('722-51-5454', 'MC3021', 1, .75)
go
insert into titleauthors
values('899-46-2035', 'MC3021', 2, .25)
go
insert into titleauthors
values('807-91-6654', 'TC3218', 1, 1.00)
go
insert into titleauthors
values('274-80-9391', 'BU7832', 1, 1.00)
go
insert into titleauthors
values('427-17-2319', 'PC8888', 1, .50)
go
insert into titleauthors
values('846-92-7186', 'PC8888', 2, .50)
```

```

go
insert into titleauthors
values('756-30-7391', 'PS1372', 1, .75)
go
insert into titleauthors
values('724-80-9391', 'PS1372', 2, .25)
go
insert into titleauthors
values('724-80-9391', 'BU1111', 1, .60)
go
insert into titleauthors
values('267-41-2394', 'BU1111', 2, .40)
go
insert into titleauthors
values('672-71-3249', 'TC7777', 1, .40)
go
insert into titleauthors
values('267-41-2394', 'TC7777', 2, .30)
go
insert into titleauthors
values('472-27-2349', 'TC7777', 3, .30)
go
insert into titleauthors
values('648-92-1872', 'TC4203', 1, 1.00)
go
insert into titles
values ('PC8888', 'Secrets of Silicon Valley',
'popular_comp', '1389', $20.00, $8000.00, 4095,1,
'Muckraking reporting on the world's largest computer hardware and
software manufacturers.',
'06/12/85')
go
insert into titles
values ('BU1032', 'The Busy Executive's Database Guide',
'business', '1389', $19.99, $5000.00, 4095, 1,
'An overview of available database systems with emphasis on common
business applications. Illustrated.',
'06/12/85')
go
insert into titles
values ('PS7777', 'Emotional Security: A New Algorithm',
'psychology', '0736', $7.99, $4000.00, 3336, 1,
'Protecting yourself and your loved ones from undue emotional stress in
the modern world. Use of computer and nutritional aids emphasized.',
'06/12/85')
go
insert into titles
values ('PS3333', 'Prolonged Data Deprivation: Four Case Studies',
'psychology', '0736', $19.99, $2000.00, 4072,1,
'What happens when the data runs dry? Searching evaluations of
information-shortage effects.',
'06/12/85')
go

```

```

insert into titles
values ('BU1111', 'Cooking with Computers: Surreptitious Balance Sheets',
'business', '1389', $11.95, $5000.00, 3876, 1,
'Helpful hints on how to use your electronic resources to the best
advantage.', '06/09/85')
go
insert into titles
values ('MC2222', 'Silicon Valley Gastronomic Treats',
'mod_cook', '0877', $19.99, $0.00, 2032, 1,
'Favorite recipes for quick, easy, and elegant meals tried and tested by
people who never have time to eat, let alone cook.',
'06/09/85')
insert into titles
values ('TC7777', 'Sushi, Anyone?',
'trad_cook', '0877', $14.99, $8000.00, 4095, 1,
'Detailed instructions on improving your position in life by learning how
to make authentic Japanese sushi in your spare time. 5-10% increase in
number of friends per recipe reported from beta test. ',
'06/12/85')
go
insert into titles
values ('TC4203', 'Fifty Years in Buckingham Palace Kitchens',
'trad_cook', '0877', $11.95, $4000.00, 15096, 1,
'More anecdotes from the Queen''s favorite cook describing life among
English royalty. Recipes, techniques, tender vignettes.',
'06/12/85')
go
insert into titles
values ('PC1035', 'But Is It User Friendly?',
'popular_comp', '1389', $22.95, $7000.00, 8780, 1,
'A survey of software for the naive user, focusing on the ''friendliness''
of each.',
'06/30/85')
go
insert into titles
values('BU2075', 'You Can Combat Computer Stress!',
'business', '0736', $2.99, $10125.00, 18722, 1,
'The latest medical and psychological techniques for living with the
electronic office. Easy-to-understand explanations.',
'06/30/85')
go
insert into titles
values('PS2091', 'Is Anger the Enemy?',
'psychology', '0736', $10.95, $2275.00, 2045, 1,
'Carefully researched study of the effects of strong emotions on the body.
Metabolic charts included.',
'06/15/85')
go
insert into titles
values('PS2106', 'Life Without Fear',
'psychology', '0736', $7.00, $6000.00, 111, 1,

```

