ГАЛИНА МИРОШНИЧЕНКО

bhv

РЕЛЯЦИОННЫЕ БАЗЫ ДАННЫХ:

ПРАКТИЧЕСКИЕ ПРИЕМЫ ОПТИМАЛЬНЫХ РЕШЕНИЙ

CASE-СРЕДСТВА – МОДЕЛИРОВАНИЕ РЕЛЯЦИОННЫХ ДАННЫХ

НОРМАЛИЗАЦИЯ БАЗЫ ДАННЫХ – РАЗУМНЫЙ УРОВЕНЬ

ИНДЕКСЫ В БАЗЕ ДАННЫХ – ПОЛЬЗА И ВРЕД

ОПТИМИЗАТОР ЗАПРОСОВ SQL-СЕРВЕРА – ПОМОГАЕТ ИЛИ ВВОДИТ В ЗАБЛУЖДЕНИЕ

ПРОИЗВОДИТЕЛЬНОСТЬ ВЫПОЛНЕНИЯ ТРАНЗАКЦИЙ – О НЕЙ НАДО ПОМНИТЬ ВСЕГДА

PRO

ПРОФЕССИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ

+CD

Галина Мирошниченко

РЕЛЯЦИОННЫЕ БАЗЫ ДАННЫХ:

ПРАКТИЧЕСКИЕ ПРИЕМЫ ОПТИМАЛЬНЫХ РЕШЕНИЙ

Санкт-Петербург
«БХВ-Петербург»
2005

УДК 681.3.06
ББК 32.973.26
М64

Мирошниченко Г. А.

М64 Реляционные базы данных: практические приемы оптимальных решений. — СПб.: БХВ-Петербург, 2005. — 400 с.: ил.

ISBN 5-94157-551-3

Рассмотрены практические вопросы увеличения производительности функционирования клиент-серверных приложений путем использования различных технических приемов на этапах проектирования, разработки и сопровождения реляционной базы данных. Описаны программные средства и преимущества использования CASE-технологий при проектировании БД. Изложены вопросы логического и физического моделирования. Подробно рассмотрено использование индексов. Большое внимание уделено вопросам программирования: понятию об оптимизаторе и плане выполнения запросов; измерению времени выполнения запросов; использованию хранимых процедур, триггеров и многому другому. Приведены рекомендации по разработке транзакций, мониторингу активности пользователя и выявлению критических мест программы. Книга сопровождается многочисленными примерами. При этом на стадии проектирования используется CASE-средство Erwin, а на стадии реализации — MS SQL Server 2000.

CD содержит рассмотренные в книге примеры моделей, учебную базу данных, а также текстовые файлы с листингами программ на T-SQL.

Для программистов, студентов, бизнес-аналитиков и администраторов, работающих с клиент-серверными технологиями

УДК 681.3.06
ББК 32.973.26

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Игорь Шишигин*
Зав. редакцией	*Григорий Добин*
Редактор	*Наталья Довгулевич*
Компьютерная верстка	*Ольги Сергиенко*
Корректор	*Зинаида Дмитриева*
Дизайн серии	*Инны Тачиной*
Оформление обложки	*Игоря Цырульникова*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 27.06.05.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 32,25.
Тираж 3000 экз. Заказ №
"БХВ-Петербург", 194354, Санкт-Петербург, ул. Есенина, 5Б.

Санитарно-эпидемиологическое заключение на продукцию
№ 77.99.02.953.Д.006421.11.04 от 11.11.2004 г. выдано Федеральной службой
по надзору в сфере защиты прав потребителей и благополучия человека.

Отпечатано с готовых диапозитивов
в ГУП "Типография "Наука"
199034, Санкт-Петербург, 9 линия, 12

ISBN 5-94157-551-3

© Мирошниченко Г. А., 2005
© Оформление, издательство "БХВ-Петербург", 2005

Оглавление

Введение **9**

Описание компакт-диска 12

Глава 1. Моделирование реляционных данных **13**

1.1. Общая характеристика реляционной модели данных 13
1.1.1. Структурная часть реляционной базы данных 14
1.1.2. Целостная часть реляционной базы данных 16
Домены 16
Целостность сущностей 17
Целостность внешних ключей 19
1.1.3. Манипуляционная часть реляционной базы данных 21
1.2. Технология клиент-сервер 21
1.3. Общий обзор методологии проектирования информационных систем 26
1.4. Сбор информации для моделирования данных 29
1.5. Реляционная модель: концептуальная, логическая и физическая стадии 31
1.6. Нотации реляционной модели для методологии "сущность-связь" 33
1.6.1. Нотация Баркера, IDEF1X- и IE-нотации 34
1.6.2. Нотация Чена 38
1.6.3. Идентифицирующие и неидентифицирующие связи. Сильные и слабые сущности 39
1.7. Компоненты реляционной модели в методологии "сущность-связь" 40
1.7.1. Сущности 41
1.7.2. Атрибуты 43
1.7.3. Связи 45
Бинарные связи, их параметры и классификация 47
1.7.4. Основные этапы логического проектирования в методологии "сущность-связь" 48
1.8. Приемы построения проектных решений по ER-диаграммам 50
1.8.1. Бинарные связи с обязательным классом принадлежности 52
1.8.2. Бинарные связи с необязательным классом принадлежности 56
1.8.3. Связи более высокого порядка, чем бинарные 62
1.8.4. Ролевые связи 65
1.8.5. Некоторые специальные аспекты логического моделирования 69
Правила выбора первичных ключей сущностей 69

Связи без права передачи ... 70
Особенности работы со временем ... 71
1.9. Обзор возможностей CASE-средства Erwin ... 73
1.9.1. Интерфейс Erwin ... 75
Уровни отображения модели ... 76
Установка цвета и шрифта ... 77
Подмножества модели и сохраняемые отображения ... 78
1.9.2. Сущности и атрибуты в Erwin ... 78
Работа с редактором атрибутов ... 80
1.9.3. Связи в Erwin ... 81
Редактор связей, общие параметры бинарной связи ... 83
Редактор связей, ролевые имена атрибутов в бинарной связи ... 85
Редактор связей, ссылочные действия ... 87
Связь "много-ко-многим" ... 88
Категориальная связь "супертип/подтип" и иерархия наследования ... 88
Переход к физической модели и реализация наследования в реляционной базе данных ... 90
Рекурсивные связи ... 94
1.9.4. Работа с доменами и ключами ... 96
1.10. Примеры проектных решений в Erwin ... 97
1.10.1. Использование функциональных (ролевых) имен ... 97
1.10.2. Примеры реализации рекурсии ... 98
1.10.3. Сущности, вводимые только на логическом уровне ... 101
1.10.4. Пример анализа влияния проектного решения на последующие стадии жизненного цикла базы данных ... 105

Глава 2. Логическое проектирование и оптимизация ... 115

2.1. Бизнес-среда: OLTP- и OLAP-системы ... 116
2.2. Уровни моделирования реляционной базы данных ... 118
2.3. Критерии оценки качества логической модели ... 121
2.3.1. Адекватность базы данных предметной области ... 122
2.3.2. Скорость операций обновления данных ... 122
2.3.3. Скорость операций выборки данных ... 123
2.3.4. Легкость разработки и сопровождения базы данных ... 123
2.4. Избыточность данных ... 124
2.4.1. Избыточность, связанная с вычислимыми атрибутами ... 124
2.4.2. Примеры вычислимых атрибутов в простой и сложной транзакции ... 125
Пример 1. Вычислимый атрибут в простой транзакции ... 125
Пример 2. Вычислимые атрибуты в сложной транзакции ... 128
2.5. Критерий нормализации ... 129
2.5.1. Избыточность, связанная с нормализацией ... 130
2.5.2. Нормальные формы отношений ... 131
2.5.3. Функциональные зависимости ... 132
2.5.4. Вторая нормальная форма ... 133
2.5.5. Третья нормальная форма (НФ Бойса — Кодда) ... 134
2.5.6. Правила вывода ФЗ. Минимальное покрытие ... 135
2.5.7. Четвертая нормальная форма ... 136
2.5.8. Пятая нормальная форма ... 138

2.5.9. Алгоритм декомпозиции отношения с целью его нормализации.....140
2.5.10. Анализ критериев оптимальности реляционной базы данных.....145
2.6. Примеры нормализации, основанной на НФ 146
2.6.1. Приведение к 1 НФ.....146
2.6.2. Построение диаграмм ФЗ и минимальных покрытий.....149
Пример 1. Построение простейшей диаграммы ФЗ..... 150
Пример 2. Построение диаграммы ФЗ с псевдотранзитивными зависимостями 150
2.7. Построение нормализованных проектных решений для задачи материального учета..... 152
2.7.1. Пример 1. Произвольные цены без фиксированной прибыли 153
2.7.2. Пример 2. Произвольные цены с фиксированной прибылью 154
2.7.3. Пример 3. Постоянные цены и история цен.....156

Глава 3. Физическое представление базы данных.....159

3.1. Понятие о системных ресурсах 161
3.2. Общие сведения о файлах SQL-сервера..... 161
3.2.1. Понятие группы файлов 162
3.2.2. Создание базы данных 164
3.3. Управление объемом базы данных..... 165
3.4. Представление о хранении данных на SQL-сервере..... 168
3.4.1. Способ хранения кластерных таблиц.....169
3.4.2. Работа с кучей 170
3.4.3. Вычисление размера таблицы кучи 172
3.5. Управление выделением экстентов и поиск свободного места 173
3.6. Понятие о кэшировании данных 174
3.7. Средства настройки SQL-сервера..... 176

Глава 4. Индексы как объект в реляционных базах данных 179

4.1. Структуры данных, ведущие к индексам 179
4.1.1. Механизм реализации кластерных индексов.....183
4.1.2. Механизм реализации некластерных индексов.....184
4.1.3. Пример поиска по диапазону для таблицы кучи 186
4.1.4. Пример поиска по диапазону при отображении некластерного индекса в ключ кластеризации.....186
4.2. Использование индексов с точки зрения бизнес-среды базы данных..... 189
4.3. Средства работы с индексами на MS SQL-сервере..... 191
4.3.1. Создание индексов.....191
4.3.2. Получение информации об индексах 195
Системная таблица *sysindexes* 195
Системные процедуры и средства DBCC 197
4.3.3. Сопровождение индексов 198

Глава 5. Программирование на SQL-сервере 205

5.1. Понятие об оптимизаторе запросов..... 208
5.1.1. Общая схема выполнения SQL-запроса 208
5.1.2. Графическое представление плана запроса SQL-сервером.....212
5.1.3. Оценка и измерение времени выполнения запроса 216

5.2. Управление статистикой ... 219
5.3. Общие рекомендации по использованию индексов SQL-сервером ... 224
5.3.1. Использование кластерных индексов ... 224
5.3.2. Использование некластерных индексов ... 226
5.3.3. Составные и покрывающие индексы ... 228
5.3.4. Использование мастера настройки индексов ... 230
5.4. Оптимизация поиска данных по заданному условию (опция *WHERE*) ... 234
5.4.1. Некоторые правила вычисления логических условий ... 235
5.4.2. Опция *WHERE* и агрегирование ... 236
5.4.3. Использование индексов с опцией *WHERE* ... 237
5.4.4. Примеры тестов для индексов с опцией *WHERE* ... 240
Пример 1. Влияние селективности столбца на использование индекса ... 240
Пример 2. Индексы по паре высокоселективных столбцов ... 243
5.5. Соединение таблиц ... 244
5.5.1. Способы соединения таблиц ... 246
5.6. Использование подсказок оптимизатору ... 249
5.6.1. Подсказки таблицы ... 251
5.6.2. Подсказки запроса ... 253
5.7. Хранимые процедуры и производительность SQL-сервера ... 254
5.7.1. Управление перекомпиляцией и кэшированием ... 256
5.7.2. Другие рекомендации по оптимизации хранимых процедур ... 261
5.7.3. Пример использования утилиты SQL Profiler для мониторинга хранимых процедур ... 262
5.8. Автогенерация данных ... 269
5.8.1. Пример 1. Одна таблица с автоинкрементным ключом ... 270
5.8.2. Пример 2. Пара таблиц, связанных FK-ограничением ... 273
5.8.3. Пример 3. Дочерняя таблица с составным первичным ключом и с парой FK-ограничений ... 277
5.8.4. Пример 4. Работа с графическими данными ... 280
5.9. Оптимизация быстродействия для некоторых типичных запросов ... 285
5.9.1. Пример 1. Поиск максимального значения ... 285
5.9.2. Пример 2. Поиск значений, отсутствующих во внешнем ключе ... 290
5.9.3. Пример 3. Поиск максимума от агрегатной функции ... 294
5.9.4. Пример 4. Сложное агрегирование ... 296
5.10. Использование курсоров ... 299
5.10.1. Примеры использования курсоров ... 300
Пример 1. Использование курсора для задачи администрирования ... 301
Пример 2. Замена курсора табличной обработкой ... 302
5.11. Использование триггеров ... 305
5.11.1. Пример 1. Триггер на вычисление итоговых значений ... 306
5.11.2. Пример 2. Триггер на представление ... 307
5.11.3. Пример 3. Триггер для сохранения данных о сделанных изменениях ... 310
5.12. Использование представлений ... 310
5.12.1. Индексированные представления ... 311

Глава 6. Транзакции и производительность ... 313

6.1. Основные понятия, связанные с транзакциями ... 313
6.1.1. Атомарность и согласованность транзакций ... 314

6.1.2. Изолированность транзакций ... 315
Транзакции и параллелизм ... 316
Конфликты между транзакциями ... 317
Пример потери результатов обновления ... 318
Пример чтения "грязных" данных ... 318
Пример неповторяемого считывания ... 319
Пример появления фиктивных элементов (фантомов) ... 320
Пример несовместимого анализа ... 320
Блокировки и разрешение тупиковых ситуаций ... 322
6.1.3. Долговечность транзакций и восстановление данных ... 327
6.2. Представление о способах оценивания производительности транзакций ... 330
6.2.1. Пример 1. Генерация базы данных ... 333
6.2.2. Пример 2. Обслуживание приложения ... 336
6.3. Средства управления транзакциями на SQL-сервере ... 339
6.3.1. Явный и неявный режимы работы транзакций ... 340
6.3.2. Управление уровнем изоляции транзакций ... 343
Пример 1. Чтение данных с эффектом "грязного чтения" ... 345
Пример 2. Эффект "неповторяемого чтения" ... 346
Пример 3. Чтение данных с эффектом "фантомов" ... 347
6.3.3. Управление блокировками на SQL-сервере ... 348
Параметры настройки соединения, связанные с работой транзакций ... 348
Средства мониторинга блокировок на SQL-сервере ... 350
6.3.4. Общие рекомендации по программированию транзакций ... 353
Основные приемы, используемые при программировании транзакций ... 354
Вложенные транзакции ... 356
Другие рекомендации для сервера ... 362
Использование возможностей клиента ... 364

Приложение 1. Средства мониторинга производительности SQL-сервера ... 371

Использование утилиты SQL Profiler ... 372

Приложение 2. Классификация ограничений целостности и средства их реализации на SQL-сервере ... 385

Ограничения целостности по способам реализации ... 386
Ограничения целостности по времени проверки ... 387
Ограничения целостности по области действия ... 388
Ограничения целостности по классификации Microsoft для SQL-сервера ... 390

Список литературы ... 393

Предметный указатель ... 395

Введение

На сегодняшний день реляционные базы данных уверенно являются неотъемлемой частью большинства информационных систем. В этой области прикладного программирования стремительно развиваются новые технологии, платформы реализации и среды разработки приложений. Но тем не менее остается классическая часть и общепризнанный подход к процессу проектирования, разработки и сопровождения реляционной базы данных, разбитый на последовательные этапы, которые обычно приходится реализовывать.

С проблемой производительности так или иначе сталкиваются все разработчики программного обеспечения. Конечно, хорошо, если выбранная целевая система управления базами данных (СУБД) вместе с аппаратными средствами ее реализации предоставляет существенный запас по критерию производительности для конкретной прикладной задачи, так что над этими вопросами можно особенно не задумываться. Однако очевидным правилом является следующее: чем выше этот запас, тем сама СУБД вместе с ее аппаратной конфигурацией становится все более дорогостоящей. Поэтому работа на предельных нагрузках по различным параметрам производительности вполне возможна, и этими вопросами с неизбежностью приходится заниматься.

Основная идея данной книги состоит в том, чтобы рассмотреть решения, принимаемые на всех этапах процесса разработки клиент-серверного приложения с оперативной обработкой транзакций, иначе называемого OLTP-приложением (On-Line Transaction Processing) реляционной базы данных, расположенной на одном сервере, с точки зрения того, к какому уровню производительности эти решения приведут. Для каждого из этих этапов предлагаются стандартные приемы, позволяющие оптимизировать те или иные критерии, основным из которых является производительность информационной системы. Выбор предлагаемых решений объясняется, в необходимых случаях выполняется сравнение альтернативных вариантов, рассматриваются их достоинства и недостатки.

Проблемы, рассматриваемые в книге, актуальны для всех реляционных СУБД, реализующих тот или иной диалект структурного языка запросов SQL (Structured Query Language), и решаются в большинстве случаев аналогично. В нашем случае в качестве среды реализации выбрана СУБД под названием MS SQL-сервер 2000, который является одной из наиболее популярных клиент-серверных СУБД, обладающей развитыми средствами настройки оптимального функционирования приложения. Данная СУБД интенсивно развивается, анонсирована ее новая версия под названием Yucon, которую Microsoft планирует выпустить в самое ближайшее время. Однако изложение материала преследует, в первую очередь, задачу демонстрации концепций, общих для всех реляционных клиент-серверных СУБД, — ведь проблемы всюду одни и те же.

Книга построена достаточно просто. По порядку рассматривается процесс разработки клиент-серверного OLTP-приложения реляционной базы данных, разбитый на общепринятые этапы. Для каждого из этих этапов приводятся и анализируются рекомендации, выполнение которых, в конечном счете, может оказаться полезным для повышения производительности приложения. В качестве примера, проходящего через всю книгу, рассматривается простейшая база данных, предназначенная для классической задачи материального учета.

Глава 1 посвящена проектированию реляционной базы данных и построению ее информационных моделей. В этой главе достаточно подробно изложена общая теория реляционной модели данных, приведено и описано несколько ее наиболее распространенных графических нотаций, пригодных для всех фаз семантического моделирования, — от концептуальной до физической. Далее разбирается достаточно большое количество примеров, связанных с созданием логической модели реляционной базы данных. Для одного и того же описания предметной области обычно приводится несколько вариантов корректной модели. Эти варианты анализируются, объясняется, к каким преимуществам и проблемам на последующих стадиях разработки базы данных приведет тот или иной вариант проектного решения.

Для изображения моделей выбраны стандартизованные нотации IDEF1X (Integration DEFinition for Information Modeling, интегрированная среда определений для информационного моделирования) и IE (Information Engineering, информационная инженерия), предлагаемые CASE-средством (Computer-Aided Software Engineering, автоматизированное проектирование и создание программ) Erwin. Поэтому в *главе 1* книги даются минимальные сведения по работе с этой программой, которых хватает только для того, чтобы научиться изображать и понимать нужную модель. Конечно, для полноценной и эффективной работы с Erwin этого совершенно не достаточно — более подробно ознакомиться с возможностями этого популярного CASE-средства можно в работах [14] и [15] *(см. список литературы)*.

В *главе 2* рассматривается применение основного приема оптимизации проектного решения для OLTP-приложения реляционной базы данных, основанного на таком подходе, как нормализация.