```

'New exercise, meditation, and nutritional techniques that can reduce the
shock of daily interactions. Popular audience. Sample menus
included, exercise video available separately.',
'10/05/85')
go
insert into titles
values('MC3021', 'The Gourmet Microwave',
'mod_cook', '0877', $2.99, $15000.00, 22246, 1,
'Traditional French gourmet recipes adapted for modern microwave
cooking.',
'06/18/85')
go
insert into titles
values('TC3218',
'Onions, Leeks, and Garlic: Cooking Secrets of the Mediterranean',
'trad_cook', '0877', $20.95, $7000.00, 375, 1,
'Profusely illustrated in color, this makes a wonderful gift book for a
cuisine-oriented friend.',
'10/21/85')
go
insert into titles (title_id, title, pub_id, contract)
values('MC3026', 'The Psychology of Computer Cooking', '0877', 0)
go
insert into titles
values ('BU7832', 'Straight Talk About Computers',
'business', '1389', $19.99, $5000.00, 4095, 1,
'Annotated analysis of what computers can do for you: a no-hype guide for
the critical user.',
'06/22/85')
go
insert into titles
values('PS1372',
'Computer Phobic and Non-Phobic Individuals: Behavior Variations',
'psychology', '0736', $21.59, $7000.00, 375, 1,
'A must for the specialist, this book examines the difference between
those who hate and fear computers and those who think they are swell.',
'10/21/85')
go
insert into titles (title_id, title, type, pub_id, contract, notes)
values('PC9999', 'Net Etiquette', 'popular_comp', '1389', 0,
'A must-read for computer conferencing debutantes!')
go
insert into editors
values ('321-55-8906', 'DeLongue', 'Martinella', 'project',
'415 843-2222', '3000 6th St.', 'Berkeley', 'CA', '94710')
go
insert into editors
values ('723-48-9010', 'Sparks', 'Manfred', 'copy',
'303 721-3388', '15 Sail', 'Denver', 'CO', '80237')
go
insert into editors
values ('777-02-9831', 'Samuelson', 'Bernard', 'project',
'415 843-6990', '27 Yosemite', 'Oakland', 'CA', '94609')

```

```

go
insert into editors
values ('777-66-9902', 'Almond', 'Alfred', 'copy',
'312 699-4177', '1010 E. Devon', 'Chicago', 'IL', '60018')
go
insert into editors
values ('826-11-9034', 'Himmel', 'Eleanore', 'project',
'617 423-0552', '97 Bleaker', 'Boston', 'MA', '02210')
go
insert into editors
values ('885-23-9140', 'Rutherford-Hayes', 'Hannah', 'project',
'301 468-3909', '32 Rockbill Pike', 'Rockbill', 'MD', '20852')
go
insert into editors
values ('993-86-0420', 'McCann', 'Dennis', 'acquisition',
'301 468-3909', '32 Rockbill Pike', 'Rockbill', 'MD', '20852')
go
insert into editors
values ('943-88-7920', 'Kaspchek', 'Christof', 'acquisition',
'415 549-3909', '18 Severe Rd.', 'Berkeley', 'CA', '94710')
go
insert into editors
values ('234-88-9720', 'Hunter', 'Amanda', 'acquisition',
'617 432-5586', '18 Dowdy Ln.', 'Boston', 'MA', '02210')
go
insert into titleditors values
('826-11-9034', 'BU2075', 2)
go
insert into titleditors values
('826-11-9034', 'PS2091', 2)
go
insert into titleditors values
('826-11-9034', 'PS2106', 2)
go
insert into titleditors values
('826-11-9034', 'PS3333', 2)
go
insert into titleditors values
('826-11-9034', 'PS7777', 2)
go
insert into titleditors values
('826-11-9034', 'PS1372', 2)
go
insert into titleditors values
('885-23-9140', 'MC2222', 2)
go
insert into titleditors values
('885-23-9140', 'MC3021', 2)
go
insert into titleditors values
('885-23-9140', 'TC3281', 2)
go
insert into titleditors values

```

```
('885-23-9140', 'TC4203', 2)
go
insert into titleditors values
('885-23-9140', 'TC7777', 2)
go
insert into titleditors values
('321-55-8906', 'BU1032', 2)
go
insert into titleditors values
('321-55-8906', 'BU1111', 2)
go
insert into titleditors values
('321-55-8906', 'BU7832', 2)
go
insert into titleditors values
('321-55-8906', 'PC1035', 2)
go
insert into titleditors values
('321-55-8906', 'PC88888', 2)
go
insert into titleditors values
('321-55-8906', 'BU2075', 3)
go
insert into titleditors values
('777-02-9831', 'PC1035', 3)
go
insert into titleditors values
('777-02-9831', 'PC8888', 3)
go
insert into titleditors values
('943-88-7920', 'BU1032', 1)
go
insert into titleditors values
('943-88-7920', 'BU1111', 1)
go
insert into titleditors values
('943-88-7920', 'BU2075', 1)
go
insert into titleditors values
('943-88-7920', 'BU7832', 1)
go
insert into titleditors values
('943-88-7920', 'PC1035', 1)
go
insert into titleditors values
('943-88-7920', 'PC8888', 1)
go
insert into titleditors values
('993-86-0420', 'PS1372', 1)
go
insert into titleditors values
('993-86-0420', 'PS2091', 1)
go
```