Если материал *глав 1 и 2* в минимальной степени привязан собственно к SQL-серверу, то в *главе 3* рассматриваются вопросы, связанные с физической реализацией проектного решения в контексте выбранной целевой СУБД, которой в нашем случае является SQL-сервер. Здесь, кроме общих рекомендаций по реализации проектного решения в рамках файловой системы, собраны воедино рекомендации по настройкам SQL-сервера, влияющим на те или иные аспекты производительности.

Глава 4 посвящена такому объекту, который имеется в любой реляционной СУБД, как индексы. Индексам уделено особое внимание, потому что именно они являются основным средством, правильное использование которого позволяет существенно повысить быстродействие выполнения главных для базы данных реляционных операций. Описывается механизм физической реализации индексов разных типов и суть основанной на них оптимизации запросов. Рассматриваются команды создания индексов для SQL-сервера, разбираются параметры этих команд, объясняется, в какой ситуации какие значения этих параметров наиболее предпочтительны.

Глава 5 посвящена собственно программированию на SQL-сервере. В связи с этим заметим, что данная книга не является учебным пособием по SQL,-серверу в том смысле, что в ней не описывается базовый синтаксис языка T-SQL (Transact-SQL), являющегося языком программирования для MS SQL-сервера. Для того чтобы в полной мере разобраться в материале *глав 5 и 6*, желательно знание языка T-SQL, для чего можно порекомендовать работы [16], [17] и [18].

Основное внимание в *главе 5* уделяется оптимизации выполнения запросов на выборку данных, т. е. выполнению такого стандартного оператора языка SQL, как `SELECT`. При этом широко используются и анализируются планы выполнения запросов, разбираются и сравниваются по быстродействию альтернативные варианты решения одной и той же типичной задачи. Изложение начинается с простых запросов и заканчивается достаточно сложными SQL-конструкциями.

Далее в *главе 5* подробно рассматривается ряд полезных и интересных примеров практического программирования на T-SQL. Например, приводятся листинги программ автогенерации данных для базовых таблиц, включая способы заполнения данными столбца таблицы, содержащего рисунки.

Глава 6 посвящена вопросам оптимального проектирования транзакций. Сначала приведены сведения по общетеоретическим аспектам этого вопроса. Да-

лее подробно разбираются средства управления транзакциями собственно на SQL-сервере.

Книга ориентирована на бизнес-аналитиков, программистов приложений и администраторов, работающих с клиент-серверными технологиями реляционных баз данных, а также на преподавателей и студентов по соответствующим специальностям. При этом для чтения глав по проектированию баз данных какой-либо специальной подготовки не требуется. Однако для понимания идеологии оптимизации запросов необходимо знание языка SQL, хотя бы в объеме стандарта SQL-92.

Описание компакт-диска

Описание компакт-диска, прилагаемого к книге, приведено в табл. 1.

Таблица 1. *Описание компакт-диска*

Папки	Описание	Комментарии
\БД	Базы данных в формате MS SQL сервер 2000	
	MAT_УЧЕТ – БД для задачи материального учета	MAT_УЧЕТ_Data.MDF
	MAT_UCHET – БД для примеров	mat_uchet_Data.MDF
\PART1	Модели (проектные решения) для соответствующих рисунков главы 1 в виде файлов в формате Erwin 3.5.2 (*.er1). Листинги программ на T-SQL – текстовые файлы	mod_ris1_*.er1 Номер (номера), упомянутые после "1_" соответствуют номерам рисунков в тексте книги
\PART2… \PART6	Листинги программ на T-SQL – текстовые файлы	Номер, указанный в названии файла, соответствует номеру листинга в тексте книги. В начале каждого файла с листингом указано, в контексте какой БД нужно данную программу запускать. Если этой информации нет, то данные для соответствующей программы нужно задать согласно пояснениям в тексте книги
\PART5 \ГРАФИКА	Папка с графическими файлами	

ГЛАВА 1

Моделирование реляционных данных

База данных в самом общем смысле — это унифицированная совокупность данных, совместно используемых широким кругом пользователей, динамически изменяющаяся в процессе своего функционирования.

Если унифицированным форматом представления данных являются таблицы, то речь идет о реляционной базе данных.

1.1. Общая характеристика реляционной модели данных

Наиболее распространенная трактовка *реляционной модели данных* на сегодняшний день принадлежит К. Дейту, согласно которой эта модель состоит из трех частей.

- *Структурная часть* описывает, какие объекты рассматриваются реляционной моделью. Постулируется, что реляционной базой данных называется база данных, состоящая из набора отношений. Схемой реляционной базы данных называется набор заголовков отношений, входящих в базу данных.
- *Целостная часть* описывает ограничения специального вида, которые должны выполняться для любых отношений в любых реляционных базах данных. Это *целостность сущностей* и *целостность внешних ключей*.
- *Манипуляционная часть* описывает два эквивалентных способа манипулирования реляционными данными — реляционную алгебру и реляционное исчисление. С практической точки зрения, важным является вывод о реляционной полноте структурного языка запросов SQL в том или ином стан-

дарте, который и реализует манипуляционную часть реляционной модели в реальных СУБД.

1.1.1. Структурная часть реляционной базы данных

Основы реляционной модели данных были впервые изложены в статье Е. Кодда в 1970 г., которая послужила стимулом для дальнейшего развития реляционной модели данных. Сам термин *реляционное представление данных*, введенный Коддом, происходит от латинского RELATIO, что означает "отношение" или "таблица". Таблица является наиболее привычным и удобным вместилищем для хранения информации, имеет жестко оговоренное количество поименованных и упорядоченных столбцов (структуру), и может неограниченно расти по количеству строк.

Теоретической основой реляционных баз данных являются такие математические дисциплины, как теория множеств и реляционная алгебра или алгебра отношений. Фундаментальным понятием реляционной модели данных является теоретико-множественное определение понятия *отношения*, которое формулируется следующим образом.

Пусть даны N множеств $D1$, $D2$, ..., DN (не обязательно различных), называемых *доменами* отношения R. Тогда R есть отношение над этими множествами, если само множество R есть множество упорядоченных n-кортежей вида $<d1, d2, ..., dn>$, где $d1$ — элемент из $D1$, $d2$ — элемент из $D2$, ..., dn — элемент из DN.

Отношение R, определенное на множестве доменов $D1$, $D2$, ..., DN, имеет две части: заголовок и тело.

Заголовок отношения содержит фиксированное количество названий атрибутов (столбцов) отношения. Имена атрибутов должны быть уникальны в пределах отношения. Часто имена атрибутов отношения совпадают с именами соответствующих доменов. Заголовок статичен, он не меняется во время работы с базой данных. Если в отношении изменены, добавлены или удалены атрибуты, то в результате получим уже другое отношение (пусть даже с прежним именем).

Тело отношения представляет собой набор кортежей, т. е. подмножество декартового произведения доменов, что собственно и является отношением в математическом смысле слова. Тело отношения может модифицироваться во время работы с базой данных — кортежи могут изменяться, добавляться и удаляться.

Понятие отношения фактически лежит в основе теории реляционных баз данных. В практической реализации любой реляционной СУБД отношения являются математическим аналогом *таблиц*. В табл. 1.1 приведены термины, которые употребляются как синонимы.

Таблица 1.1. *Сравнение терминологии*

Реляционный термин	**Соответствующий "табличный" термин, связанный с программной реализацией СУБД**
Отношение	Таблица
Заголовок отношения	Заголовок таблицы
Тело отношения	Тело таблицы
Атрибут отношения	Наименование столбца (поля) таблицы
Кортеж отношения	Строка (запись) таблицы
Степень отношения	Количество столбцов таблицы
Мощность (кардинальность) отношения	Количество строк таблицы
Домен	Базовый или пользовательский тип данных

Свойства отношений непосредственно следуют из приведенного выше определения. Принципиальными здесь являются следующие моменты.

1. Все элементы отношения есть однотипные кортежи, что позволяет считать их аналогами строк в простой таблице, т. е. в такой таблице, в которой все строки состоят из одинакового числа ячеек и в соответствующих ячейках содержатся одинаковые типы данных.

2. За исключением крайнего случая отношение включает в себя не все возможные кортежи из декартового произведения доменов, на которых оно определено. Это значит, что для каждого отношения имеется критерий, позволяющий определить, какие кортежи входят в отношение, а какие нет. Этот критерий определяет смысл, или семантику, отношения. Он является логическим выражением и называется *предикатом отношения R*.

3. В отношении нет одинаковых кортежей, и кортежи не упорядочены сверху вниз, т. к. само отношение есть множество кортежей, и, следовательно, неупорядочено. Кроме того, как всякое множество отношение не может содержать неразличимые элементы. Поэтому одно и то же отношение может быть *изображено* разными таблицами, в которых *строки идут в различном порядке*.

1.1.2. Целостная часть реляционной базы данных

Домены

В реляционной теории принято считать, что все значения атрибутов отношения атомарны. Это следует из трактовки понятия домена. Домен можно рассматривать как подмножество значений некоторого типа данных, имеющих определенный смысл. Реляционная модель требует, чтобы типы используемых данных были простыми (скалярными), т. е. не обладающими внутренней структурой.

Конечно, понятие атомарности является весьма относительным. Так, строковый тип данных можно рассматривать как одномерный массив символов, а целый тип данных — как набор битов. Важно лишь то, что при переходе на такой низкий уровень теряется *семантика (смысл) данных*. Если строку, выражающую, например, фамилию сотрудника, разложить в массив символов, то при этом теряется смысл такой строки как единого целого.

Итак, *домен* — это семантическое понятие, которое несет определенную смысловую нагрузку. Домен имеет уникальное имя в пределах базы данных, определен на простом типе данных или на другом домене. Возможно наличие некоторого логического условия, позволяющего описать подмножество данных, допустимых для данного домена. Например, домен D, имеющий смысл "возраст сотрудника", можно описать как следующее подмножество множества натуральных чисел: ($D = n \in N$: $n \geq 18$ и $n \leq 60$).

Собственно для реляционной модели данных тип используемых данных не важен. Требование того, чтобы тип данных был *простым*, нужно понимать так, что *в реляционных операциях не должна учитываться внутренняя структура данных*. Конечно, должны быть описаны действия, которые можно производить с данными как с единым целым, например, данные числового типа можно складывать, для строк разрешается операция конкатенации и т. д. С этой точки зрения, если рассматривать массив как единое целое и не использовать поэлементных операций, то его можно считать простым типом данных. Более того, допустимо создать свой, сколь угодно сложный тип данных, описать возможные действия с ним, и если в операциях не требуется знание внутренней структуры данных, то такой тип данных также будет простым с точки зрения реляционной теории.

Основное назначение доменов состоит в том, что они *ограничивают сравнения*. Некорректно, с логической точки зрения, сравнивать значения из различных доменов, даже если они имеют одинаковый тип. Синтаксически правильный запрос "выдать список всех товаров, у которых учетная цена больше имеющегося на складе количества", не соответствует смыслу понятий "количество" и "цена". Таким образом, понятие домена помогает правильно *моделировать* предметную область.

Перейдем к введению понятий, определяющих целостную часть реляционной базы данных. Основными компонентами целостности являются, как уже было сказано, два ограничения, которые должны выполняться в *любой* реляционной базе данных: целостность сущностей и целостность внешних ключей.

Целостность сущностей

Под *сущностью* понимается некоторый объект, представляющий интерес для описания предметной области при создании модели базы данных. Понятие сущности возникает из методологии семантического моделирования данных, которая подробно рассматривается в *главе 1*. Финальной целью этого процесса является представление сущностей в виде отношений, поэтому для введения понятий, связанных с описанием целостной части реляционной базы данных, достаточно понимания сущности как синонима понятия отношения или таблицы.

Вводятся следующие понятия.

Ключ (возможный ключ, потенциальный ключ) отношения — это минимальный набор атрибутов, который может быть использован для однозначной идентификации любого кортежа отношения. Ключи отношений бывают *атомарными* и *составными*. Атомарный ключ состоит из единственного атрибута. Составной ключ представляет собой набор атрибутов. Ключ не должен содержать лишних атрибутов. Это означает, что если любой атрибут исключить из ключевого набора, то оставшихся атрибутов будет уже недостаточно для идентификации кортежа.

Любое отношение имеет по крайней мере один потенциальный ключ. Действительно, если никакой атрибут или группа атрибутов не являются потенциальным ключом, то в силу уникальности кортежей все атрибуты вместе образуют потенциальный ключ.

Отношение может иметь несколько потенциальных ключей. Традиционно один из них объявляется *первичным ключом* (PK, Primary Key), а остальные — *альтернативными*. Различия между первичными и альтернативными ключами могут быть важны в конкретной реализации реляционной СУБД. Далее этому вопросу будет уделено должное внимание.

Понятие потенциального ключа является *семантическим* и отражает введенную определенным образом трактовку понятий из конкретной предметной области. Например, рассмотрим отношение "Сотрудники", представленное табл. 1.2.

На первый взгляд может показаться, что в этом отношении имеются три потенциальных ключа — в каждой колонке таблицы содержатся уникальные данные. Однако среди сотрудников могут быть однофамильцы и сотрудники

с одинаковой зарплатой. Табельный же номер по сути своей уникален для каждого сотрудника, и именно *понимание семантики данных* привело к утверждению, что в данном отношении имеется только один потенциальный ключ — "Табельный номер".

Таблица 1.2. *Отношение "Сотрудники"*

Табельный номер	Фамилия	Зарплата
1002	Иванов	1000
1003	Петров	2000
1004	Сидоров	3000

Если представить отношение "Сотрудники" в виде табл. 1.2, не сообщив смысл наименований атрибутов, то окажется невозможным судить, может или не может в этом отношении появиться, например, кортеж (1002, Петров, 3000). Если бы такой кортеж появился, что не противоречит уникальности кортежей, то можно точно сказать, что *не является* альтернативным ключом — ни один из атрибутов по отдельности. Но нельзя сказать, что же *является* ключом отношения.

Потенциальные ключи служат единственным *средством адресации на уровне кортежей* в отношении. Только знание значения потенциального ключа кортежа позволяет точно указать этот кортеж.

С точки зрения семантического моделирования данных, потенциальные ключи служат *средством идентификации* объектов предметной области — экземпляров сущностей, данные о которых хранятся в отношении. Поскольку эти экземпляры должны быть различимы по определению, их идентификаторы не могут содержать неизвестные значения.

Обычно для ситуации наличия неизвестных или неполных данных используются типы данных, пополненные так называемым *`NULL`-значением*.

`NULL`-значение — это, собственно, не значение, а некий маркер, показывающий, что значение неизвестно. За кажущейся естественностью такого подхода скрываются весьма неоднозначные проблемы, наиболее явной из которых является необходимость использования трехзначной логики при оперировании `NULL`-значениями.

Проблема использования `NULL`-значения в теории реляционных баз данных окончательно не решена. Практически все реализации современных реляционных СУБД позволяют использовать `NULL`-значения, несмотря на их недостаточную теоретическую обоснованность. Безусловно, следует избегать появления `NULL`-значений, обусловленных некорректным проектированием базы

данных, но в реальной ситуации, когда какие-то данные об экземпляре сущности просто неизвестны, без `NULL`-значений не обойтись.

Правило целостности сущностей. Каждая сущность, реализованная тем или иным отношением, должна иметь по крайней мере один потенциальный ключ, входящие в состав которого атрибуты не могут принимать `NULL`-значений. Этот потенциальный ключ лучше всего объявлять первичным ключом таблицы, соответствующей данной сущности.

Следует отметить, что большинство СУБД вполне позволяют создавать таблицы и без первичных ключей. Однако нарушение правила целостности сущностей на практике сразу дает о себе знать. Например, для СУБД MS SQL-сервер станет невозможным доступ к данным по технологии OLE DB Provider, что сразу отчетливо видно при работе с приложением SQL Server Enterprise Manager — ни вставить, ни удалить записи визуальными средствами в таблице не удастся.

Целостность внешних ключей

Различные объекты предметной области, информация о которых хранится в базе данных, всегда взаимосвязаны. Наиболее типичный способ подобной связи между отношениями описывается ограничением внешнего ключа (FK, Foreign Key). Дадим точное определение.

Пусть дано отношение R. Подмножество атрибутов FK отношения R будем называть *внешним ключом*, если существует отношение S (S может совпадать с R) с потенциальным ключом K, и каждое значение FK в отношении R всегда совпадает со значением K для некоторого кортежа из S либо является `NULL`-значением.

Отношение S называется *родительским отношением*, а отношение R — *дочерним отношением*. Атрибут K часто называют вторичным ключом, а пару атрибутов K и FK — полями связи отношений S и R.

Другими словами, для каждого отличного от `NULL`-значения поля, входящего во внешний ключ дочернего отношения, существует одно и только одно значение поля связи в родительском отношении. На уровне реализации СУБД поле связи в родительском отношении — это чаще всего его первичный ключ, а если нет, то оно должно быть объявлено альтернативным ключом отношения.

Внешний ключ, так же как и потенциальный, может быть простым и составным.

Внешний ключ должен быть определен на тех же доменах, что и соответствующий первичный ключ родительского отношения.

Внешний ключ, как правило, *не обладает свойством уникальности*. Так и должно быть, поскольку в дочернем отношении может быть несколько кортежей, ссылающихся на один и тот же кортеж родительского отношения, что, собственно, и дает тип связи между отношениями *"один-ко-многим"*. Это стандартный тип связи с сохранением ссылочной целостности. Если внешний ключ все-таки обладает свойством уникальности, то связь между отношениями имеет тип "*один-к-одному*".

Хотя каждое значение внешнего ключа обязано совпадать со значениями потенциального ключа в некотором кортеже родительского отношения, обратное, вообще говоря, неверно. В поле связи родительской таблицы могут присутствовать значения, на которые не ссылается ни одно из значений внешнего ключа.

`NULL`-значения для атрибутов внешнего ключа допустимы только в том случае, когда атрибуты внешнего ключа не входят в состав никакого потенциального ключа.

Поскольку внешние ключи фактически служат ссылками на кортежи в другом (или в том же самом) отношении, то эти ссылки не должны указывать на несуществующие объекты.

Сформулированные выше соображения определяют *правило целостности внешних ключей* или *ссылочной целостности* реляционной базы данных: внешние ключи не должны быть несогласованными, т. е. для каждого значения внешнего ключа должно существовать соответствующее значение в поле связи в родительском отношении.

Примером ограничения внешнего ключа может служить следующая ситуация. Пусть имеются студенты, объединенные в группы. Каждый студент может обучаться только в одной группе. Студенты идентифицируются студенческим билетом (`Студбилет`), а группы студентов идентифицируются специально введенными уникальными номерами (`№_группы`). Тогда между множеством значений атрибута `Студбилет` и множеством значений атрибута `№_группы` имеет место связь внешнего ключа, причем атрибут `№_группы` является внешним ключом для атрибута `Студбилет`.

Замечание

Существуют и другие типы ограничений целостности в реляционной базе данных, которые реализуются разными способами. Но на современном уровне конкретная СУБД считается в полном смысле реляционной, если в ней есть стандартные средства автоматической реализации ограничений целостности сущностей и целостности внешних ключей.

Механизмы реализации ограничений целостности реляционной базы данных будут подробно рассматриваться на протяжении всей книги.

1.1.3. Манипуляционная часть реляционной базы данных

Манипуляционная часть реляционной базы данных описывает средства, с помощью которых, во-первых, из данных, хранящихся в базе данных, можно получать какие-либо выборки или сводные результаты, а во-вторых, изменять сами эти данные или их структуру.