```

insert into titleditors values
('993-86-0420', 'PS2106', 1)
go
insert into titleditors values
('993-86-0420', 'PS3333', 1)
go
insert into titleditors values
('993-86-0420', 'PS7777', 1)
go
insert into titleditors values
('993-86-0420', 'MC2222', 1)
go
insert into titleditors values
('993-86-0420', 'MC3021', 1)
go
insert into titleditors values
('993-86-0420', 'TC3218', 1)
go
insert into titleditors values
('993-86-0420', 'TC4203', 1)
go
insert into titleditors values
('993-86-0420', 'TC7777', 1)
go

insert into sales
values(1,'7066', 'QA7442.3', '09/13/85')
go
insert into sales
values(2,'7067', 'D4482', '09/14/85')
go
insert into sales
values(3,'7131', 'N914008', '09/14/85')
go
insert into sales
values(4,'7131', 'N914014', '09/14/85')
go
insert into sales
values(5,'8042', '423LL922', '09/14/85')
go
insert into sales
values(6,'8042', '423LL930', '09/14/85')
go
insert into sales
values(7, '6380', '722a', '09/13/85')
go
insert into sales
values(8,'6380', '6871', '09/14/85')
go
insert into sales
values(9,'8042', 'P723', '03/11/88')
go
insert into sales

```

```
values(19,'7896','X999', '02/21/88')
go
insert into sales
values(10,'7896','QQ2299', '10/28/87')
go
insert into sales
values(11,'7896','TQ456', '12/12/87')
go
insert into sales
values(12,'8042','QA879.1', '5/22/87')
go
insert into sales
values(13,'7066','A2976', '5/24/87')
go
insert into sales
values(14,'7131','P3087a', '5/29/87')
go
insert into sales
values(15,'7067','P2121', '6/15/87')
go
insert into salesdetails
values(1, 75, 75,'PS2091', '9/15/85')
go
insert into salesdetails
values(2, 10, 10,'PS2091', '9/15/85')
go
insert into salesdetails
values(3, 20, 720,'PS2091', '9/18/85')
go
insert into salesdetails
values(4, 25, 20,'MC3021', '9/18/85')
go
insert into salesdetails
values(5, 15, 15,'MC3021', '9/14/85')
go
insert into salesdetails
values(6, 10, 3,'BU1032', '9/22/85')
go
insert into salesdetails
values(7, 3, 3,'PS2091', '9/20/85')
go
insert into salesdetails
values(8, 5, 5,'BU1032', '9/14/85')
go
insert into salesdetails
values(9, 25, 5,'BU1111', '03/28/88')
go
insert into salesdetails
values(19, 35, 35,'BU2075', '03/15/88')
go
insert into salesdetails
values(10, 15, 15,'BU7832', '10/29/87')
go
```

```

insert into salesdetails
values(11, 10, 10,'MC2222', '1/12/88')
go
insert into salesdetails
values(12, 30, 30,'PC1035', '5/24/87')
go
insert into salesdetails
values(13, 50, 50,'PC8888', '5/24/87')
go
insert into salesdetails
values(14, 20, 20,'PS1372', '5/29/87')
go
insert into salesdetails
values(14, 25, 25,'PS2106', '4/29/87')
go
insert into salesdetails
values(14, 15, 10,'PS3333', '5/29/87')
go
insert into salesdetails
values(14, 25, 25,'PS7777', '6/13/87')
go
insert into salesdetails
values(15, 40, 40,'TC3218', '6/15/87')
go
insert into salesdetails
values(15, 20, 20,'TC4203', '5/30/87')
go
insert into salesdetails
values(15, 20, 10,'TC7777', '6/17/87')
go
drop view titleview
go
create view titleview
as
select title, au_ord, au_lname,
price, ytd_sales, pub_id
from authors, titles, titleauthors
where authors.au_id = titleauthors.au_id
and titles.title_id = titleauthors.title_id
go
drop view oaklanders
go
create view oaklanders
as
select au_fname, au_lname, title
from authors, titles, titleauthors
where authors.au_id = titleauthors.au_id
and titles.title_id = titleauthors.title_id
and city = 'Oakland'
go
drop view books
go
create view books

```

```

as
select titles.title_id, au_ord, au_lname, au_fname
from authors, titles, titleauthors
where authors.au_id=titleauthors.au_id and
      titles.title_id = titleauthors.title_id
go
drop view royaltychecks
go
create view royaltychecks
as
select au_lname, au_fname,
      sum(price*ytd_sales*royalty*royaltyshare) as Total_Income
from authors, titles, titleauthors, roysched
where authors.au_id=titleauthors.au_id
      and titles.title_id = titleauthors.title_id
      and titles.title_id = roysched.title_id
      and ytd_sales between lorange and hirange
group by au_lname, au_fname
go
drop view currentinfo
go
create view currentinfo (PUB#, TYPE, INCOME,
      AVG_PRICE, AVG_SALES)
as
select pub_id, type, sum(price * ytd_sales),
      avg(price), avg(ytd_sales)
from titles
group by pub_id, type
go
drop view hiprice
go
create view hiprice
as
select *
from titles
where price > $15
      and advance > $5000
go
drop view cities
go
create view cities (Author, Authorcity, Pub, Pubcity)
as
select au_lname, authors.city, pub_name, publishers.city
from authors, publishers
where authors.city = publishers.city
go
drop view highaverage
go
create view highaverage
as
select authors.au_id, titles.title_id, pub_name, price
from authors, titleauthors, titles, publishers
where authors.au_id = titleauthors.au_id and

```

```

titles.title_id = titleauthors.title_id and
titles.pub_id = publishers.pub_id and
price >
    (select avg(price)
     from titles)
go
drop view highBandH
go
create view highBandH
as select *
from highaverage
where pub_name = 'Binnet & Hardley'
go
drop view number1
go
create view number1
as select au_lname, phone
from authors
where zip like '94%'
go
drop view number2
go
create view number2
as select au_lname, phone
from number1
where au_lname like '[M-Z]%'
go
drop view number3
go
create view number3
as select au_lname, phone
from number2
where au_lname = 'MacFeather'
go
drop view accounts
go
create view accounts (title, advance, gross_sales)
as
select title_id, advance, price * ytd_sales
from titles
where price > $15
    and advance > $5000
go
drop view categories
go
create view categories (Category, Average_Price)
as select type, avg(price)
from titles
group by type
go