Манипуляционной частью современных промышленных СУБД является специальный язык запросов, называемый SQL, который стал фактическим стандартом доступа к реляционным базам данных. Операторы этого языка по своей сути представляют собой требование к базе данных на выполнение каких-то действий. Язык SQL является реляционно полным. Это означает, что любой оператор реляционной алгебры может быть выражен подходящим оператором языка SQL.

Все СУБД, претендующие на название "реляционные", реализуют тот или иной диалект SQL. Многие нереляционные системы также имеют средства доступа к реляционным данным. Целью стандартизации является возможность переноса приложений между различными СУБД. Однако в настоящее время ни одна из систем не реализует стандарт SQL в полном объеме. Кроме того, во всех диалектах языка имеются возможности, не являющиеся стандартными. Таким образом, можно сказать, что каждый диалект — это надмножество некоторого подмножества стандарта SQL. Подобная ситуация затрудняет перенос приложений между разными СУБД, но скорее всего добиться полной стандартизации невозможно, т. к. динамика развития этой сферы такова, что организации, ведающие установлением и распространением стандартов, просто не успевают за производителями программных новинок.

Диалектом языка SQL, реализованным в СУБД SQL-сервер, является так называемый язык T-SQL, который сочетает в себе операторы языка SQL и традиционные средства структурного программирования. Изучение синтаксиса языка T-SQL выходит за рамки данной книги. Безусловно, его знание является обязательным для понимания материала, изложенного в *главе 5*, которая посвящена программированию на SQL-сервере. Однако начальные главы, посвященные приемам моделирования реляционной базы данных, жестко не связаны с целевой СУБД.

1.2. Технология клиент-сервер

В предыдущем разделе рассмотрены особенности, которые связаны с понятием унифицированного формата данных в определении базы данных. Показано, что принятие формата данных, основанного на таблицах или отношениях,

привело к реляционной модели данных со всеми ее характерными особенностями.

Однако в этом определении имеются еще составные части, которые обязательно необходимо реализовать. Речь идет о фразе "совокупность данных ... совместно используемых широким кругом пользователей, динамически изменяющаяся в процессе своего функционирования", т. е. о том, что информация из базы данных должна быть *одновременно* и *без задержки* предоставлена широкому кругу лиц в необходимой для них форме для анализа или модификации таким образом, чтобы по возможности они не ощущали присутствия друг друга.

Технология клиент-сервер означает такой способ взаимодействия программных компонентов, при котором они образуют единую систему. Как видно из названия, существует некоторый *клиентский процесс*, требующий определенных ресурсов, а также серверный процесс, который эти ресурсы предоставляет. При этом совсем необязательно, чтобы эти процессы выполнялись на одном и том же компьютере. Наоборот, на практике принято размещать серверный процесс на одном узле локальной сети, а клиентские — на других узлах.

В контексте базы данных клиент управляет пользовательским интерфейсом и логикой приложения, действуя как интеллектуальная рабочая станция, на которой выполняются приложения баз данных. Клиент принимает от пользователя запрос, проверяет его синтаксис и генерирует SQL-запрос к серверу на предоставление тех или иных данных. Этот запрос передается на сервер по компьютерной сети. Сервер принимает и обрабатывает запрос к базе данных, затем в зависимости от типа запроса либо передает полученные данные обратно клиенту, либо выполняет модификацию содержимого базы данных. Клиент принимает переданные сервером данные, форматирует их и представляет пользователю.

Обработка сервером запроса включает в себя проверку полномочий клиента, обеспечение требований целостности, собственно выполнение запроса и передачу клиенту необходимых результатов. При этом сервер занимается такими проблемами, как поддержка параллельности работы многих клиентов, которая включает в себя, в первую очередь, согласование данных, одновременно предоставляемых и изменяемых разными клиентами. Подобная архитектура обладает следующими преимуществами:

- обеспечивается более широкий доступ к существующим базам данных;
- повышается общая производительность системы. Поскольку клиенты и сервер находятся на разных компьютерах, их процессы способны выполнять приложения параллельно. При этом настройка производительности

сервера упрощается, если на компьютере, выделенном под сервер, выполняется только работа с базой данных;

- сокращается нагрузка на компьютерную сеть. Это происходит прежде всего за счет того, что в ответ на запрос клиента сервер возвращает ему готовые результаты запроса, а не все данные, необходимые для их получения;
- считается, что существует тенденция к снижению стоимости аппаратного обеспечения, т. к. наиболее ресурсоемкими операциями для любой СУБД является выполнение реляционных запросов. В случае клиент-серверной платформы все они выполняются на сервере, следовательно, высокие аппаратные требования предъявляются теперь в первую очередь к компьютеру, выделенному под сервер. Кроме того, более скромные требования теперь предъявляются и к аппаратным средствам поддержки сети;
- появляется возможность использования специализированного аппаратного обеспечения для сервера, которое может быть сконструировано именно для работы на сервере баз данных. Это очень существенный аспект повышения общей производительности работы системы;
- повышается уровень непротиворечивости данных, поскольку все проверки выполняются в одном месте.

Клиент-серверные платформы приходят на смену так называемым платформам типа *файл-сервер*, уверенно их вытесняя. В подобной ситуации, разумеется, интересными представляются данные о том, насколько более производительными являются клиент-серверные технологии по сравнению с файл-серверными, в чем заключаются их дополнительные преимущества и какую же цену за все это нужно платить.

Рассмотрим вкратце, как работает типичное файл-серверное приложение. Основной принцип его реализации состоит в том, что обработка данных ведется рабочей станцией, а сервер служит просто как дополнительное, доступное всем пользователям дисковое устройство. Это означает, что при выполнении задачи, например при сборке отчета, *вся* база данных или значительная ее часть прокачивается по сети на рабочую станцию и там обрабатывается процессором рабочей станции. Быстродействие такой системы зависит от быстродействия диска сервера, скорости передачи данных по сети, мощности процессора рабочей станции, объема ее оперативного запоминающего устройства и некоторых других факторов. Центральный процессор сервера играет второстепенную роль и должен просто обеспечивать передачу потока данных с сетевого канала на диск и обратно, по возможности не замедляя этот процесс. Главным при таком подходе является то, что практически вся база данных перегоняется по сети на рабочую станцию для последующей обработки. Поэтому если несколько станций одновременно выполняют сборку

отчетов, то на все эти рабочие станции закачивается база данных, и естественно система в целом существенно замедляет работу из-за перегрузки сети.

Когда, с точки зрения передачи данных по сети, выполняются менее накладные операции, например ввод нового документа, объем перекачки данных будет значительно меньше даже с учетом того, что в реальности ввод документа, как правило, сопровождается поиском клиента в справочнике, вычислением его задолженности и т. п. Эти моменты увеличивают объем передачи количества информации с диска сервера на рабочую станцию, но в итоге нагрузка на сеть обычно бывает меньшей, чем при составлении отчетов.

Не следует также забывать о необходимости синхронизации доступа рабочих станций к данным. Поскольку вся обработка ведется на уровне рабочих станций, а файл-сервер просто играет роль разделяемого дискового устройства, задачи синхронизации решаются в таких системах с помощью организации различных файлов блокировок на диске файл-сервера. В эти файлы каждая рабочая станция записывает информацию о данных, которые она модифицирует в данный момент, а при попытке считать данные проверяет, не заняты ли эти данные другой рабочей станцией. Несмотря на ряд недостатков, например таких, как висячие блокировки при аварийном выключении рабочих станций или замедление работы всей системы при модификации большого числа записей, способ управления блокировками, основанный на концепции файл-сервер, вполне работоспособен.

Легко сообразить, что производительность файл-серверной системы напрямую связана с объемом обрабатываемой и, следовательно, передаваемой по сети базы данных с учетом трафика всех одновременных соединений. В ситуации, когда сеть перестает справляться и "торможение" ощутимо дает о себе знать, пользователи системы идут на различные хитрости: закрывают старую базу и открывают новую каждый квартал, пытаются удалить старые данные и т. п. Однако любой бухгалтер может сказать, что данные нужны ему не за квартал, а как минимум за год и предпочтительно в динамике, а не в виде отдельных кусков. Да и долги клиентов иногда тянутся годами.

Временным решением проблемы в такой ситуации может стать увеличение пропускной способности сети за счет установки более мощного и дорогостоящего аппаратного обеспечения. Однако стратегически это не спасет ситуацию. Зачем увеличивать пропускную способность сети, если при росте базы данных все равно наступит момент, когда жесткий диск сервера достигнет предела своей производительности, т. к. все данные, которые уже перегрузили сеть, должны быть прочитаны именно с этого диска? На первый взгляд кажется, что последнее замечание не очень обосновано, — ведь и в клиент-серверных системах данные тоже читаются с того же диска. Однако объем физического чтения в последнем случае существенно уменьшается

благодаря наличию мощного механизма кэширования данных, который является неотъемлемой частью любой клиент-серверной системы.

Не следует забывать и еще об одной важной детали. Это архивирование, которое нужно делать каждый день. И при этом во время архивирования ни один из пользователей с базой данных работать не может.

Теперь рассмотрим клиент-серверную систему. С одной стороны, сами серверы, предназначенные для нее, должны располагать значительно большими ресурсами. С другой стороны, наличие при этом фантастической пропускной способности сети совсем не обязательно. Все дело в том, что в случае клиент-серверной технологии при работе с сервером рабочая станция не качает всю базу данных к себе по сети, а просто передает достаточно компактный запрос на сервер, где запрос и выполняет, после чего результаты запроса передаются обратно на рабочую станцию. Таким образом, трафик по сети значительно снижается. Конечно, если при разработке клиентской части программного обеспечения будут созданы запросы, результатами выполнения которых является вся база данных или значительная ее часть, то основное преимущество клиент-серверной технологии будет утрачено. Но этот недочет будет лежать на совести разработчиков прикладных программ.

Итак, первое, что необходимо понять и запомнить: основным преимуществом клиент-серверных систем не является ускорение выполнения выборок данных. Несколько лет назад журнал PC Magazine проводил сравнительный анализ (в том числе и по быстродействию) СУБД, построенных на основе обычных файл-серверов и с использованием клиент-серверных систем. Естественно, условия испытаний по возможности были нивелированы, т. е. применялись одинаковые объемы баз данных, одинаковые их структуры, один и тот же компьютер в качестве сервера, одинаковое количество рабочих станций и т. д. Во всех случаях средняя скорость выполнения запросов в клиент-серверных системах была ниже, чем у файл-серверных систем. Сейчас об этом эффекте можно прочесть в любой серьезной книге по SQL. Тестировщики не были удивлены полученным эффектом, поскольку понимали причину такого поведения клиент-серверных систем при заданных условиях эксперимента. Ведь задачей эксперимента было сравнение быстродействия двух видов систем, и поэтому были выбраны условия, позволяющие произвести это сравнение. В частности, для тестов использовались базы данных объемом 1,5—2 Гбайт. Ведь если проводить испытания, используя базы данных на порядок большего размера, то клиент-серверные варианты не с чем было бы сравнивать, поскольку обычная файл-серверная система на таких объемах информации просто не имеет возможности функционировать. Вот в этом-то и состоит основное отличие и достоинство клиент-серверных систем: они будут работать с вполне приемлемой скоростью с базами данных такого объема и с таким числом одновременных соединений, с которыми файл-

серверная система работать не сможет в том смысле, что ее функциональность, в том числе и быстродействие, станут неприемлемыми для коммерческих приложений.

К дополнительным преимуществам клиент-серверных систем можно отнести следующие моменты. Системы этого типа имеют встроенный механизм работы с транзакциями, в том числе и их отката. В файл-серверных системах также существует механизм работы с транзакциями, однако способ их реализации принципиально отличается. Это отличие заключается в том, что механизм управления транзакциями представляет собой не что иное, как просто блокировку всей базы данных до завершения выполнения критических по времени операций одной из рабочих станций. Откат возможен только при сохранении работоспособности рабочей станции, инициировавшей транзакцию.

В клиент-серверной системе этот механизм значительно более сложен. Он позволяет получить "слепок" базы данных на момент начала транзакции без блокировки базы данных. И слепков таких может быть много: для каждой рабочей станции свой. В случае "зависания" рабочей станции, открывшей транзакцию, эта транзакция может быть просто откачена, а база данных будет восстановлена в том виде, в каком она была до инициации транзакции. Откат осуществляется либо по запросу рабочей станции (при сохранении ее работоспособности), либо при перезагрузке рабочей станции, либо администратором клиент-серверной системы. Таким образом, выход из строя рабочей станции не столь опасен для целостности базы данных. Кроме того, клиент-серверная система ведет так называемый журнал транзакций. По сути база данных хранится в виде ее начального содержимого и модификаций, записанных в журнал транзакций. Такой способ хранения позволяет производить архивирование базы данных во время работы всей системы: просто состояние базы данных фиксируется на момент начала архивирования, отсекаются незавершенные транзакции, а основная база и часть журнала транзакций, содержащая завершенные транзакции, записываются в архив. Процесс архивирования легко поддается автоматизации, при которой присутствие оператора необязательно, т. к. клиент-серверные платформы обычно для этого имеют достаточно простые и удобные встроенные средства.

1.3. Общий обзор методологии проектирования информационных систем

Информация должна рассматриваться не только как некая принадлежащая предприятию ценность, но и как исходная точка для построения информационной системы, обслуживающей предприятие. *Информационное проектирование* становится наиболее популярной методологией, пронизывающей все

стадии жизненного цикла системы. Главная посылка, на которой строится данная стратегия, состоит в том, что целостность информационных систем определяется степенью охвата информационных элементов объекта соответствующей логической моделью. Информационное проектирование предполагает прежде всего глубокое исследование всех информационных систем, преследующее своей целью поиск ответа на вопрос: каким образом информация используется и разделяется системами. Программные структуры выступают по отношению к информационной модели "надстройкой", определяющей общую информационную инфраструктуру для создания соответствующих систем на предприятии. Таким образом, процедуры логически вытекают из данных, а качество информационных систем определяется качеством построения информационной модели. Информационное обследование организации — трудоемкая и продолжительная работа. Только опытным аналитикам, вооруженным мощным инструментарием и технологией, под силу обеспечить точность и целостность описания информации.

Начальные этапы жизненного цикла приложения базы данных, т. е. собственно ее проектирование, требуют применения адекватных средств автоматизированного создания программ, которые принято называть *CASE-средствами*. В самом широком смысле термин *CASE-инструмент* применим к любым средствам автоматизированного проектирования и создания программ. В частности, CASE-инструменты незаменимы при проектировании реляционной базы данных, т. к. они позволяют существенно повысить производительность и качество подобной работы. Этот позитивный эффект достигается за счет приведенных далее преимуществ.

- *Стандартизация*. CASE-инструменты способствуют расширению использования стандартов на всех этапах жизненного цикла приложения, начиная со стадии его проектирования и заканчивая повседневным обслуживанием в процессе эксплуатации.
- *Интеграция*. CASE-инструменты позволяют сохранять всю генерируемую информацию в специальном хранилище, или *словаре данных*. Таким образом, появляется возможность хранить полный объем данных, собранных на всех этапах жизненного цикла приложения, компоновать их между собой, что является важной предпосылкой успешной интеграции системы в целом.
- *Графические возможности*. Как правило, любые структурированные технологии широко используют диаграммы, которые достаточно трудно создавать и поддерживать вручную. Использование CASE-инструментов позволяет существенно упростить этот процесс, предоставляя возможность создания более корректной и актуальной документации.
- *Непротиворечивость*. Поскольку вся информация в словаре данных взаимосвязана, CASE-инструменты способны обеспечить автоматическую проверку этой информации на непротиворечивость.

- *Инжиниринг*, *реинжиниринг* и *синхронизация*. Современные CASE-инструменты позволяют автоматически преобразовывать фрагменты спецификации проекта в программный код, выполняя тем самым инжиниринг модели, и наоборот, создавать разнообразные модели, используя готовое приложение в качестве входных данных, осуществляя процесс, называемый *реинжинирингом*. Кроме того, если в функционирующее приложение вносятся изменения, то с помощью CASE-инструментов можно выполнять автоматическую синхронизацию словаря данных в соответствии со сделанными изменениями.

Методологии или технологии и инструментальные средства проектирования (CASE-средства) составляют основу проекта любой информационной системы. Методология реализуется через конкретные технологии и поддерживающие их стандарты, методики и инструментальные средства, которые обеспечивают реализацию всех стадий жизненного цикла информационной системы.

Технология проектирования определяется как совокупность трех составляющих:

- *пошаговой процедуры*, определяющей последовательность технологических операций проектирования;
- *критериев и правил*, используемых для оценки результатов выполнения технологических операций;
- *нотаций* (графических и текстовых средств), используемых для описания проектируемой системы.

Моделирование данных является одной из важнейших составляющих проектирования информационной системы в целом. Цель моделирования данных состоит в обеспечении разработчика информационной системы концептуальной схемой базы данных в форме одной модели или нескольких локальных моделей, которые относительно легко могут быть отображены в любую СУБД. Наиболее распространенным средством моделирования данных в случае, когда выбрана реляционная СУБД, является так называемое *семантическое (инфологическое) моделирование*, которое представляет собой моделирование структуры данных основываясь на смысле этих данных. Другим общепринятым названием этого метода является термин *"сущность-связь"*. В качестве инструмента семантического моделирования используются различные варианты *диаграмм "сущность-связь"* (ER, Entity-Relationship или ERD). С их помощью определяются важные для предметной области объекты (сущности), их свойства (атрибуты) и отношения друг с другом (связи).

1.4. Сбор информации для моделирования данных

Перед тем как разрабатывать диаграмму "сущность-связь", необходимо собрать информацию о возможных сущностях предметной области и их атрибутах. Для этого существует множество способов, как формальных, так и неформальных. Укажем на некоторые из них.

Простейший неформальный способ — взять описание предметной области (возможно, полученное в процессе проведения интервью с заказчиком) и выделить в нем существительные и глаголы. Существительные будут кандидатами на роль сущностей, глаголы — кандидатами на роль связей. Естественно, приведенное правило является очень приблизительным, и каждая полученная таким образом сущность-кандидат должна проверяться коллективом аналитиков, возможно, даже голосованием.

Можно получить список сущностей на основе формального описания бизнес-процессов системы, так называемых *диаграмм бизнес-процессов* (business process diagram) или *диаграмм потоков данных* (DFD, Data Flow Diagram). *Подробную информацию см. в [14] и [15] списка литературы*. Два этих вида диаграмм близки по смыслу.

Построение диаграмм бизнес-процессов на начальном этапе проектирования должно служить одной основной цели: выделению сущностей в предметной области. Часто возникает желание смоделировать бизнес-процессы в организации до построения модели данных, т. е. разработать функциональную спецификацию системы, а потом уже определить данные, которых требует каждая функция. Это выглядит естественным, но может привести и к проблемам.

Как уже говорилось, желательна приоритетность информационного проектирования, т. е. построение модели данных до функциональной спецификации. Такой подход дает следующие преимущества:

- многократное использование данных для функций, которые не предвидятся в момент построения модели, и применение ранее выделенных функций в качестве тестовых для модели данных;
- установление устойчивых соглашений об именах и определениях для данных. В противном случае в функциональной спецификации одна функция могла бы быть названа "получить сведения о клиенте", а вторая — "обслужить заказчика", и возникнет вопрос, чем же заказчик отличается от клиента;
- автоматическое получение значительной части функциональной спецификации, т. к. просто посмотрев на построенную модель данных, можно предположить, какие функции необходимы для работы с этими данными;

- определение концепции, с помощью которой можно лучше понять и определить предметную область: что в ней происходит, какие вещи имеет смысл реализовывать, а какие нет, и т. д.