/*Конец*/

```

Список литературы

Дейт, К.

Введение в системы баз данных.—М.:Наука.—1980.— 463с.

Мартин, Дж.

Организация баз данных в вычислительных системах.—М.:Наука.—1980.— 560с.

ANSI SQL Standart.

The 1992 ISO-ANSI SQL standart is available through ANSI as document X3.135-1992 and through ISO as document ISO/IEC 9075:1992.

ANSI is located at 11 West 42nd Street, New York, N. Y., 10036; 212-642-4900.

Codd, E.F.

“*A Relational Model of Data for Large Shared Data Banks*”, Communications of the ACM 13, no. 6 (June 1970) 377-87.

Codd, E.F.

“*Is Your DBMS Really Rational?*” Computerworld, October 14, 1985, 1-9. Authors proposes twelve criteria for testing relational database management systems.

Codd, E.F.

“*Does Your DBMS Run by the Rules?*” Computerworld, October 21, 1985, 49-55. Details on the thirty essential features of the relational model.

Date, C.J.

Database: A Primer. Reading, Mass.: Addison-Wesley Publishing Company, 1983.

Date, C.J.

An Introduction to Database Systems, Volume 2, Reading, Mass.: Addison-Wesley Publishing Company, 1983.

Date, C.J.

An Introduction to Database Systems, vol. 1, 4th ed., Reading, Mass.: Addison -Wesley Publishing Company, 1986.

Date, C.J.

Relational Database: Selected Writings. Reading, Mass.: Addison-Wesley Publishing Company, 1986.

Date, C.J.

A Guide to the SQL Standart. Reading, Mass.: Addison-Wesley Publishing Company, 1987. The third edition (1993) is written with Hugh Darwin and includes SQL-92.

Epstein, Robert

Relational Performance: Understanding the Performance of Relational DBMSs. Emeryville, Calif.: Sybase, Inc., 1986. Photocopy.

Gane, Chris

Developing Business Systems in SQL Using ORACLE on the IBM-PC. New York: Rapid System Development, Inc., 1986.

IBM Corporation

SQL/Data System Terminal User's Reference for VSE. Endicott, N.Y.: IBM Corporation, 1984.

Informix Software, Inc.

Informix Guide to SQL: Reference. Englewood Cliffs, N.J.: Informix Press, Prentice-Hall, Inc., 1995

Kent, William

“*A Simple Guide to Five Normal Forms in Relational Database Theory*”, in Communications of the ACM 20, no.2 (February 1983): 120-25

Koch, George, and Kevin Loney

ORACLE: The Complete Reference, 3rd ed. Berkeley, Calif.: Osborne McGraw-Hill, 1995.

Larson, Bruce L.

The Database Expert's Guide to Database 2. New York: McGraw-Hill Book Company, 1988.

Melton, Jim and Alan R Simon

Understanding the New SQL: A Complete Guided. San Mateo, Calif.: Morgan Kaufmann Publishers, 1993.

Microsoft Corporation

Transact-SQL Reference: Microsoft SQL Server Version 6.0. Redmond, Wash.: Microsoft Corporation, 1995.

Perkinson, R. C.

Data Analysis: The Key to Data Base Design. Wellesley, Mass.: QED Information Sciences, Inc., 1984.

Relational Database Systems, Inc.

INFORMIX-SQL Reference Manual. Menlo Park, Calif.: Relational Database Systems, Inc., 1986.

Relational Technology

INGRES Quick Reference Summary SQL Release 5.0. Alameda, Calif.: Relational Technology, 1986.

Ross, Ronald G.

Entity Modeling: Techniques and Application. Boston: Database Research Group, Inc., 1987.

Sachs, Jonathon, et al.

SQL Plus Reference Guide.* Belmont, Calif.: ORACLE Corporation, 1987.

Sybase, Inc.

Sybase SQL Server Reference Manual, vols.1 and 2. Emeryville, Calif.: Sybase, Inc., 1995.

Sybase, Inc.

Sybase SQL Anywhere: User's Guide, vols.1 and 2. Emeryville, Calif.: Sybase, Inc., 1995.

van der Lans, Rick E.

Introdaction to SQL. Reading, Mass.: Addison-Wesley Publishing Company, 1988.

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

А

агрегирующая функция, 100
администратор базы данных, 23
администрирование данных, 17
аргумент, 114
атрибут, 16