Естественно, что использование таких CASE-средств моделирования диаграмм бизнес-процессов, как BpWin, снимает ряд проблем (например с устойчивыми соглашениями об именах сущностей), но всех проблем эти средства все равно решить не могут. Поэтому одним из основных правил можно считать следующее. Не нужно увлекаться процессом построения диаграмм ради самого процесса. Всегда следует помнить о том, что конечная цель состоит в сборе информации о возможных сущностях, а также в намерении чуть лучше понять систему. При этом можно попытаться в какой-то мере дать ответ на вопросы, подобные следующему: *будет ли сущность или связь иметь смысл два года спустя для команды, которая будет работать с системой после нас?*

Информацию о сущностях можно получить из *диаграмм вариантов использования* (Use Case Diagrams), построенных для данной системы. Этот подход позволяет рассматривать систему с точки зрения различных сценариев (еще один перевод термина *use case*) ее использования различными категориями пользователей и других систем, взаимодействующих с ней. Варианты использования в значительной степени могут заменить диаграммы бизнес-процессов и потоков данных.

Для получения первичной информации также полезно использовать такие источники информации о сущностях, их атрибутах и связях, как:

- сложные структуры данных предметной области;
- бизнес-правила;
- каталог данных.

Под *сложными структурами данных предметной области* обычно понимают ненормализованные структуры данных, которые играют в ней ту или иную роль. К таким структурам, например, могут относиться понятия "заказ", "счет", "журнал посещаемости" и т. д. Какие-то из подобных структур сохранятся в модели неизменными, в то время как другие структуры подвергнутся декомпозиции.

Бизнес-правила (их еще называют *ограничениями*) описывают правила работы с данными, независимые от способа их представления в базе данных, которые связаны только с предметной областью. Примеры бизнес-правил:

- налог с продаж снимается с каждого проданного товара и равен 30 %;
- работа преподавателей допускается не более чем на 1,5 ставки;
- на должность бухгалтера можно принять человека с высшим финансовым или экономическим образованием;

- нельзя выписывать накладную на отпуск товара, если его нет на складе в достаточном количестве.

Имеются разные способы реализации бизнес-правил. Они могут быть заданы при проектировании базы данных как ограничения на данные, могут храниться как программы (в виде триггеров или хранимых процедур). Конкретный способ реализации зависит от характера правила. Выделяют следующие типы бизнес-правил.

- *Определение*, которое оправдывает существование сущности или атрибута и описывает его возможные значения.
- *Факт*, т. е. такое бизнес-правило, которое описывает связь или отношение между двумя типами объектов, например, "на Новый год покупатель получает или подарок (сущность) или скидку (атрибут), но не то и другое вместе".
- *Правило вывода*, которое определяет, каким образом экземпляры одной сущности могут быть получены из экземпляров другой сущности.
- *Ограничение*, которое позволяет ограничивать множество экземпляров сущности или значений атрибута.

Словарь или *каталог данных* — это база метаданных, в которую заносятся все обнаруженные на этапе сбора информации сложные структуры данных, уже найденные сущности, атрибуты (возможно, без сущностей) и связи. При построении модели этот каталог исследуется в поисках сущностей, атрибутов и связей.

1.5. Реляционная модель: концептуальная, логическая и физическая стадии

Цель моделирования данных состоит в том, чтобы построить формальное представление данных, которое:

- скрывает неинтересные детали;
- заменяет значительные по объему компоненты компактными символами;
- подчеркивает важные факты;
- способствует пониманию системы в целом.

На разных этапах моделирования рассматривают концептуальную, логическую и физическую модели данных, объединенные общими принципами методологии "сущность-связь". Более точное определение модели каждого уровня дается в *главе 2*, а сейчас важно в первую очередь понять то обстоятельство, что каждая модель последующего уровня уточняет модель предыдущего уровня.

- *Концептуальная модель данных* отображает данные с точки зрения их смысла и в контексте пользователя.
- *Логическая модель данных* — это обобщенная реляционная схема базы данных, которая не зависит от способа ее физической реализации.
- *Физическая модель данных* — это проект базы данных для выбранного конкретного продукта СУБД и определенной конфигурации компьютера, на котором базу данных предполагается реализовать.

Существуют различные соображения по поводу того, что к какой модели относится. Все же в рамках данной книги будем придерживаться следующих соглашений: концептуальная модель скрывает технические детали для того, чтобы подчеркнуть наиболее важные сущности, с точки зрения предметной области и пользователя; способ изображения модели — диаграммы "сущность-связь"(ER-диаграммы) любого из имеющихся типов.

В рамках построения концептуальной модели обычно выполняются следующие действия:

- сбор информации о сущностях и связях;
- отображение сущностей на ER-диаграмме;
- определение всех потенциальных ключей сущностей;
- отображение связей на ER-диаграмме;
- выявление и отображение иерархических связей.

Атрибуты сущностей в рамках концептуальной модели не уточняются, связи не реализуются унифицированным способом. Допустимы связи типа "многие-ко-многим".

Логическая модель данных должна быть непротиворечивой, нормализованной и достаточно полной, реализующей и уточняющей полученную на первом этапе концептуальную модель. В рамках перехода от концептуальной модели к логической обычно выполняются такие действия:

- нормализация в рамках первой нормальной формы и обусловленное ею уточнение перечня ключей сущностей и атрибутов;
- уточнение полного перечня атрибутов сущностей;
- агрегация элементов данных в новые сущности, если это необходимо;
- уточнение и формирование полного перечня возможных ключей, разделение этих ключей на первичные и альтернативные;
- нормализация с учетом новых ключей;
- замена связей "многие-ко-многим" ассоциативными сущностями и определение всех атрибутов уточненного списка сущностей;

- замена иерархических связей структурами, основанными на ограничении внешнего ключа.

В логической модели могут использоваться нефизические (абстрактные) обозначения для доменов и подтипов, и не определяется конкретная целевая СУБД, т. е. логическая модель разрабатывается с учетом выбранной модели организации данных, но независимо от типа целевой СУБД и других физических методов реализации. Способ изображения логической модели — унифицированные графические нотации, принятые в стандартных CASE-средствах.

Для наших целей вполне подойдет достаточно простое, но вместе с тем универсальное средство моделирования реляционной базы данных — программа, называемая Erwin. Ее возможности и приемы работы с ней более подробно рассматриваются далее. Сейчас нам важно отметить, что данная программа поддерживает две наиболее распространенные нотации моделей реляционной базы данных, называемые IDEF1X- и IE-нотации, которые дальше будут подробно изучаться.

С точки зрения выбранного нами CASE-средства моделирования реляционной базы данных, называемого Erwin, и концептуальной, и логической модели данных соответствует один и тот же режим программы (опция **Logical**) с одинаковым интерфейсом и нотацией модели. Можно считать, что на уровне концептуальной модели изображаются сущности в режиме, когда их атрибуты отключены, при этом допустимы связи "многие-ко-многим" без применения опции **Resolve Many To Many**.

Физическая модель данных — это проект базы для определенной целевой СУБД, который должен содержать полную информацию о содержимом системного каталога базы данных. Способ изображения физической модели — унифицированные графические нотации, принятые в выбранном CASE-средстве. Программа Erwin позволяет получать физическую модель базы данных из логической автоматически при выборе конкретной СУБД и переходе в режим работы программы, называемый **Physical**.

1.6. Нотации реляционной модели для методологии "сущность-связь"

Под термином *методология "сущность-связь"* будем понимать взаимосвязанные приемы работы с моделями всех трех уровней. Цель этой работы состоит в создании так называемого *проектного решения*, которое представляет собой набор объектов системного каталога целевой СУБД, реализованный средствами реляционной модели в некоторой стандартизованной нотации и

подготовленный к передаче на целевую СУБД путем выполнения сгенерированного на основе проектного решения DDL-сценария (Data Define Language, язык определения данных) в синтаксисе выбранной целевой СУБД.

Первый вариант модели "сущность-связь" был предложен в середине 1970-х гг. Питером Пин-Шэн Ченом. В дальнейшем многими авторами были разработаны свои варианты подобных моделей и основанные на них нотации этих моделей. По сути, все варианты диаграмм "сущность-связь" исходят из одной идеи — рисунок всегда нагляднее текстового описания. Все диаграммы этого типа используют графическое изображение сущностей предметной области, их свойств (атрибутов) и связей между сущностями.

На современном уровне работа бизнес-аналитика, проектирующего модель данных для информационной системы, немыслима без использования CASE-средств. Обычно различные программные CASE-средства, реализующие одну и ту же нотацию реляционной модели, могут отличаться своими возможностями. Однако общим для CASE-инструментов проектирования реляционной базы данных является то обстоятельство, что все они содержат набор инструментов, необходимых для изображения модели реляционной базы данных на уровне модели логической.

Основными объектами *инфологических моделей реляционных данных* являются сущности, связи и атрибуты. Суть любой модели состоит в отображении того факта, что сущности вступают в связи между собой, а атрибуты принадлежат сущностям. Основной способ изображения таких моделей — графический. Существуют различные нотации, особенно для начальной стадии моделирования на уровне ER-диаграмм.

1.6.1. Нотация Баркера, IDEF1X- и IE-нотации

Сначала очень кратко описывается нотация ER-диаграмм, близкая к нотации Баркера. Каждая сущность в модели изображается в виде прямоугольника с наименованием (именем сущности). Внутри этого прямоугольника перечислены атрибуты сущности. Ключевые атрибуты выделяются подчеркиванием. Все вместе изображается примерно так, как показано на рис. 1.1, где представлена сущность с названием `СОТРУДНИК`, ключ которой имеет название `Табельный номер`, а атрибутами этой сущности являются `Фамилия`, `Имя`, `Отчество`, `Должность` и `Зарплата`.

Существуют разные нотации моделей, например, на рис. 1.1 приведено сравнение изображения сущностей в нотациях Баркера, IDEF1X и IE, которые очень похожи.

Возможен способ изображения ER-диаграмм, при котором атрибуты сущностей не указываются вовсе.

Рис. 1.1. Графическое изображение сущности в нотации Баркера и в нотациях IDEF1X и IE

Кроме того, в стандартной ER-нотации атрибуты могут представляться эллипсами, но это ведет к очень большой перегруженности диаграмм (рис. 1.2).

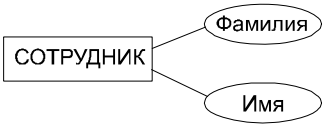


Рис. 1.2. Другой способ изображения атрибутов связи в нотации Баркера

В настоящее время более предпочтительным является простое перечисление атрибутов внутри прямоугольников (возможно с примером значения). Так принято в стандартизованных нотациях IDEF1X и IE, это согласуется с нотацией UML.

Связи между сущностями изображаются с помощью линии, вид которой определяет такие обязательные параметры связи, как *мощность связи*, или *степень связи* (cardinality), а также *класс принадлежности*, или *модальность* (optionality), которые еще называются *обязательностью связи*. В соответствии с этим выделяют связи *обязательные* и *необязательные* (другое название — *полные* и *частичные*). На рис. 1.3 приведено изображение связи между сущностями в нотации Баркера. Подобную связь из-за ее характерного вида иногда называют "вороньей лапкой" (crow's foot).

Рис. 1.3. Пример изображения связи в нотации Баркера

Вид линии определяет параметры связи. Объяснение смысла этих параметров приведено на рис. 1.4.

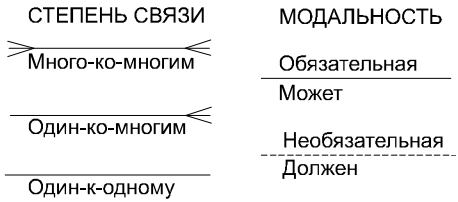


Рис. 1.4. Способы изображения параметров связи в нотации Баркера

Описанный графический синтаксис позволяет *однозначно* читать диаграммы, пользуясь следующей схемой построения фраз:

```
<Каждый экземпляр СУЩНОСТИ1> <МОДАЛЬНОСТЬ СВЯЗИ> <НАИМЕНОВАНИЕ СВЯЗИ>
<ТИП СВЯЗИ> <экземпляр СУЩНОСТИ2>.
```

Каждая связь может быть прочитана как слева направо, так и справа налево. Связь, изображенная на рис. 1.4, читается слева направо: "каждый сотрудник может иметь несколько детей" и справа налево: "каждый ребенок обязан принадлежать ровно одному сотруднику".

Нотация ER-диаграмм Баркера тесно связана со стандартизованной нотацией CASE-средств, называемой IE-нотацией. Сущности здесь изображаются, как и раньше, прямоугольниками, имя сущности записывается над этим прямоугольником. Ключевые атрибуты отделяются от неключевых с помощью горизонтальной линии, что также показано на рис. 1.1. Различие, в первую очередь, заключается в том, что в IE-нотации иначе изображается модальность связи, а именно обязательность обозначается вертикальной чертой, а необязательность — окружностью, причем *данный символ указывается с противоположной стороны от сущности, с точки зрения которой характеризуется данная связь*. Например, рассмотрим ситуацию, связанную с зачислением студентов в группы. Мощность этой связи будем считать равной "один-ко-многим", т. е. в одну группу зачисляется несколько студентов и студент не может быть зачислен более чем в одну группу. Разберем различные варианты модальности описанной связи и способы их изображения в IE-нотации.

Например, мы хотим показать, что не может быть студента вне группы, а, с другой стороны, группа без студентов имеет право на существование. В этом случае связь со стороны "один", т. е. с точки зрения группы, необязательная, а со стороны "много", т. е. с точки зрения студента, обязательная. Читается это так: *"в группе может учиться один или несколько студентов, при этом студент обязан быть зачисленным ровно в одну группу"*. Таким образом, если, с точки зрения студента, связь обязательная, то со стороны группы нужно задать вертикальную черту, а если со стороны группы связь необязательная, то со стороны студента нужно задать вместе с вертикальной чертой еще и окружность. Обозначается это так, как показано на рис. 1.5 и 1.6.

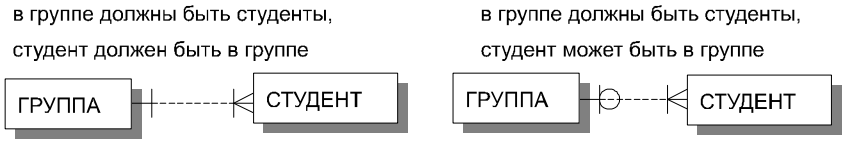


Рис. 1.5. Пример изображения модальности связи в IE-нотации с обязательной модальностью для сущности `ГРУППА`

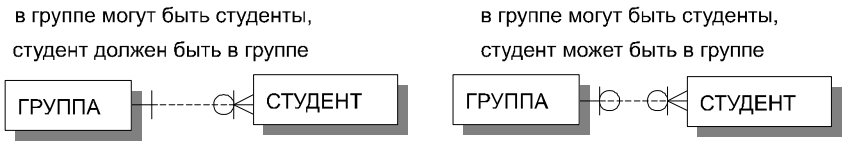


Рис. 1.6. Пример изображения модальности связи в IE-нотации с необязательной модальностью для сущности `ГРУППА`

Замечание

Прерывистая линия обозначает в нотации IE *неидентифицирующую связь*, и это никак не связано с модальностью связи. Подробно понятие идентифицирующей и неидентифицирующей связи будет рассмотрено далее.

Существует еще одна стандартизованная нотация моделей реляционной базы банных, называемая IDEF1X. Сущности в этой нотации, как и раньше, изображаются прямоугольником, связи — линиями. Мощность связи со стороны "много" изображается черным кружком. Необязательность связи со стороны "один" указывается ромбом. В нотации IDEF1X набор средств для отображения модальности связи более ограничен по сравнению с другими нотациями, т. к. нет возможности обозначать обязательность участия в связи сущности со стороны "много". Последнее обстоятельство не является существенным недостатком, поскольку методика перехода от ER-диаграммы к физической модели требует одинакового приема, результат которого не зависит от модальности связи со стороны "много". Заметим, что в CASE-средстве Erwin в нотации IE тоже невозможно показать графически модальность связи со стороны "много". Пример ER-диаграммы для IDEF1X показан на рис. 1.7.

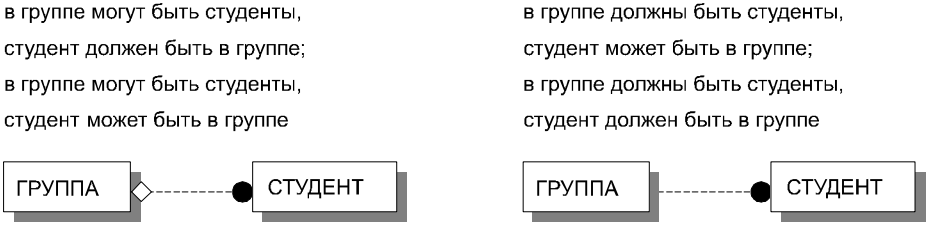


Рис. 1.7. Способы изображения связи в IDEF1X-нотации

Замечание

Рис. 1.5, 1.6 и 1.7 построены как модель реляционной базы данных с помощью CASE-средства Erwin. При этом выбран режим отображения модели на уровне сущностей (опции меню **Display Level/Entity**). Описание модальности связи оформлено как комментарии с помощью текстовых блоков. Изображение обязательности связи со стороны "много" для нотации IE осуществлено удалением соответствующего кружка в графическом редакторе.

Для создания моделей данных можно использовать обе нотации: как IDEF1X, так и IE. Методология IDEF1X была разработана для армии США и широко распространена в государственных учреждениях США, финансовых и промышленных корпорациях. Методология IE используется преимущественно в промышленности. Обе эти нотации применяются в большинстве CASE-средств, таких как Erwin, PowerDesigner, S-Designer.

1.6.2. Нотация Чена

Логическую модель реляционной базы данных также представляют в виде ER-диаграмм в нотации Чена. Возможно использование диаграмм ER-экземпляров, которые являются более подробной формой изображения ER-диаграммы.

На ER-диаграммах в нотации Чена сущность обозначается прямоугольником, содержащим имя сущности, а связь — ромбом, от которого проведены линии к каждой из сущностей, участвующих в связи. Числа над линиями означают кардинальность связи. Класс принадлежности (модальность) связи для каждой сущности показывается наличием прямоугольной рамки вокруг точки. Под прямоугольниками указывается ключевой атрибут сущности. Неключевые атрибуты при необходимости можно изображать так, как это сделано на рис. 1.2. Пример ER-диаграммы в нотации Чена приведен на рис. 1.8.

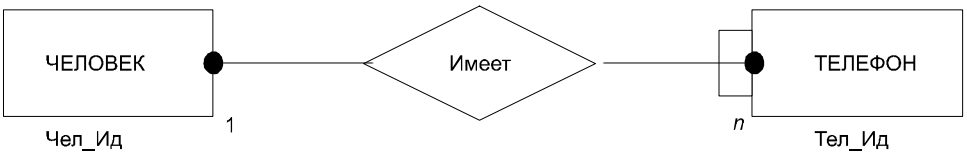


Рис. 1.8. ER-диаграмма в нотации Чена

Диаграммы ER-экземпляров являются еще более подробным способом изображения связей между сущностями, но при этом слишком громоздким. На них точками изображаются все допустимые значения ключевых атрибутов сущностей, вступивших в связь, а с помощью линий представлены возможные связи между этими точками. Пример диаграммы ER-экземпляров приведен на рис. 1.9.

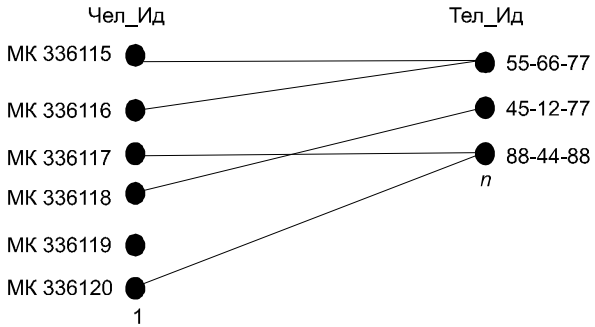


Рис. 1.9. Пример диаграммы ER-экземпляров

При моделировании базы данных мы будем пользоваться ER-диаграммами в нотации Чена только на первоначальной стадии в пределах концептуальной модели. Сразу же сформулируем правила преобразования ER-диаграммы в логическую модель, которая будет изображаться в IDEF1X-нотации в рамках выбранного CASE-средства. Диаграммы ER-экземпляров будут использоваться только для обоснования правил перехода от ER-диаграммы к моделям следующих уровней, основанных на отношениях, ключах отношений и связях внешнего ключа.

1.6.3. Идентифицирующие и неидентифицирующие связи. Сильные и слабые сущности

Как уже стало ясно, основные нотации моделей реляционной базы данных основаны на понятиях идентифицирующей и неидентифицирующей связи.

Неидентифицирующая связь имеет место, когда в связь вступают две самостоятельные сущности. Сама связь осуществляется путем так называемой *миграции ключевых атрибутов* родительской сущности в область неключевых атрибутов дочерней сущности. Мигрирующие атрибуты в дочерней сущности подчиняются определению внешнего ключа. Обозначается неидентифицирующая связь прерывистой линией.

Идентифицирующая связь является особым случаем обязательной связи. Если сущность *A* связана с сущностью *B* идентифицирующей связью, то существование экземпляра *B* невозможно и не имеет смысла без существования экземпляра *A*. Сущность *B* в этом случае называют *зависимой* или *слабой* сущностью. Существование экземпляра такой сущности всегда зависит от существования экземпляра другой сущности.

Идентифицирующая связь практически всегда носит характер "один-ко-многим".

Экземпляр *сильной*, или независимой, сущности существует сам по себе, независимо от всех других сущностей.

На диаграммах слабые сущности обозначаются прямоугольниками с закругленными краями, в то время как прямоугольник, изображающий сильные сущности, имеет прямые углы. Так принято в нотациях IDEF1X и IE. При этом идентифицирующая связь обозначается сплошной линией, а неидентифицирующая — прерывистой.

Не следует смешивать идентифицирующие связи с обязательными. Обязательная связь не обязательно должна быть идентифицирующей: одна сильная сущность может быть в обязательной связи с другой сильной сущностью (если студент обязан быть в группе, это не значит, что его существование без группы вообще не имеет смысла).

При реализации идентифицирующей связи миграция первичного ключа происходит таким образом, что первичный ключ сущности со стороны "один" становится частью первичного ключа сущности со стороны "много", оставаясь при этом внешним ключом. Когда из двух отношений мигрируют два одноименных первичных ключа, то целесообразно провести так называемую *интеграцию внешнего ключа*, при которой два внешних ключа объединяются в один. В Erwin интеграция внешнего ключа выполняется автоматически, а выглядит это так, как показано на рис. 1.10.

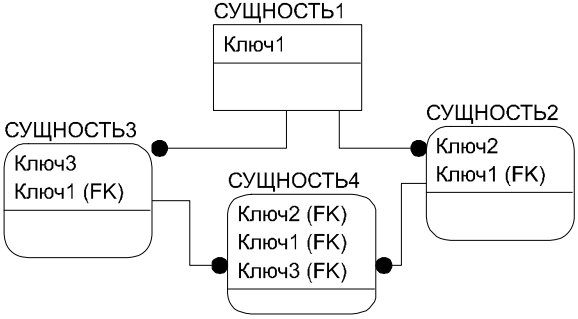


Рис. 1.10. Пример интеграции внешнего ключа

Если же в отношении с именем `СУЩНОСТЬ4` ключи с одинаковым именем `Ключ1`, мигрировавшие из разных сущностей, могут существовать сами по себе (неидентифицирующая связь), то подход, основанный на интеграции внешнего ключа, не годится, и нужно различать внешние ключи с помощью ролевых имен.

1.7. Компоненты реляционной модели в методологии "сущность-связь"

Как уже говорилось, методология "сущность-связь" в простейшем виде ставит своей целью поиск ответов на следующие вопросы: какие в предметной

области имеются объекты, заслуживающие внимания (эти объекты называются *сущностями*), какими эти объекты обладают качествами и свойствами (эти качества и свойства называются *атрибутами*) и какие между этими объектами просматриваются *связи*. Ответы на эти вопросы должны рассматриваться в контексте сформулированного описания предметной области с учетом всех аспектов ее реализации в выбранной информационной модели.

В качестве средства достижения поставленной цели данный подход использует представление модели данных в графическом виде с использованием определенных нотаций.

1.7.1. Сущности

Неформально сущности представляют собой вещи, которые существуют в рамках определенной предметной области. Например, в качестве сущностей могут выступать люди, места, среда, объекты, события, роли, которые все эти вещи играют. *Сущность* — это событие реального мира, измеримая вещь или неизмеримая концепция. Сущность всегда представляет совокупность вещей, а не только один экземпляр. В рамках методологии семантического моделирования реляционных данных сущность является основным понятием, вокруг которого строится вся методика. Но попытки дать формальное определение этому понятию трудно считать удовлетворительными. Поэтому обычно вслед за определением сущности сразу же приводятся примеры, которые в значительной степени проясняют суть дела.

Итак, *сущность* — это класс однотипных объектов, информация о которых должна быть учтена в модели. Эти объекты должны иметь *экземпляры*, отличающиеся друг от друга и допускающие однозначную идентификацию. Каждая сущность должна иметь уникальное в рамках данной модели наименование. При словесном описании предметной области сущности это, как правило, существительное. Наименование сущности рекомендуется задавать существительным в единственном числе. Однако Microsoft в своих учебных примерах нередко дает названия таблицам, за которыми стоят конкретные сущности, используя существительные во множественном числе. Так, в базе данных `NorthWind` имеются таблицы с именами `ORDERS`, `DETAILS` и т. д.

Формальных определений понятия сущности, аналогичных друг другу, можно привести множество, например:

- *сущность* — это имеющее смысл в данной предметной области, существующее в действительности или воображаемое явление или объект, информацию о котором нужно сохранять или выяснять (Р. Баркер);
- *сущность* — это информационный контейнер в первой нормальной форме, который сохраняет фиксированный набор атрибутов и может существо-

вать в числе экземпляров от нуля до нескольких, являясь реляционной абстракцией некоторой концепции реального мира.

Иногда класс экземпляров называют типом сущности, а экземпляр — просто сущностью, но это затрудняет понимание.

Экземпляр сущности (entity instance) — это конкретный представитель данной сущности. Экземпляры сущностей должны быть *различимы*, т. е. сущности должны иметь некоторые свойства, уникальные для каждого экземпляра этой сущности.

Ключ сущности — это *неизбыточный* набор атрибутов, значения которых в совокупности являются *уникальными* для каждого экземпляра сущности. Неизбыточность заключается в том, что удаление любого атрибута из ключа нарушает его уникальность.

Поскольку в реляционной модели сущность в конце концов преобразуется в отношение, то определение ключа сущности практически аналогично определению ключа отношения. Напомним, что ключи бывают атомарными и составными. Атомарный ключ состоит из единственного атрибута. Составной ключ представляет собой набор атрибутов.

Наборы ключевых атрибутов сущностей определяются на самых ранних стадиях проектирования базы данных исходя из объективных связей между данными. Содержимое того или иного экземпляра сущности на конкретный момент времени не является основанием для определения в качестве ключа произвольных наборов, не совпадающих по значениям экземпляров атрибутов.

По окончании проектирования при переходе к физической модели сущности должны быть определены как таблицы, первичные и альтернативные ключи сущностей — как уникальные индексы, атрибуты сущностей — как поля этих же таблиц, а экземпляры сущностей — как записи с данными в соответствующих таблицах. Как правило, опытный бизнес-аналитик сразу, еще на стадиях концептуального и логического моделирования мыслит именно этими категориями.

Имя сущности должно всегда обозначать *тип или класс вещей*, а не отдельный экземпляр, например, в качестве имени нужно выбирать `ПРЕДМЕТ` (общий тип вещей "Предмет для изучения"), а не "Базы данных" (конкретный предмет). В приведенном примере "Базы данных" будут являться *экземпляром сущности* `ПРЕДМЕТ`.

Основное правило для сущностей следующее: каждый экземпляр сущности должен уникальным и четко определимым образом отличаться от любого другого экземпляра этой сущности и от любых экземпляров других сущностей.

Тем самым выполняется правило, что каждое явление или объект может быть представлен только одной сущностью. Не может быть двух одинаковых сущностей, и должен существовать *уникальный идентификатор для каждой сущности* (no entity without identity).

Каждая сущность должна обязательно сопровождаться определением, которое обычно отвечает на два вопроса.

- Что отличает экземпляры данной сущности от экземпляров других сущностей?
- Что отличает экземпляры данной сущности друг от друга?

Ответы на эти вопросы и определяют выполнение основного правила для сущностей. Кроме этой информации, определение может включать в себя примеры экземпляров данной сущности, примеры атрибутов, возможные запросы для получения данных на основании сущности и т. д.

В качестве примера можно привести определение сущности `ПРЕДМЕТ`.

ПРЕДМЕТ *определяет тему* ЛЕКЦИОННОГО КУРСА, *который читается конкретной* ГРУППЕ, *а также (необязательно) тему* ПРОЕКТА.

Примеры экземпляров сущности `ПРЕДМЕТ`: "Базы данных", "C++", "Английский язык".

1.7.2. Атрибуты

Атрибут сущности — это именованная характеристика, являющаяся некоторым свойством сущности. Наименование атрибута должно быть выражено существительным в единственном числе (возможно, с характеризующими прилагательными).

Атрибутом называется базовая единица информации об экземпляре сущности, которая этот экземпляр качественно или количественно оценивает, идентифицирует, классифицирует или выражает его состояние. *Атрибут* — это свойство сущности, которое зависит только от этой сущности и ни от чего другого и может иметь только одно значение в данный момент времени.

При проектировании атрибутов полезно задавать такие вопросы.

- Какие данные о сущности мы хотим хранить?
- Какие свойства есть у экземпляра этой сущности, даже если вокруг нее больше ничего нет?
- Есть ли у экземпляра этой сущности только один экземпляр этой вещи?
- Может ли изменяться описанная атрибутом характеристика сущности с течением времени?

Например, у человека всегда имеется только один возраст, у здания — только одна высота в данный момент времени, у человека всегда есть рост, вес и возраст.

Имена атрибутов рекомендуется делать *уникальными* в рамках всей схемы базы данных. Поскольку атрибут описывает только одну сущность, он и должен быть во всей схеме один (по правилу "один факт — в одном месте"). Такие средства, как Erwin, позволяют не допускать повторения имен атрибутов в рамках модели.

Укажем некоторые правила, которыми полезно пользоваться при проектировании атрибутов.

- *Атрибут должен описывать именно свою сущность*. Это простое на первый взгляд правило на самом не так просто применять на практике. Затруднения возникают, когда для уже имеющейся части модели обнаруживается новый атрибут, после чего производится попытка найти для него подходящую сущность. Ее может и не оказаться, и, следовательно, модель придется изменить довольно серьезно.
- *Не стоит объединять в один атрибут разнородные данные*. Это соображение чаще всего относится к попыткам соединить в одном атрибуте количество и единицу измерения, например, хранить в одном атрибуте что-то вроде "20 ящиков" и "100 бутылок" или "100 руб." и "10 долларов" и т. п. Следует разделять подобные атрибуты на атрибут для хранения количества и атрибут для хранения единицы измерения.
- *Не рекомендуется создавать атрибуты, которые могут иметь разный смысл в зависимости от своего конкретного значения*. Например, атрибут `КЛИЕНТ` может содержать данные о физическом или юридическом лице. Понятно, что эти данные окажутся совершенно различными по своему смыслу, и без информации о том, к какому типу относится клиент, трактовать их крайне затруднительно.
- *Не стоит задавать "особенных" диапазонов значений и правил трактовки данных*. То есть следует избегать ситуаций, когда, например, атрибут `ОЦЕНКА` содержит оценку, полученную студентом на экзамене (число 3, 4 или 5) в том случае, если экзамен сдан успешно; `NULL`-значение, если срок сдачи экзамена еще не наступил, и отрицательное число, которое обозначает, что экзамен сдавался несколько раз (модуль значения), но так и не был сдан.

Атрибуты бывают *обязательными*, т. е. такими, которые не могут принимать `NULL`-значений, и *необязательными*, если `NULL`-значения для них допустимы. Согласно правилу целостности сущностей `NULL`-значения недопустимы для атрибутов, выбранных в качестве первичных ключей сущности.

Разрешается существование так называемых *производных (вычислимых) атрибутов* — т. е. таких, значение которых никогда не устанавливается напрямую ни пользователем, ни как бы то ни было еще, но всегда вычисляется в тот момент, когда оно требуется на основе базовых атрибутов. Введение производных атрибутов тесно связано с проблемой внесения контролируемой избыточности в базу данных и подробно обсуждается в *главе 2*. Пока же только кратко отметим, что имеются два подхода к реализации производных атрибутов: можно перевычислять их значения только тогда, когда меняются значения *базовых* атрибутов, на основании которых это значение вычисляется, а можно пересчитывать его при каждом доступе. Первый подход эффективнее, если базовые атрибуты меняются реже, чем обращаются к производному атрибуту, а второй — если базовые атрибуты меняются чаще.

Иногда удается выделить сущности, исследуя предметную область в поисках атрибутов, для которых пока что не нашлось сущностей.

Другой источник сущностей — это атрибуты, для которых оказывается важным хранить дополнительную информацию. Здесь нужно применять простое правило: если необходимо хранить дополнительную информацию об атрибуте, значит, это сущность. Такие сущности обычно называются *атрибутивными сущностями*. Пример атрибутивной сущности: аудитория, в которой происходят занятия, могла бы быть атрибутом сущности "проведение занятия", но если нужно хранить характеристики аудитории, такие как вместимость, наличие доски и т. д., то целесообразно сделать ее атрибутивной сущностью.

1.7.3. Связи

Как обычно, в ER-методе можно привести несколько похожих по смыслу и мало что объясняющих определений связи. Например:

- *связь* является логическим соотношением между сущностями. Каждая связь должна именоваться глаголом или глагольной фразой (Relationship Verb Phrase);
- *связь* — это некоторая ассоциация между несколькими сущностями. Другими словами, связь представляет собой соединение между двумя или более сущностями и обычно выражается глаголом;
- *связь* — это отношение или правило, существующее между сущностями, непротиворечивое, неизменяемое выражение некоторого правила предметной области или логический объект, который может быть прикреплен к другой сущности;
- *связь* — это именованная, значащая ассоциация между двумя сущностями или сущности с самой собой (Р. Баркер).

Одна сущность может быть связана с другими сущностями или сама с собой. Приведем следующие примеры описания связей:

- студент *входит* в группу;
- студент *может являться* старостой группы;
- потоку, *состоящему из нескольких* групп, *читается* лекция по определенному предмету;
- курс *может читаться только после* другого курса;
- сотрудник *может иметь* несколько детей или *не иметь* их вовсе.

Связь между двумя сущностями называется *бинарной*. Это основной тип связи, с помощью которого в конечном счете выражаются связи другого типа. Для каждого из двух концов бинарной связи задаются основные *характеристики* связи:

- *название*, которое описывает семантику связи, с точки зрения одной из сущностей;
- *кардинальное число (степень, мощность)*, которое описывает количество возможных связей для каждой из сущностей-участниц;
- *степень участия сущности в связи (обязательность связи, модальность связи, класс принадлежности)*.

Как известно, бинарная связь бывает "один-к-одному" (1:1), "один-ко-многим" ($1 : n$) или "много-ко-многим ($n : m$), что задается опцией, называемой *мощность связи*, или *кардинальность связи* (cardinality). Кроме того, связь характеризуется таким параметром, как модальность, который может принимать всего два значения: обязательная и необязательная.

Связи на диаграмме обозначаются линиями, соединяющими сущности. Концы этих линий выделяются различными символами, задающими степень связи и ее обязательность.

В стандартной ER-нотации посередине линии задается ромб, но это, как и в случае задания эллипсов для атрибутов, ведет к перегруженности диаграммы и, кроме того, отвлекает от смысла связи как отношения между сущностями, т. к. этот ромб выглядит почти как сущность.

Если речь идет о бинарных связях, то каждая связь имеет два конца и одно или два наименования. Наименование обычно выражается в неопределенной глагольной форме: "иметь", "принадлежать" и т. п. Каждое из наименований относится к своему концу связи. Иногда наименования не пишутся ввиду их очевидности.

Всегда удобно давать не одно, а два названия связи с двух сторон линии, т. е. с точки зрения сущностей, по обе стороны связи, например, студент *находится в* группе, группа *состоит из* студентов.

Правила записи названий связей различны: иногда имя пишут возле той сущности, с точки зрения которой данная связь носит это имя. В нашем примере слово *"находится"* расположено возле сущности СТУДЕНТ. В других случаях название связи размещают возле противоположной сущности, на которую эта связь действует, а иногда (например в Erwin) оба имени пишутся вместе, разделенные символом "/". При этом в случае связи "один-ко-многим" первым пишется название с точки зрения сущности со стороны "один", например *"состоит из/находится в"*.

Бинарные связи, их параметры и классификация

Бинарные связи — это связи, в которые вступают ровно две сущности. Важнейшее свойство связи — кардинальное число. Синонимами этого понятия являются мощность связи или степень связи. Оно описывает максимально возможное количество связей для каждого из экземпляров сущностей, участвующих в связи. В зависимости от значения кардинального числа различают бинарные связи следующих типов:

- связь типа *"один-к-одному"* означает, что один экземпляр первой сущности связан не более чем с одним экземпляром второй сущности и, наоборот, один экземпляр второй сущности связан не более чем с одним экземпляром первой сущности;
- связь типа *"один-ко-многим"* означает, что один экземпляр первой сущности связан с несколькими экземплярами второй сущности, но при этом один экземпляр второй сущности связан не более чем с одним экземпляром первой сущности. Сущность со стороны "один" (в нашем случае вторая сущность) называется *родительской*, а со стороны "много" (в нашем случае первая сущность) — *дочерней*. На рис. 1.8 как раз изображен тип связи "один-ко-многим". Это наиболее часто используемый тип связи;
- связь типа *"много-ко-многим"* означает, что каждый экземпляр первой сущности может быть связан с несколькими экземплярами второй сущности, и каждый экземпляр второй сущности может быть связан с несколькими экземплярами первой сущности. Связь типа "много-ко-многим" должна быть заменена двумя связями типа "один-ко-многим" путем создания промежуточной сущности.

Вторым параметром связи является ее обязательность, называемая также *модальностью связи*, или *классом принадлежности*. Каждая связь может иметь одну из двух модальностей:

- модальность *"может"* соответствует необязательному классу принадлежности и означает, что любой экземпляр сущности с таким типом модальности может быть связан с одним или несколькими экземплярами другой сущности или не связан ни с одним экземпляром другой сущности;

- модальность *"должен"* соответствует обязательному классу принадлежности и означает, что каждый экземпляр сущности *обязан быть связан не менее чем с одним* экземпляром другой сущности.