Б

база данных, 16
базовая таблица, 20; 183
блокировка, 208

В

виртуальная таблица, 22
включающий диапазон, 90
владелец, 23
вложенная сортировка, 103
вложенный запрос, 72
внешнее объединение, 146
внешний ключ, 28
восстановление, 200
вспомогательная таблица, 37
выборка данных, 17
выражение, 67

Г

главная таблица, 37
групповой индекс, 56

Д

декартово произведение, 143
декомпозиция без потерь, 27
диаграмма зависимостей между объектами, 27
домен, 215

Е

естественное объединение, 147

З

запись, 16
запрос, 18
значение, 16

И

идентификатор, 45
индекс, 17

К

команда, 17
команды управления данными, 19
контроль совпадений, 200
кортеж, 16
курсор, 22
кэш данных, 213

Л

логическая независимость, 17
логические операторы, 86

М

моделирование зависимостей, 26
модификация данных, 17

Н

набор символов, 100
назначение полномочий, 200
немодифицированный оператор
сравнения, 174
непроцедурный язык программирования, 19
нормализация, 26
нормальная форма, 36

О

общий подязык данных, 17
объект, 16
объектная целостность, 24
ограничение, 19
оператор, 17
определение данных, 17
оптимизатор запросов, 213
отношение, 16

П

первичный ключ, 16
подзапрос, 159
поисковая таблица, 42

поле, 16
пользовательская таблица, 16
порядок сортировки, 100
права на доступ и модификацию
данных, 23
правило, 58
проектирование базы данных, 25
производная таблица, 20
просматриваемая таблица, 183
псевдоним, 83

Р

различаемый ноль, 136

С

группированный курсор, 197
системная таблица, 16
системный администратор, 23
системный журнал транзакций, 209
системный каталог, 16
сканирование таблицы, 58
составной индекс, 56
список выбора, 75
список таблиц, 83
сравнение с эталоном, 211
ссылочная целостность, 24
столбец, 16
столбец соединения, 141
стратегия доступа, 17
строка, 16
структура данных, 27
сущность, 16
схема, 48

Т

таблица, 16
терминатор, 47
транзакция, 207
триггерные действия, 218
триггерные условия, 218

У

уникальный индекс, 56
управление транзакциями, 24; 207
устройство базы данных, 48

Ф

файл, 16
физическая независимость данных,
17
фиктивное значение, 67
форма, 64

Ц

целостность, 24

Ш

шаблон, 96

Э

экземпляр, 31

ОГЛАВЛЕНИЕ

Предисловие	5
Предисловие ко второму и третьему изданиям	7
Введение	9
Глава 1	
SQL и управление реляционными базами данных	15
УПРАВЛЕНИЕ РЕЛЯЦИОННЫМИ БАЗАМИ ДАННЫХ	15
Реляционная модель: одни таблицы	16
Независимость	17
Язык высокого уровня	17
Реляционные операции	19
Альтернативный способ просмотра данных	22
Нули	23
Безопасность	23
Целостность	24
ПРИСТУПАЯ К ПРОЕКТИРОВАНИЮ БАЗЫ ДАННЫХ	24
Глава 2	
Проектирование баз данных	25
СТРУКТУРА БАЗЫ ДАННЫХ	25
Как подходить к проектированию базы данных	26
Что такое “хорошая структура”	28
Описание нашей базы данных	29
ДАнные И ВЗАИМОСВЯЗИ	30
Объекты	30
Отношение один-ко-многим	32
Отношение многих-ко-многим	34
Отношение один-к-одному	35
Последние замечания к объектному подходу	35
РУКОВОДСТВО ПО НОРМАЛИЗАЦИИ	36
Первая нормальная форма	37
Вторая нормальная форма	38
Третья нормальная форма	38
Четвертая и пятая нормальные формы	40
ОБЗОР БАЗЫ ДАННЫХ	41
Последние замечания о базе данных bookbiz	42
Проверка структуры базы данных	44
Рассмотрение других понятий из области баз данных	44
Реализация структуры	44
Глава 3	
Создание и заполнение базы данных	45
СИНТАКСИС SQL	45
Обработка ошибок	47
СОЗДАНИЕ БАЗ ДАННЫХ	48
Выбор базы данных	49
СОЗДАНИЕ ТАБЛИЦ	49
Выбор типа данных	51
Оглавление	313

Назначение нулевого статуса	53
Процесс создания таблицы	54
СОЗДАНИЕ ИНДЕКСОВ	55
Оператор CREATE INDEX	55
Как, что и зачем нужно индексировать	57
СОЗДАНИЕ ТАБЛИЦ С ПОМОЩЬЮ ОГРАНИЧЕНИЙ SQL-92	58
ИЗМЕНЕНИЕ И УДАЛЕНИЕ БАЗ ДАННЫХ И ИХ ОБЪЕКТОВ	62
Изменение баз данных	62
Изменение определений таблицы	62
Удаление базы данных	63
Удаление таблиц	63
Удаление индекса	63
ДОБАВЛЕНИЕ, ИЗМЕНЕНИЕ И УДАЛЕНИЕ ДАННЫХ	64
Добавление новой строки	64
Использование оператора SELECT в команде INSERT	66
ИЗМЕНЕНИЕ СУЩЕСТВУЮЩИХ ДАННЫХ	68
Оператор UPDATE	68
Предложение SET	68
Предложение WHERE	69
УДАЛЕНИЕ ДАННЫХ: КОМАНДА DELETE	70
ПРИСТУПАЯ К ВЫБОРКЕ ДАННЫХ	71
Глава 4	
Выборка информации из базы данных	72
ПЕРЕД ВЫБОРОМ	72
Синтаксис оператора SELECT	72
ВЫБОР СТОЛБЦОВ: СПИСОК ВЫБОРА	75
Выбор всех столбцов: SELECT *	75
Выбор отдельных столбцов	77
Выражения: больше, чем просто имена столбцов	77
УКАЗАНИЕ ТАБЛИЦ: СПИСОК ТАБЛИЦ	83
ВЫБОР СТРОК: ПРЕДЛОЖЕНИЕ WHERE	84
Операторы сравнения	84
Совместное использование условных и логических операторов	86
Диапазоны (BETWEEN и NOT BETWEEN)	90
Списки (IN и NOT IN)	92
Выборка нулевых значений	94
Поиск по подстрокам: предложение LIKE	96
ЧТО ДАЛЬШЕ	99
Глава 5	
Сортировка данных и другие методы выбора	100
ДОПОЛНИТЕЛЬНЫЕ ВОЗМОЖНОСТИ ОПЕРАТОРА SELECT	100
СОРТИРОВКА РЕЗУЛЬТАТОВ ЗАПРОСА: ПРЕДЛОЖЕНИЕ ORDER BY	100
Порядок сортировки	100
Как выполняется сортировка	101
Синтаксис предложения ORDER BY	102
Сортировка внутри сортировки	102
Сортировка по возрастанию и по убыванию	104
А как насчет выражений?	105
Как сортировать нулевые значения	107

УСТРАНЕНИЕ ПОВТОРЯЮЩИХСЯ СТРОК: ПРЕДЛОЖЕНИЯ DISTINCT И ALL	108
Синтаксис предложения DISTINCT	109
Почувствуйте разницу!	109
АГРЕГИРУЮЩИЕ ФУНКЦИИ	112
Синтаксис агрегирующих функций	114
СКАЛЯРНЫЕ И ВЕКТОРНЫЕ ФУНКЦИИ	120
Глава 6	
Группировка данных и построение отчетов	121
ГРУППИРОВКА	121
ПРЕДЛОЖЕНИЕ GROUP BY	121
Синтаксис предложения GROUP BY	122
Упорядоченные группы	131
ПРЕДЛОЖЕНИЕ HAVING	132
Разновидности предложения HAVING	132
Предложения HAVING и WHERE	133
ЕЩЕ О НУЛЕВЫХ ЗНАЧЕНИЯХ	135
Нули и проектирование баз данных	136
Сравнение нулевых значений	136
Нули и вычисления	137
Нули и группы	138
Значения по умолчанию в качестве альтернативы нулевым значениям	138
РАБОТА С НЕСКОЛЬКИМИ ТАБЛИЦАМИ	140
Глава 7	
Объединение таблиц и сложный анализ данных	141
ЧТО ТАКОЕ ОБЪЕДИНЕНИЕ	141
Синтаксис операции объединения	141
ПОЧЕМУ НЕОБХОДИМО ОБЪЕДИНЕНИЕ	142
Объединения и реляционная модель	142
ПРИМЕР ОБЪЕДИНЕНИЯ	143
Проверка правильности объединения	143
КАК ПОЛУЧИТЬ ХОРОШЕЕ ОБЪЕДИНЕНИЕ	144
Объединения и нулевые значения	144
УЛУЧШЕНИЕ ЧИТАЕМОСТИ РЕЗУЛЬТАТОВ ОБЪЕДИНЕНИЯ	144
Выбор столбцов для запросов на объединение	145
Псевдонимы в списке таблиц улучшают читаемость запросов	146
ОПРЕДЕЛЕНИЕ УСЛОВИЙ ОБЪЕДИНЕНИЯ	146
Объединения, основанные на равенстве	146
Объединения, не основанные на равенствах	147
Объединение таблицы с самой собой: самообъединение	148
Использование при самообъединении оператора неравенства	150
Объединение нескольких таблиц	151
Внешние объединения	152
КАК ОБЪЕДИНЕНИЯ ОБРАБАТЫВАЮТСЯ СИСТЕМОЙ	154
ОПЕРАТОР UNION	155
Полезный трюк с оператором UNION	157
ПОДЗАПРОСЫ	158

Глава 8

Структурированные запросы и подзапросы	159
ЧТО ТАКОЕ ПОДЗАПРОС	159
Упрощенный синтаксис подзапроса	159
КАК РАБОТАЮТ ПОДЗАПРОСЫ	160
Некоррелированная обработка	161
Коррелированная обработка	162
ОБЪЕДИНЕНИЯ ИЛИ ПОДЗАПРОСЫ?	162
Подзапросы!	162
Объединения!	164
Подзапросы или самообъединения?	164
Что лучше?	165
ПРАВИЛА ПОДЗАПРОСОВ	165
ПОДЗАПРОСЫ, НЕ ВОЗВРАЩАЮЩИЕ ЗНАЧЕНИЙ ИЛИ ВОЗВРАЩАЮЩИЕ НЕСКОЛЬКО ЗНАЧЕНИЙ	166
Подзапросы, начинающиеся с IN	166
Подзапросы, начинающиеся с NOT IN	167
Коррелированные подзапросы с IN	168
Подзапросы, начинающиеся с операторов сравнения и включающие ключевые слова ANY или ALL	170
ПОДЗАПРОСЫ, ВОЗВРАЩАЮЩИЕ ЕДИНСТВЕННОЕ ЗНАЧЕНИЕ	174
Агрегирующие функции гарантируют единственное значение	175
Предложения GROUP BY и HAVING должны возвращать единственное значение	176