Бинарная связь может иметь *разную модальность* и *разную кардинальность* с разных концов. В зависимости от комбинаций параметров кардинальности и модальности бинарной связи выделяют шесть ситуаций, каждой из которых соответствует определенное *проектное решение в виде набора предварительных отношений*, выраженное стандартными средствами реляционной модели данных.

Под проектным решением в вышеупомянутом контексте мы будем понимать графическое изображение набора сущностей с указанием их первичных ключей, полного перечня атрибутов, а также возможных альтернативных ключей, со связями, реализованными с помощью ограничения внешнего ключа. Проектное решение должно быть представлено с использованием выбранного CASE-средства (в нашем случае Erwin) в одной из общепринятых нотаций (IE или IDEF1X) на уровне логической модели реляционной базы данных.

1.7.4. Основные этапы логического проектирования в методологии "сущность-связь"

Проектирование базы данных — это всегда отдельная творческая задача, не имеющая полностью формализованного решения. Приведенный далее алгоритм проектирования не является собственно алгоритмом в формальном смысле. Скорее, приведенные ниже пункты можно рассматривать как наиболее общие рекомендации, направленные на создание целостной OLTP-системы, адекватно отображающей предметную область и оптимизированной, в первую очередь, по критерию быстродействия. Придерживаясь этих рекомендаций, вероятнее всего разные проектировщики получат различные решения. Вопрос о том, какое из этих решений является наилучшим, обычно остается без ответа, т. к. реализовать несколько проектных решений и сравнить их на практике невозможно. Кроме того, достаточно большое количество мероприятий по повышению быстродействия системы находится за рамками логической модели, а выполняется на уровне физического моделирования и программно-аппаратной реализации базы данных. Все же можно попытаться свести в некоторый перечень основные действия, которые обычно приходится выполнять в процессе проектирования реляционной базы данных с использованием метода "сущность-связь" при нормализации проектного решения до третьей нормальной формы. Эти действия следующие:

- детально изучить предметную область в целях составления ее словесного описания и конкретизации необходимых бизнес-правил. Сформулировать

функциональные требования к базе данных, выявив основные запросы и транзакции;

- выявить сущности, которые необходимо включить в модель, и определить первичные ключи каждой из выявленных сущностей;
- выявить и описать смысл, тип и возможные значения всех представляющих интерес атрибутов для каждой сущности. Определить домены для каждого атрибута;
- определить типы связей между выявленными сущностями;
- построить графическое описание логической модели предметной области в виде ER-диаграммы с указанием степеней связей и классов принадлежностей. Начиная с этого момента целесообразно применять какое-либо CASE-средство проектирования;
- построить графическое описание функциональных зависимостей между атрибутами и выявить минимальное покрытие;
- для построенной ER-диаграммы следует получить предварительные отношения, пользуясь правилами их генерации, и разместить в них выявленные атрибуты сущностей;
- выполнить нормализацию модели, используя минимальное покрытие;
- принять обоснованные решения по введению в модель контролируемой избыточности и способов ее контроля;
- получить окончательное проектное решение в виде наборов отношений с присвоенными им смысловыми именами, указанными ключами и перечнем атрибутов для каждого отношения. Проверить, все ли атрибуты нашли свое место;
- убедиться в возможности реализации необходимой функциональности базы данных на полученном проектном решении. После этого можно переходить к работе с физической моделью базы данных и генерировать ее системный каталог.

Разумеется, на практике эти пункты выполняются не в жесткой последовательности. Опытные проектировщики обычно подразумевают наличие ER-диаграммы или концептуальной модели и зачастую ограничиваются изображением логической модели базы данных сразу в нотации выбранного CASE-средства проектирования. По мере работы возможно возвращение к каким-либо этапам по несколько раз, при этом могут вноситься значительные уточнения вплоть до создания новых сущностей. Таким образом, процесс логического моделирования реляционной базы данных носит итерационный характер: результаты очередного этапа часто вызывают изменения в проектных решениях, выработанных на более ранних этапах.

1.8. Приемы построения проектных решений по ER-диаграммам

Как уже неоднократно упоминалось, на данный момент разработаны соответствующие CASE-средства, позволяющие работать с логическими и физическими моделями реляционной базы данных в графическом виде. Одной из самых простых программ подобного рода является CASE-средство Erwin корпорации PLATINUM. Краткое руководство по работе с ней приведено в *разд. 1.9*.

Нотации CASE-средств используют несколько другие базовые понятия, чем ER-диаграммы. Это касается в первую очередь способа изображения связей. Поскольку реально модель рисуется сразу с помощью CASE-средства, то нотация выбранного CASE-средства и будет считаться основной.

В нотации CASE-средств, используемых для инфологического моделирования, связь изображается линией и бывает *идентифицирующей* (сплошная линия) или *неидентифицирующей* (пунктирная линия). Сама связь осуществляется путем миграции ключевых атрибутов родительской сущности в область ключевых атрибутов дочерней сущности для идентифицирующей связи и в область неключевых атрибутов для неидентифицирующей связи. Способ миграции атрибутов при образовании связи и определяет ее тип. Мигрирующие атрибуты в дочерней сущности подчиняются определению внешнего ключа, и возле них ставится соответствующая пометка — *FK*.

Рассмотрим способы изображения связей обоих типов в IDEF1X-нотации CASE-средства Erwin. На рис. 1.11 изображены оба типа связи: идентифицирующая связь и неидентифицирующая. При этом на обеих связях рис. 1.11 `СУЩНОСТЬ2` является родительской, т. к. она участвует в связи со стороны "один", а `СУЩНОСТЬ1` — дочерней, т. к. она участвует в связи со стороны "много".

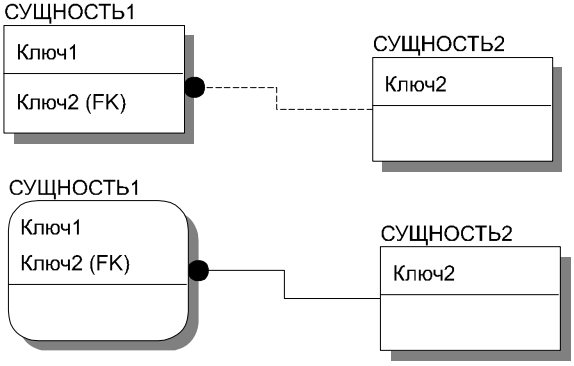


Рис. 1.11. Пример изображения связей в логической модели в нотации IDEF1X

Из приведенного рисунка четко видно отличие идентифицирующей связи от неидентифицирующей, которое заключается в способе миграции первичного ключа родительской сущности.

Как уже говорилось, бинарная связь — это связь между двумя сущностями. В рамках концептуального моделирования удается, как правило, выделить сущности предметной области, определить типы связей между ними и отобразить эту информацию на ER-диаграммах. Далее ER-диаграмма должна быть введена в компьютер. Для этого необходимо сформировать так называемый набор *предварительных отношений*, которые будут включать в себя отношения, необходимые для хранения собственно сущностей с указанием их первичных ключей. Кроме того, должны быть созданы отношения, реализующие связи.

Правильно построенные ER-диаграммы позволяют генерировать предварительные отношения по готовым правилам, изображая их сразу в нотации выбранного CASE-средства. Построение предварительных отношений — это, в первую очередь, работа с ключами сущностей и изображение таких параметров связей, как кардинальность и модальность средствами миграции ключевых атрибутов сущностей, т. е. приведение ER-диаграммы к такому состоянию, когда параметры ее связей выражены с помощью идентифицирующих или неидентифицирующих связей, реализующих ограничение внешнего ключа.

Кроме картинок можно работать непосредственно с перечнем предварительных отношений, т. к. он намного компактнее, чем рисунок, хотя таким способом не могут быть отображены связи. Ключи полученных предварительных отношений подчеркнуты. Если ключей два, то они подчеркнуты по-разному. ER-диаграммы изображаются в нотации Баркера. Предварительные отношения — в IDEF1X-нотации логической модели.

Для некоторых правил приведены примеры описания предметной области, требующие их применения. Кроме того, большое число подробных примеров не только с ER-диаграммами, но и с диаграммами ER-экземпляров можно найти в работе [10].

Приведенные правила позволяют генерировать корректный набор предварительных отношений для каждого типа связи. Известно, что любой тип бинарной связи можно разрешить через три предварительных отношения. Правила же выводятся из требования минимизации числа предварительных отношений при условии отсутствия `NULL`-значений в полях, задающих ключи сущностей. Кроме того, сгенерированные по правилам предварительные отношения наиболее подходят для их нормализации до третьей нормальной формы. Таким образом, уже на уровне логического моделирования принимается в рассмотрение такой важный критерий качества базы данных, как нормализация.

При данном подходе физическая модель генерируется только с помощью таблиц. При использовании правил объекта "связь" нет как такового, что делает данную методику универсальной и пригодной даже для тех СУБД, где объект "связь" отсутствует.

Вместе с каждым правилом будет рассказано, как реализовать сгенерированные им предварительные отношения с помощью такого CASE-средства проектирования базы данных, как Erwin.

1.8.1. Бинарные связи с обязательным классом принадлежности

Начнем рассмотрение с бинарных связей с обязательным классом принадлежности.

Правило 1 (рис. 1.12). Если кардинальность бинарной связи равна 1:1 и класс принадлежности обеих сущностей является обязательным, то требуется только одно отношение. Ключом этого отношения может быть ключ любой из двух сущностей.

Рис. 1.12. ER-диаграмма для связи 1:1 с обязательным классом принадлежности для обеих сущностей

Предварительное отношение для правила 1:

`R1`(`Ключ1`, `Ключ2` ...) — соответствует сущности `СУЩНОСТЬ1`, сущности `СУЩНОСТЬ2` и является отношением связи.

Примером для данного правила может быть ситуация, когда в базе данных для каждого сотрудника, обязательно идентифицируемого уникальным табельным номером (`Ключ1`), требуется знать его уникальный налоговый код (`Ключ2`). Так как по законодательству у каждого совершеннолетнего гражданина налоговый код должен обязательно существовать и быть уникальным, то данная ситуация описывается как раз связью из правила 1.

Рассмотрим возможность связи из правила 1 между табельным номером сотрудника и его номером паспорта. Нарисуем ER-диаграмму для этого случая (рис. 1.13).

Очевидно, что со стороны сущности `СОТРУДНИК` модальность связи всегда будет обязательной до тех пор, пока человек является сотрудником организации, представляющей интерес для предметной области. Действительно, если

базу данных заказывает именно эта организация, то она может устанавливать правило по идентификации своих сотрудников так, что, например, табельный номер не может быть потерян. Паспорт же выдается внешней по отношению к этой организации структурой. Правила выдачи паспорта не могут быть установлены при проектировании базы данных. Поэтому, во-первых, обязательность связи со стороны сущности `ГРАЖДАНИН` может оказаться под вопросом, если, например, сотрудник потерял паспорт или его паспорт находится в процессе замены. Во-вторых, в данной ситуации нельзя выбирать атрибут `Паспорт` в качестве первичного ключа полученного предварительного отношения, т. к. его изменения невозможно контролировать.

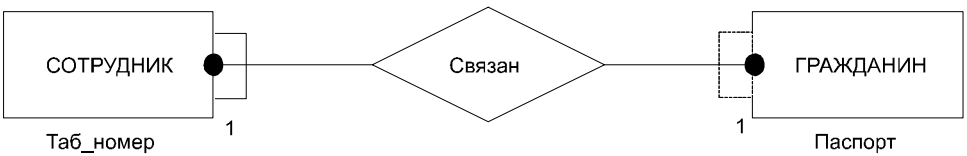


Рис. 1.13. ER-диаграмма для связи 1:1 при возможности нарушения обязательности класса принадлежности для одной из сущностей

Для того чтобы реализовать правило 1 в Erwin, проще всего создать единственную сущность, например с именем `R1`, в которую помещаются оба ключа. После этого какой-либо из ключей `Ключ1` или `Ключ2` объявляется первичным ключом созданной сущности, а по другому ключу создается альтернативный уникальный вход (уникальный индекс). Сама эта сущность будет выглядеть, как показано на рис. 1.14.

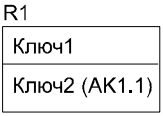


Рис. 1.14. Реализация связи из правила 1 в Erwin

Правило 2 (рис. 1.15). Если степень бинарной связи равна *n* : 1 и класс принадлежности *n*-связной сущности является обязательным (класс принадлежности односвязной сущности роли не играет), то достаточно использовать два предварительных отношения, по одному на каждую сущность, при условии, что ключ каждой сущности служит ключом для соответствующего отношения. Дополнительно ключ односвязной сущности должен быть добавлен как атрибут в отношение, отводимое *n*-связной сущности.

Предварительные отношения для рис. 1.15:

- `R1(Ключ1, Ключ2 ...)` — соответствует сущности `СУЩНОСТЬ1` и является отношением связи;
- `R2(Ключ2 ...)` — соответствует сущности `СУЩНОСТЬ2`.

Рис. 1.15. ER-диаграмма для связи *n*:1 с обязательным классом принадлежности *n*-связной сущности

Связи по правилу 2 являются наиболее распространенными. Если в качестве `СУЩНОСТЬ1` выбрать студента дневного отделения института и идентифицировать его номером студенческого билета (`Ключ1`), то в качестве `СУЩНОСТЬ2` можно выбрать группу, в которой этот студент обучается, идентифицируемую номером (`Ключ2`).

В Erwin тип связи, описанный правилом 2, соответствует неидентифицирующей связи с запрещенными `NULL`-значениями в мигрирующем атрибуте внешнего ключа (рис. 1.16).

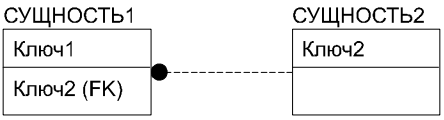


Рис. 1.16. Реализация связи из правила 2 в Erwin

Замечание

В приведенных обозначениях в терминологии IDEF1X-нотации `СУЩНОСТЬ2` является родительской, а `СУЩНОСТЬ1` — дочерней.

Для бинарной связи "много-ко-многим" класс принадлежности не влияет на правило генерации предварительных отношений. Это обстоятельство на ER-диаграмме отображено пунктиром. В связях этого типа невозможно уменьшить количество отношений до значения меньшего, чем три. Приходится вводить *ассоциативную сущность* `R3`.

Рис. 1.17. ER-диаграмма для связи *m*:*n*

Правило 3 (рис. 1.17). Если степень бинарной связи равна $m : n$, то необходимы три предварительных отношения по одному для каждой сущности,

причем ключ каждой сущности используется в качестве ключа соответствующего отношения; и третье *отношение для связи*, которое должно иметь в числе своих ключевых атрибутов ключ каждой сущности. То есть отношение связи имеет составной ключ, в который включаются оба ключа исходных сущностей.

Предварительные отношения для рис. 1.17:

- R1(Ключ1 ...) — соответствует сущности СУЩНОСТЬ1;
- R2(Ключ2 ...) — соответствует сущности СУЩНОСТЬ2;
- R3(Ключ1, Ключ2 ...).

Примером, соответствующим правилу 3, является связь между СУЩНОСТЬ1, в качестве которой выступает преподаватель, идентифицируемый табельным номером (Ключ1), и СУЩНОСТЬ2, в качестве которой выступает студент, идентифицируемый номером студенческого билета (Ключ2). Сама связь описывает ситуацию "студент обучается у преподавателя".

В Erwin связь из правила 3 соответствует связи, называемой *ассоциативной*, которая автоматически разрешается через третье отношение R3. На рис. 1.18 представлена связь "много-ко-многим" как без ее разрешения через третье отношение, так и с явным введением отношения связи R3(Ключ1, Ключ2 ...).

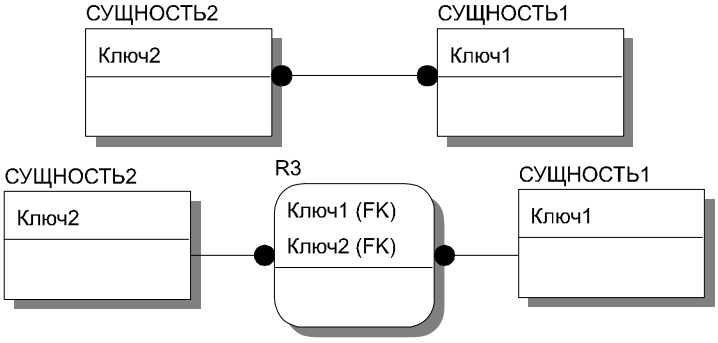


Рис. 1.18. Связь "много-ко-многим" в нотации IDEF1X

Чтобы добиться в Erwin автоматического создания ассоциативной сущности (или отношения связи) R3 в режиме логической модели, необходимо выделить ассоциативную связь с двумя кружками на обеих ее концах и в контекстном меню нажать опцию **Resolve Many To Many**. Можно просто создать третью сущность R3, после чего создать две идентифицирующие связи вручную, используя сущности СУЩНОСТЬ1 и СУЩНОСТЬ2 как родительские, а сущность R3 — как дочернюю.

1.8.2. Бинарные связи с необязательным классом принадлежности

Рассмотрим ситуации, когда класс принадлежности должен учитываться при создании проектного решения по ER-диаграмме. На этих примерах можно проиллюстрировать принципы обоснования правил, для чего будут использоваться еще и диаграммы ER-экземпляров.

Правило 4 (рис. 1.19). Если степень бинарной связи равна 1:1 и класс принадлежности ни одной из сущностей не является обязательным, то необходимо использовать три отношения, по одному для каждой сущности, ключи которых служат в качестве ключей в соответствующих отношениях, и одно отношение для связи. Среди атрибутов отношение, выделяемое связи, будет иметь по одному ключу от каждой сущности. Любой из этих атрибутов можно выбрать ключом отношения связи.

Рис. 1.19. ER-диаграмма для связи 1:1 с обязательным классом принадлежности для обеих сущностей

Предварительные отношения для связи, приведенной на рис. 1.19:

- R1(Ключ1 ...) — соответствует сущности СУЩНОСТЬ1;
- R2(Ключ2 ...) — соответствует сущности СУЩНОСТЬ2;
- R3(Ключ1, Ключ2 ...) — является отношением связи.

Предварительные отношения получены исходя из предпосылки, согласно которой NULL-значения считаются недопустимыми в ключах сущностей. Посмотрим, почему нельзя ограничиться одним отношением вида R(Ключ1, Ключ2 ...) по аналогии с правилом 1, где класс принадлежности обеих сущностей обязателен.

Рассмотрим гипотетический пример описания предметной области, соответствующий связи из правила 4. Предположим, что проектируется база данных, предназначенная для хранения информации о преподавателях факультета и о тех курсах, за которые они отвечают. Двумя сущностями, представляющими в данном случае интерес, являются ПРЕПОДАВАТЕЛЬ и КУРС. Преподаватели идентифицируются табельными номерами (Таб_номер), а курсы — их названиями (Курс_Ид). На факультете принято правило: каждый преподаватель отвечает не более чем за один курс, и за каждый курс отвечает не более чем

один преподаватель, следовательно, кардинальность данной связи равна 1:1. При этом допускается наличие преподавателей, не отвечающих ни за один из курсов, а также курсов, за которые не отвечает ни один из преподавателей, т. е. обе сущности ПРЕПОДАВАТЕЛЬ и КУРС имеют необязательный класс принадлежности.

Нарисуем диаграмму ER-экземпляров для этого случая (рис. 1.20).