Коррелированные подзапросы с операторами сравнения	176
ПОДЗАПРОСЫ, ВЫПОЛНЯЮЩИЕ ПРОВЕРКУ НА СУЩЕСТВОВАНИЕ	177
NOT EXISTS отыскивает пустой набор	179
Использование EXISTS для поиска пересечения и разности	180
ПОДЗАПРОСЫ С РАЗНЫМИ УРОВНЯМИ ВЛОЖЕНИЯ	181
ПОДЗАПРОСЫ В ОПЕРАТОРАХ UPDATE, DELETE И INSERT В ПОЛЕ ЗРЕНИЯ КУРСОРА	182

Глава 9

Создание и использование виртуальных таблиц (курсоров)	183
КУРСОР ОБЕСПЕЧИВАЕТ ГИБКОСТЬ	183
СОЗДАНИЕ КУРСОРОВ	183
Удаление курсоров	184
ПРЕИМУЩЕСТВА КУРСОРОВ	184
Почему же все-таки курсор?	187
КАК РАБОТАЮТ КУРСОРЫ	189
Правила присвоения имен столбцам курсора	190
Создание курсоров с объединениями и подзапросами	191
Ограничения на создание курсоров	192
Предложение Check Option	192
Разборка курсора	194
Переопределение курсоров	194
МОДИФИКАЦИЯ ДАННЫХ ПОСРЕДСТВОМ КУРСОРОВ	196
Правила в соответствии с ANSI	196
СОЗДАНИЕ КОПИЙ ДАННЫХ	198
ВОПРОСЫ АДМИНИСТРИРОВАНИЯ БАЗ ДАННЫХ	199

Глава 10

Безопасность, транзакции, производительность и целостность	200
УПРАВЛЕНИЕ БАЗАМИ ДАННЫХ В РЕАЛЬНОМ МИРЕ	200
БЕЗОПАСНОСТЬ ДАННЫХ	201
Идентификация пользователя и особые пользователи	201
Команды GRANT и REVOKE	202
Курсоры как механизм обеспечения безопасности	206
ТРАНЗАКЦИИ	207
Транзакции и совпадения	208
Транзакции и восстановление	209
Транзакции, определяемые пользователем	209
Получение резервной копии и восстановление	210
ПРОИЗВОДИТЕЛЬНОСТЬ	211
Сравнение с эталоном	211
Структура и индексация	212
Запросы	213
Другие инструменты для мониторинга и повышения производительности	213
ЦЕЛОСТНОСТЬ ДАННЫХ	215
Ограничения на домен	215
Целостность объекта	216
Ссылочная целостность	217
ОТ АБСТРАКЦИЙ SQL К РЕАЛЬНОМУ МИРУ	220

Глава 11

Разрешение проблем	221
КАК ИСПОЛЬЗОВАТЬ SQL В СВОЕЙ РАБОТЕ	221
ФОРМАТИРОВАНИЕ И ОТОБРАЖЕНИЕ ДАННЫХ	222
Отображение одного поля в виде двух	222
Выравнивание строки символов по правому краю	224
Как указать число разрядов после десятичной точки	227
РАБОТА С ШАБЛОНАМИ	229
Сопоставление прописных и строчных букв	230
Поиск символьных данных заданного размера	231
Как найти данные типа дат	232
Замена пробелов на нули	234
ПОИСК ДАННЫХ С ПОМОЩЬЮ СЛОЖНЫХ ОБЪЕДИНЕНИЙ И ПОДЗАПРОСОВ	236
Сопоставление пар столбцов в разных таблицах	236
Поиск данных в определенном диапазоне, если вам не известны точные значения	238
Отображение данных в формате электронной таблицы	239
ПРЕДЛОЖЕНИЕ GROUP BY	243
Отображение данных по времени	243
ПОСЛЕДОВАТЕЛЬНЫЕ НОМЕРА	244
Нахождение максимального значения и добавление 1	245
Использование отдельной таблицы ключей	246
Использование произвольного значения	247
КАК ИЗБЕЖАТЬ ОШИБОК	247

Глава 12

Ошибки, и как их избежать	248
НЕТ, ВЫ НЕ ДУРАК	248
ПРЕДЛОЖЕНИЕ GROUP BY	248

Подсчет по единицам	249
ПРЕДЛОЖЕНИЯ WHERE И HAVING	249
Почему столько строк?	249
Сочетание значений строк и агрегирующих функций	253
Как избежать проблем с предложением HAVING	256
КЛЮЧЕВОЕ СЛОВО DISTINCT	258
DISTINCT со столбцами и выражениями	259
DISTINCT с агрегирующими функциями	260
DISTINCT и DISTINCT?	261
ДРУГИЕ НЕДОРАЗУМЕНИЯ	262
Удаление дубликатов	262
Нахождение “первого” входа	263
Приложение А	
Краткое описание синтаксиса SQL, используемого в книге	265
СОГЛАШЕНИЯ ПО СИНТАКСИСУ	265
ФОРМАТИРОВАНИЕ	265
Регистр	265
СПИСОК ОПЕРАТОРОВ	266
Приложение Б	
Аналогии между ключевыми словами разных диалектов SQL	267
СРАВНЕНИЕ СИНТАКСИСОВ	267
ОПРЕДЕЛЕНИЕ ДАННЫХ	267
Операторы базы данных	268
Создание и удаление объектов базы данных	269
МАНИПУЛЯЦИИ С ДАННЫМИ	271
АДМИНИСТРИРОВАНИЕ ДАННЫХ	273
Приложение В	
Словарь терминов	276
Приложение Г	
Описание базы данных bookbiz	284
ТАБЛИЦЫ	284
ОПЕРАТОРЫ CREATE И INSERT ДЛЯ БАЗЫ ДАННЫХ BOOKBIZ	290
Приложение Д	
Список литературы	309
Предметный указатель	311