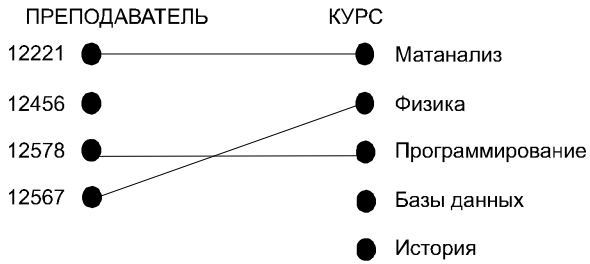


Рис. 1.20. Диаграмма ER-экземпляров для связи 1:1 для сущностей ПРЕПОДАВАТЕЛЬ и КУРС

Если мы создадим одно отношение вида R(Ключ1, Ключ2 ...), т. е. отношение R(Таб_номер, Курс_Ид ...) и станем заносить в него данные, то получим табл. 1.3 следующего вида:

Таблица 1.3. *Данные о курсах и преподавателях, занесенные в одно отношение*

Таб_номер	**Курс**
12221	Матанализ
12456	
12578	Программирование
	Базы данных
	История
12567	Физика

Видно, что у этой таблицы невозможно даже выбрать столбец для первичного ключа, т. к. в обоих столбцах присутствуют NULL-значения и невозможно удовлетворить требованию целостности сущностей. Избежать присутствия NULL-значений в данной ситуации можно, только если распределить данные по трем таблицам, созданным на основе трех предварительных отношений согласно правилу 4, которые приведены в табл. 1.4.

Таким образом, корректность правила 4 проиллюстрирована.

Таблица 1.4. *Данные о курсах и преподавателях, занесенные в три отношения*

Таб_номер	Курс
12221	Матанализ
12567	Физика
12578	Программирование

Таб_номер
12221
12456
12578
12567

Курс
Матанализ
Физика
Программирование
Базы данных
История

Чтобы в Erwin реализовать именно правило 4, не отдавая предпочтения ни одному из ключей, нужно создать между сущностями `СУЩНОСТЬ1` и `СУЩНОСТЬ2` связь "много-ко-многим". После этого в отношении связи следует создать два альтернативных ключа, как показано на рис. 1.21. Однако данный способ не очень удачен, т. к. при использовании двух идентифицирующих связей образуется составной первичный ключ, который нам совершенно ни к чему. В физической модели по нему образуется композитный уникальный индекс, который бессмыслен, поскольку каждая из его компонент сама по себе уже обладает ограничением уникальности.

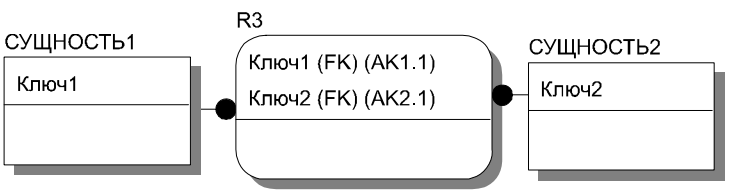


Рис. 1.21. Связь 1:1 из правила 4 в нотации IDEF1X

Наверное, более здравым будет следующее решение. Один из ключей сущностей, участвующих в правиле 4, нужно выбрать первичным ключом отношения связи, и он будет мигрировать в это отношение с помощью идентифицирующей связи, а второй ключ пусть мигрирует в отношение связи с помощью неидентифицирующей связи, после чего сделаем его альтернативным ключом (рис. 1.22).

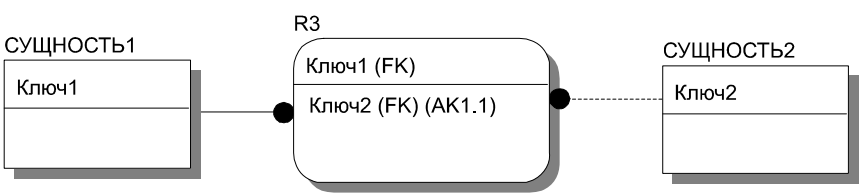


Рис. 1.22. Связь 1:1 из правила 4 в нотации IDEF1X

В физической модели предложенный способ реализации связей 1:1 организован с помощью уникальных индексов, которые по умолчанию создаются по первичным и альтернативным ключам. При переходе к физической модели и далее при ее прямом инжиниринге в целевую СУБД MS SQL-сервер обычно автоматически генерируется диаграмма, на которой стандартное ограничение внешнего ключа преобразуется в связь типа $1 : n$. Если же существуют уникальные индексы по полям связи в родительской и дочерней сущностях, то в целевой СУБД образуется связь типа 1:1. Например, логической модели, приведенной на рис. 1.22, будет соответствовать диаграмма, изображенная на рис. 1.23. Здесь тип связи 1:1 показан наличием соответствующей пиктограммы с обеих сторон от линии, изображающей связь.

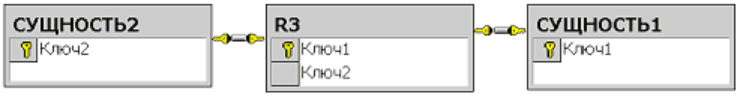


Рис. 1.23. Связь 1:1 из правила 4 в нотации целевой СУБД MS SQL-сервер

В Erwin существует еще одна возможность задания параметров связи с помощью окна свойств связи. В этом случае физическая реализация может оказаться иной и отличаться для различных целевых СУБД.

Замечание

В правиле 4 нужно целых три отношения, для того чтобы нигде не было `NULL`-значений, возникающих в отношении связи из-за необязательного класса принадлежности сущностей. Если в одном из ключевых атрибутов отношения связи допустить `NULL`-значения, то количество самих отношений можно уменьшить. Так, если в `R3` допустить наличие `NULL`-значений для атрибута `Ключ2`, то нет необходимости создавать отношение `СУЩНОСТЬ1(Ключ1 ...)`.

Рис. 1.24. ER-диаграмма для связи из правила 5

Правило 5 (рис. 1.24). Если степень бинарной связи равна 1:1 и класс принадлежности одной сущности является обязательным, а другой — необязательным, то необходимо построение двух отношений. Под каждую сущность следует выделить одно отношение, при этом ключ сущности должен служить ключом для соответствующего отношения. Кроме того, ключ сущности, для которой класс принадлежности является необязательным, добавляется в ка-

честве атрибута в отношение, выделенное для сущности с обязательным классом принадлежности.

В этом случае нужны следующие предварительные отношения:

- R1(Ключ2, ...) — соответствует сущности СУЩНОСТЬ2;
- R2(Ключ1, Ключ2 ...) — соответствует сущности СУЩНОСТЬ1 и является отношением связи.

В Erwin это правило реализуется так, как показано на рис. 1.25.

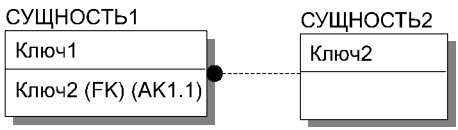


Рис. 1.25. Реализация связи из правила 5 в Erwin

Можно привести такой пример описания предметной области, требующий применения правила 5. Пусть выделено некоторое количество именных стипендий, которыми нужно премировать студентов за какие-то заслуги. Каждому студенту можно вручить только одну именную стипендию, и все стипендии должны быть вручены. Разумеется, не все студенты получают именные стипендии. Тогда именная стипендия выступает в качестве СУЩНОСТЬ1, а студент — в качестве СУЩНОСТЬ2.

Правило 6 (рис. 1.26). Если степень бинарной связи равна $n : 1$ и класс принадлежности n-связной сущности является необязательным, а класс принадлежности односвязной сущности роли не играет, то необходимо формирование трех отношений, по одному для каждой сущности, причем ключ каждой сущности служит ключом соответствующего отношения и одного отношения для связи. Отношение связи должно иметь среди своих атрибутов ключ сущности от каждой сущности.

Рис. 1.26. ER-диаграмма для связи n:1

Предварительные отношения для связи, приведенной на рис. 1.26:

- R1(Ключ1 ...) — соответствует сущности СУЩНОСТЬ1;
- R2(Ключ2 ...) — соответствует сущности СУЩНОСТЬ2;
- R3(Ключ1, Ключ2...) — является отношением связи.

Рис. 1.27. Реализация правила 6 в Erwin

Реализация правила 6 в Erwin показана на рис. 1.27.

Рассмотрим пример описания предметной области, который соответствует правилу 6 и иллюстрирует способ получения этого правила.

Приведем описание предметной области, которое соответствует ситуации связи $n : 1$ с необязательным классом принадлежности n-связной сущности. Пусть, например, у людей, которые идентифицируются номером паспорта, имеются телефоны. При этом в базе данных предполагается хранить информацию об одном номере телефона, например домашнего, для каждого человека. Существуют люди, у которых вообще нет телефона. Кроме того, возможна ситуация, когда один номер телефона соответствует разным людям, например, если они проживают в одной квартире. Тогда в качестве `СУЩНОСТЬ1` выступает человек, а в качестве `СУЩНОСТЬ2` — телефон. На рис. 1.28 показана диаграмма ER-экземпляров для приведенного описания предметной области.

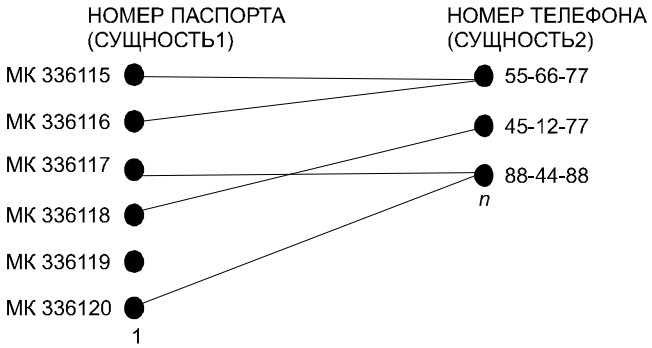


Рис. 1.28. Диаграмма ER-экземпляров для связи из правила 6

Мы уже знаем, что любую бинарную связь можно представить в виде трех отношений, по одному на каждую сущность с ключами, которые совпадают с ключом соответствующей сущности, плюс отношение связи, которое содержит в себе ключи обеих сущностей. Все правила генерации предварительных отношений получены из попыток уменьшения количества отношений в тех случаях, когда возможно избежать появления `NULL`-значений, и имеются места для неизбыточного размещения атрибутов, функционально зависящих от ключей сущностей. Кроме того, всякий раз необходимо определять ключ отношения связи. Проделаем эти действия для последнего примера.

Распишем данные из диаграммы ER-экземпляров по трем предварительным отношениям, как это сделано на рис. 1.29. Видно, что при наличии трех отношений все данные корректно размещены, а NULL-значения отсутствуют. Также видно, что уменьшить количество отношений для данного случая невозможно, т. к. во всех трех таблицах, соответствующих сущностям, разное число строк.

Номер паспорта	Номер телефона
МК 336115	55-66-77
МК 336116	55-66-77
МК 336117	88-44-88
МК 336118	45-12-77
МК 336120	88-44-88

Номер паспорта
МК 336115
МК 336116
МК 336117
МК 336118
МК 336119
МК 336120

Номер телефона
55-66-77
88-44-88
45-12-77

Рис. 1.29. Обоснование правила 6

1.8.3. Связи более высокого порядка, чем бинарные

Начнем с рассмотрения примера описания предметной области. Допустим, что имеются преподаватели, которые ведут какие-либо предметы. При этом речь идет только о чтении лекций для потока студентов. Из предыдущего утверждения видно наличие трех сущностей — ПРЕДМЕТ, ПРЕПОДАВАТЕЛЬ и ПОТОК. Пусть лекции по одному и тому же предмету могут читать несколько преподавателей. Необходимо знать, для какого потока студентов данный преподаватель читает конкретный предмет. Разрешим одному преподавателю читать лекции по нескольким предметам, т. е. сущность ПРЕДМЕТ становится *m*-связной. Ограничиться в этой ситуации бинарными связями невозможно, т. к. очевидна *необходимость связать одновременно три сущности* при наличии как минимум одной связи типа "много-ко-многим". Пусть сущность ПРЕДМЕТ идентифицируется его названием, сущность ПРЕПОДАВАТЕЛЬ — кодом преподавателя, а сущность ПОТОК идентифицируется номером потока. Диаграмма ER-экземпляров для этого случая представлена на рис. 1.30.

Связь, когда в нее вступают три сущности, носит название *триплета*. ER-диаграмма для связи-триплета приведена на рис. 1.31.

Дальнейшее увеличение количества сущностей, вступающих в связи, ведет к связям, которые носят название *многомерных*. Многомерные связи в общем случае никак специально не классифицируют. То есть если в связь вступили

K сущностей ($K > 2$), то для всевозможных комбинаций параметров связи пользуются следующим правилом.

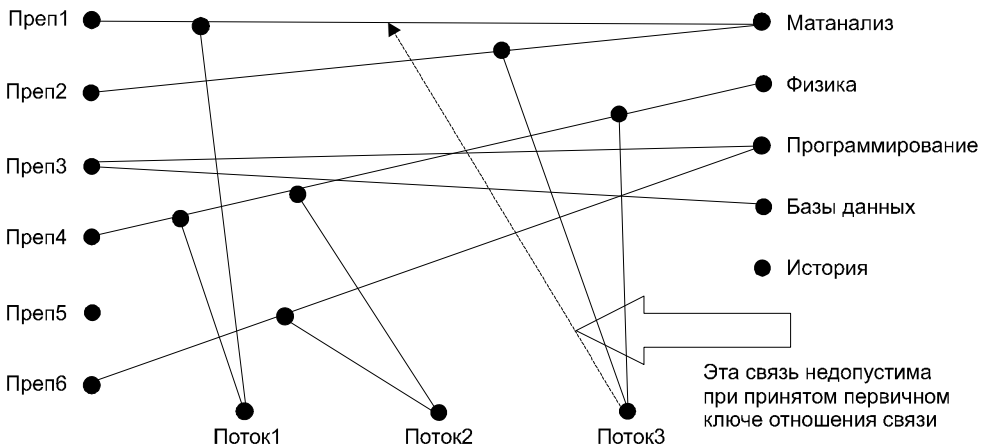


Рис. 1.30. Диаграмма ER-экземпляров для связи-триплета

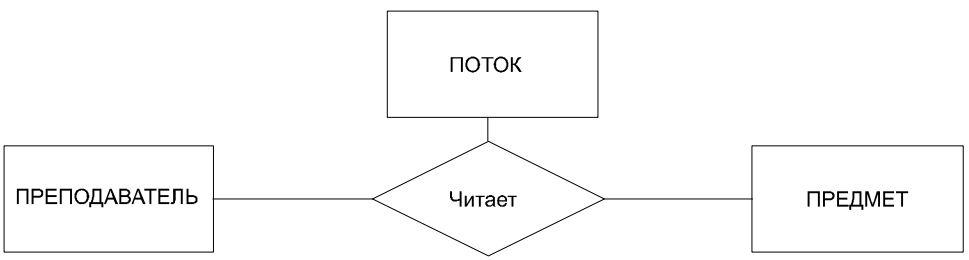


Рис. 1.31. ER-диаграмма связи-триплета

Правило 7. В случае K-сторонней связи необходимо сгенерировать $K + 1$ предварительное отношение — по одному отношению на каждую из K сущностей с ключами, совпадающими с ключами сущностей, и одно отношение для связи, в которое необходимо поместить все K ключей связываемых сущностей. Классы принадлежности сущностей для K-сторонней связи можно не рассматривать. Определение ключа отношения связи — задача неформальная, как правило, ее решают после размещения дополнительных атрибутов в предварительные отношения исходя из анализа описания предметной области.

В общем случае, если на ER-диаграмме представлена K-значная связь между набором сущностей `СУЩНОСТЬ1`, `СУЩНОСТЬ2`, ... `СУЩНОСТЬ_K` соответственно с ключами `Ключ1`, ... `Ключ_K`, то необходимы следующие предварительные отношения:

- R1(Ключ1 ...) — соответствует сущности СУЩНОСТЬ1;
- R2(Ключ2 ...) — соответствует сущности СУЩНОСТЬ2;
- и т. д;
- R_K(Ключ_K ...) — соответствует сущности СУЩНОСТЬ_K;
- R_K+1(Ключ1, Ключ2, ... Ключ_K) — является отношением связи.

Проиллюстрируем способ изображения логической модели реляционной базы данных для *K*-значной связи. В этом случае в Erwin нужно создать все *K* сущностей, соответствующие предварительным отношениям для сущностей, задав для них первичные ключи. После этого необходимо создать $K + 1$-ю сущность, соответствующую отношению связи. Далее необходимо реализовать *K*-значную связь через стандартные бинарные связи, т. е. с помощью ограничений внешнего ключа, и специально введенную *ассоциативную сущность*, называемую еще отношением связи. Для этого ключи каждой из сущностей СУЩНОСТЬ1 — СУЩНОСТЬ_K должны мигрировать в отношение связи R_K+1 с помощью либо идентифицирующей, либо неидентифицирующей связи. При этом идентифицирующую связь выбираем, если ключ соответствующей сущности входит в ключ отношения связи, в противном случае пользуемся неидентифицирующей связью. На рис. 1.32 приведена реализация этого правила для $K = 5$.

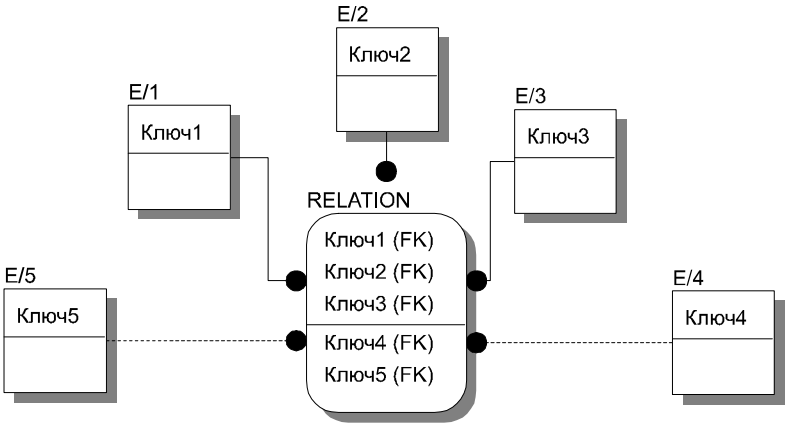


Рис. 1.32. Реализация *K*-значной связи в Erwin

Для рис. 1.32 отношение связи такое: RELATION(Ключ1, Ключ2, Ключ3, Ключ4, Ключ5). Ключ отношения связи выбран только для того, чтобы показать различия в способе миграции ключевых и неключевых атрибутов — ведь само правило ничего не говорит о том, как выбирается этот ключ, и без анализа бизнес-логики его не определить. Запрет или разрешение NULL-значений в

атрибутах, которые не входят в первичный ключ отношения связи, также устанавливается согласно бизнес-логике предметной области.

Применим правило 7 к нашему примеру про преподавателей, потоки и курсы. Предварительные отношения в этом случае будут такими:

- R1(Преподаватель, ...);
- R2(Предмет,...);
- R3(Поток,...);
- R4(Предмет, Поток, Преподаватель, ...).

В отношении связи R4 ключом выбрана пара атрибутов (Предмет, Поток), т. к. не бывает случаев, когда на одном потоке какой-то предмет читается два раза одновременно. Используя рис. 1.30, этот факт можно пояснить следующим образом: невозможно наличие связи, показанной пунктирной стрелкой от значения экземпляра "Поток3" сущности ПОТОК на связь между экземплярами сущностей ПРЕПОДАВАТЕЛЬ и ПРЕДМЕТ "Преп1" — "Матанализ", поскольку уже есть связь "Поток3" — "Преп2" — "Матанализ".

1.8.4. Ролевые связи

Ролевые связи необходимы в ситуациях, когда:

- экземпляры одной и той же сущности вступают в связи между собой;
- в зависимости от значения одного из атрибутов сущности по-разному определяется само множество других ее атрибутов;
- в зависимости от значения одного из атрибутов сущности она по-разному вступает в связи с другими сущностями.

Классическим примером ролевой связи первого типа считается случай, когда в каком-либо множестве людей с одинаковым набором сведений одни субъекты являются руководителями для других. Так, развивая пример о преподавателях, предположим, что о них нужно хранить некоторые сведения (номер паспорта, номер трудовой книжки и пр.). Некоторые из преподавателей являются заведующими кафедрой и, следовательно, руководителями для других преподавателей. Графически это показано на рис. 1.33. Ключом сущности ПРЕПОДАВАТЕЛЬ является табельный номер. Объекты данной сущности могут играть роль либо заведующего кафедрой, либо обычного преподавателя. Два ролевых набора ЗАВ_КАФЕДРОЙ и ОБЫЧН_ПРЕПОД соединяются связью РУКОВОДИТ.

На ER-диаграмме ролевая связь изображается с помощью стрелок, отходящих от сущности, порождающей роли, к ролевым сущностям. Ключи сущности, порождающей роли, и ключи ролевых сущностей принадлежат одному домену. Кроме того, множество значений ключа каждого из ролевых элемен-

тов является подмножеством множества значений ключа сущности, порождающей роли. Общий случай ER-диаграммы для ролевой связи приведен на рис. 1.34.

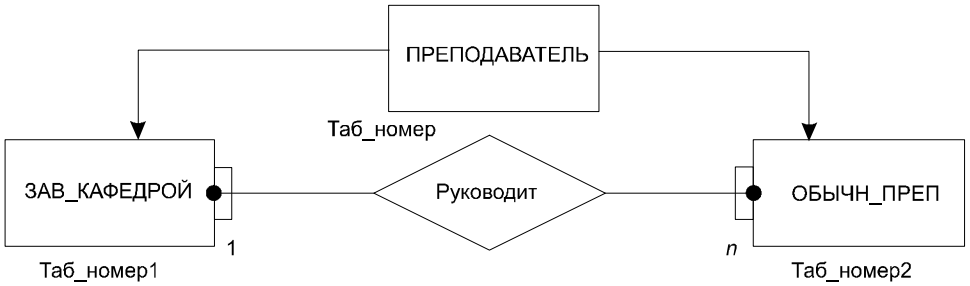


Рис. 1.33. Пример ER-диаграммы для ролевой связи

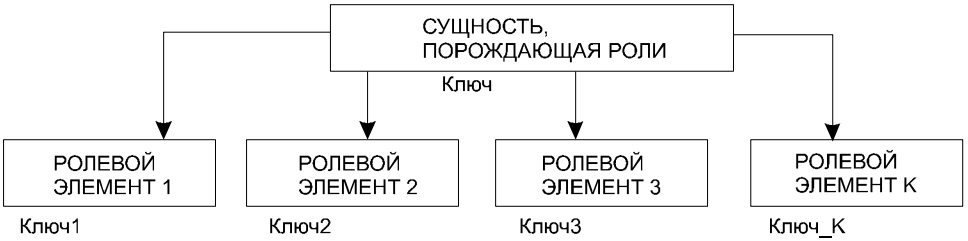


Рис. 1.34. Общий вид ER-диаграммы ролевой связи

Правило 8. Исходная сущность, порождающая роли, является источником одного предварительного отношения, причем ключ этой сущности служит ключом отношения. Ролевые элементы порождают еще по одному отношению с теми же ключами, что и исходная сущность. Связи, соединяющие ролевые отношения, генерируют столько отношений, сколько требуется для данного вида связи по соответствующим правилам. Отсюда следует, что для ER-диаграммы, приведенной на рис. 1.33, необходимы следующие предварительные отношения:

- `R(Ключ, ...)` — соответствует сущности, порождающей роли;
- `R1(Ключ1, ...)` — соответствует сущности для первого ролевого элемента;
- `R2(Ключ2 ...)` — соответствует сущности для второго ролевого элемента;
- и т. д.;
- `R_K(Ключ_K, ...)` — соответствует сущности для *K*-го ролевого элемента.

Называть ли по-разному ключевые атрибуты сущности, порождающей роли, и ролевых сущностей, зависит от решения проектировщика и особенностей СУБД.

Иногда полезно в отношение, генерирующее роли, добавить атрибут, называемый *дискриминатором*, по которому происходит разделение на роли. Это означает, что для каждого ролевого элемента значение дискриминатора фиксировано. Решение о создании поля для хранения значений атрибута-дискриминатора в отношении для сущности, порождающей роли, также зависит от решения проектировщика.

Применив правило 8 к ER-диаграмме, изображенной на рис. 1.33, получим следующие предварительные отношения:

- R1(Таб_номер, ...) — соответствует сущности ПРЕПОДАВАТЕЛЬ, порождающей роли;
- R2(Таб_номер1, ...) — соответствует ролевому элементу ЗАВ_КАФЕДРОЙ;
- R3(Таб_номер2, Таб_номер1,...) — соответствует ролевому элементу ОБЫЧН_ПРЕПОД.

Отношение R3, порождаемое ролью ОБЫЧН_ПРЕПОД, включило в себя атрибут ЗАВ_КАФЕДРОЙ для связи типа 1 : *n* РУКОВОДИТ, согласно правилу 2. Если бы связь РУКОВОДИТ имела тип *m* : *n*, то понадобилось бы четыре предварительных отношения.

В Erwin для реализации ролевых связей служит связь *"супертип-подтип" (см. разд. 1.9)*. Различают *полную* и *неполную* связи "супертип-подтип". В полной связи каждое значение ключа сущности, порождающей роли, должно быть занесено в какой-нибудь (но только в один) ролевой элемент. В неполной связи "супертип-подтип" возможно, что какие-либо из значений ключа сущности, порождающей роли, отсутствуют в ролевых элементах. Для пояснения различия между полной и неполной связями "супертип-подтип" на рис. 1.35 показана ролевая связь ER-диаграммы, изображенной на рис. 1.34, с помощью полной связи "супертип-подтип", а на рис. 1.36 — с помощью неполной. Из сравнения рисунков видно, что у символа полной связи "супертип-подтип" имеется вторая горизонтальная линия. В обоих случаях показан также *атрибут-дискриминатор*.

Замечание

В *разд. 1.9* показано, что категориальная связь однозначно заменяется обычными бинарными связями с применением такого средства, как ролевые или функциональные имена атрибутов, реализующих связь. Этим приемом также можно пользоваться для отображения ролевых связей.

В дальнейшем будут рассмотрены различные примеры моделирования реляционной базы данных. В рамках этих примеров будут приведены различные описания гипотетических предметных областей и полученные для них проектные решения. При этом для изображения моделей будет использоваться

CASE-средство Erwin. Таким образом, мы изобразим логическую модель базы данных в стандартизованной нотации, подразумевая при этом, что вся предварительная деятельность, связанная в основном с концептуальным моделированием и работой с ER-диаграммами, выполнялась при необходимости по правилам, сформулированным выше.

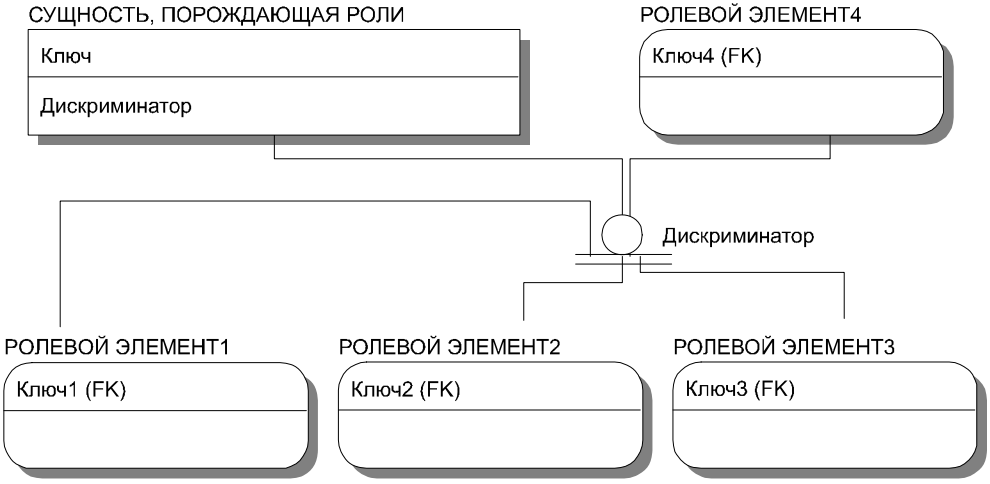


Рис. 1.35. Полная связь "супертип-подтип"

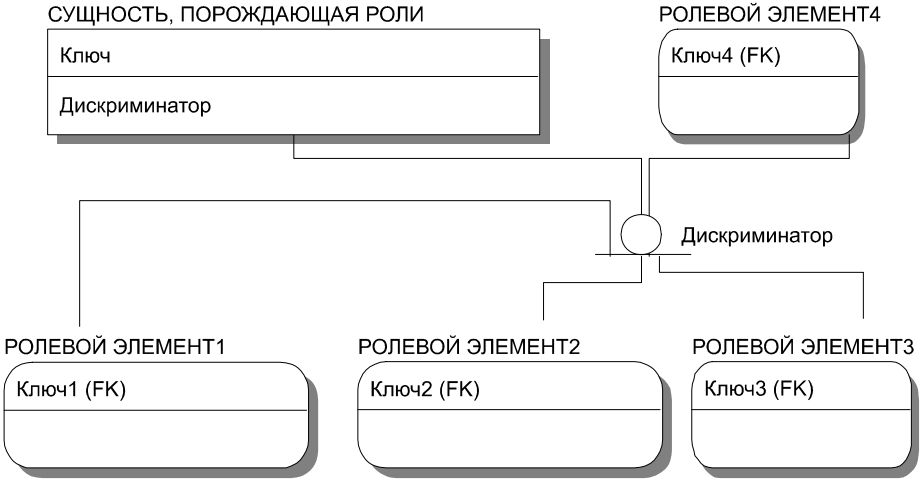


Рис. 1.36. Неполная связь "супертип-подтип"

1.8.5. Некоторые специальные аспекты логического моделирования

Правила выбора первичных ключей сущностей

Первичным ключом сущности может быть назначен один из ее потенциальных ключей. При этом следует принимать во внимание то, что на физическом уровне первичный ключ обычно реализуется как столбец (или группа столбцов) с запрещенными `NULL`-значениями, что диктуется требованием целостности сущностей, по которому создается уникальный кластерный индекс. Остальные ключи сущности на физическом уровне можно при необходимости поддерживать с помощью уникальных некластерных индексов. Далее подробно разбираются все аспекты использования тех или иных типов индексов в СУБД. Правила назначения первичных ключей сущностей сформулированы, в первую очередь, исходя из принципов работы с индексами в реляционной СУБД.

- Первичный ключ должен *однозначно идентифицировать* любой экземпляр сущности.
- По возможности первичный ключ должен быть *наиболее компактным* из всех потенциальных ключей, наилучший тип данных для первичного ключа — целочисленный.
- Первичный ключ может быть составным, но увеличение количества столбцов, входящих в него, противоречит требованию компактности. Требование компактности также не удастся выполнить, если, например, в качестве первичного ключа выбрать чересчур широкий столбец строкового типа данных.
- Значения первичного ключа *не должны подвергаться частым модификациям*. Идеально, если бизнес-логика предметной области такова, что эти значения вообще не предполагается изменять.
- *Правила модификации первичного ключа должны контролироваться внутренней бизнес-логикой* предметной области, а не решениями, которые принимаются над ней. Например, в базе данных, разрабатываемой для нужд кадрового учета какого-либо предприятия, для сущности `СОТРУДНИК` не стоит выбирать в качестве первичного ключа серию и номер паспорта сотрудника. Хотя эти данные в принципе и обладают свойствами обязательности и уникальности, но их изменение может быть инициировано сотрудником, а не администрацией предприятия.
- Если среди информации, собранной о сущности, не удается выделить данные, которые удовлетворяют перечисленным выше требованиям, то рекомендуется рассмотреть возможность создания *суррогатного первичного*

ключа, который, не неся никакой семантической нагрузки, просто служит идентификатором конкретного экземпляра сущности.

Обычно в качестве суррогатного первичного ключа выбираются всевозможные коды или идентификаторы. Суррогатный ключ чаще всего скрыт на внешнем уровне моделирования реляционной базы данных, если вести речь об архитектуре ANSI-SPARK *(см. главу 2)*, т. е. суррогатный первичный ключ скрыт от конечного пользователя. Классическим примером суррогатного первичного ключа является табельный номер сотрудника. Интересно, что при использовании табельный номер часто переходит на внешний уровень модели, т. е. зачастую он отражается в выходных формах и известен сотрудникам.

Связи без права передачи

Обычно экземпляр сущности в какой-то момент времени связан с одним экземпляром другой сущности, а потом эта связь может быть разорвана, после чего ее возможно установить между другими экземплярами сущностей. Примером такой ситуации служит переход сотрудника в другой отдел или студента в другую группу. Если подобное недопустимо, то такую связь называют *связью без права передачи*.

Простейшим примером связи без права передачи является связь между сущностями `ЧЕЛОВЕК` и `ПАСПОРТ`, поскольку паспорт нельзя передать другому человеку, а любая попытка смены паспорта ведет к уничтожению старого и выдаче нового.

Другим вариантом связи без права передачи является связь между сущностями в том случае, когда нужно сохранять сведения о времени создания и разрыва связи между конкретными экземплярами сущности. Так, если для связи между сущностями `СТУДЕНТ` и `ГРУППА` сохранять информацию о моменте времени, когда студент сменил группу, то такую связь также можно трактовать как связь без права передачи.

При моделировании базы данных существует ряд правил, касающихся права передачи связи:

- внешний ключ, реализующий связь с правом передачи, не следует делать частью первичного ключа, т. к. такой ключ будет нестабильным при передаче связи;
- нет смысла отслеживать историю изменений данных, объединенных связью без права передачи, поскольку каждое изменение сводится к уничтожению старого экземпляра и созданию нового;
- две сущности, объединенные связью "один-к-одному" с правом передачи, никогда не объединяются в одну.

Особенности работы со временем

Достаточно часто при проектировании базы данных сталкиваются с проблемой *моделирования времени*, когда нужно отслеживать динамику изменения тех или иных характеристик. Можно выделить два подхода к решению этой проблемы. Проиллюстрируем эту ситуацию на типичном примере описания предметной области, связанной с банковским учетом. Предположим, что существуют счета, на которых хранятся деньги, и с этими счетами можно выполнять операции определенного типа по изменению счета, например пополнение счета или снятие денег со счета. Сущность `СЧЕТ` идентифицируется первичным ключом `Счет_Ид`, операции идентифицируются ключом `Операция_Ид`. Сущность `СЧЕТ` имеет производный атрибут `Остаток_на_Счете`, который содержит данные о наличии денежных средств на счете после выполнения последней по времени операции со счетом, и значение этого атрибута должно быть согласовано с данными об изменениях счета. Части проектных решений, связанные с отслеживанием событий и времени их осуществления, отличаются в зависимости от выбранной схемы реализации этой части проектного решения. Возможны следующие схемы.

- *Отслеживание истории изменений* (audit trail), когда в базу данных заносятся изменения и их даты. Так как в описанном случае изменениями являются все факты снятия денег с банковского счета, то пример проектного решения для этой схемы может быть таким, как изображено на рис. 1.37.

Рис. 1.37. Проектное решение отслеживания истории изменений

Для проектного решения, приведенного на рис. 1.37 характерно, что для вычисления остатка денежных средств на счете нужно сконструировать формулу, в которой используется атрибут `Изменение_Счета`, из которого определяется, на сколько в результате выполнения операции изменился счет. Далее по значению атрибута `Тип_операции` определяется способ расчета изменения значения атрибута `Остаток_на_Счете`. Эти вычисления можно сделать в любой момент времени, и нет необходимости обязательно выполнять их в рамках транзакции по изменению счета.

При этой схеме реализации события могут составлять иерархию, допускается также ввести понятие простейших событий и сложных событий, состоящих из простейших.

Указанным способом отслеживаются и изменения числовых характеристик. Для нечисловых атрибутов можно сохранять значения перед изменением. Получается в некоторой степени не история изменений, а история данных, что подводит нас ко второму подходу.

- *Технология мгновенных снимков данных* (data snapshots). В этом случае записываются не изменения, а состояние данных в отдельные моменты времени, например состояние банковского счета на начало каждого месяца или после отдельных событий.

  Вид простейшей схемы (когда сохраняется состояние данных в заданные моменты времени) представлен на рис. 1.38.

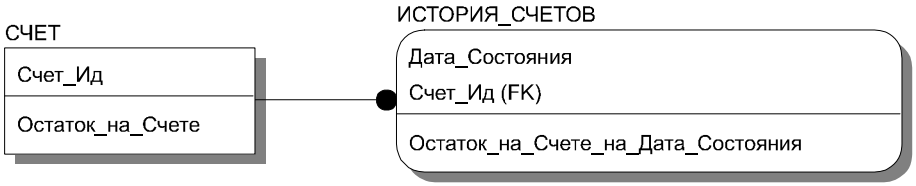


Рис. 1.38. Отслеживание истории изменений с помощью мгновенных снимков

  Следует отметить, что при использовании технологии мгновенных снимков атрибут `Остаток_на_Счете` обязательно должен перевычисляться в рамках транзакции по изменению счета. Более того, если ограничиться только двумя введенными на рис. 1.38 сущностями, то этот атрибут становится базовым, т. к. его значение нельзя получить больше ниоткуда, кроме как из таблицы `СЧЕТ`. Значение атрибута `Остаток_на_Счете_после_События` в этой ситуации может заноситься в базу данных, например, таким способом: на конец каждого месяца из таблицы `СЧЕТ` значения атрибута `Остаток_на_Счете` переносятся в таблицу `ИСТОРИЯ_СЧЕТОВ`. При этом идентификатор даты (месяца) выполнения этой операции добавляется в качестве второго ключевого атрибута в таблицу `ИСТОРИЯ_СЧЕТОВ`. Очевидно, что при подобном подходе часть информации об операциях со счетами будет потеряна, т. к. в базе данных сохранятся только месячные итоги изменения счетов, а не результаты каждой транзакции.

Если же состояние предметной области сохраняется после отдельных событий, то проектное решение получается очень похожим на то, что получилось при схеме отслеживания изменений. В этом случае опять необходимо вводить сущность `СОБЫТИЕ`, а проектное решение может выглядеть так, как показано на рис. 1.39.

Сравнение проектных решений, приведенных на рис. 1.38 и 1.39, позволяет утверждать, что они очень похожи. Различие же заключается в том, что во втором случае в базе данных сохраняется значение остатка на счете (атрибут

Остаток_на_Счете_после_События) после каждой транзакции по изменению счета, которое должно быть вычислено в рамках этой транзакции с использованием старого значения атрибута Остаток_на_Счете. Способ расчета этого значения не сохраняется в базе данных (нет атрибута, аналогичного атрибуту Тип_операции из первого случая). Поэтому наиболее разумно сразу же при выполнении транзакции по изменению счета заменить значение атрибута Остаток_на_Счете на вновь вычисленное. Заметим, что в последнем случае в таблице СЧЕТ хранится по сути последнее по времени занесения значение атрибута Остаток_на_Счете_после_События из таблицы СЧЕТ_ПОСЛЕ_СОБЫТИЯ. Если есть способ оперативно находить эти значения и затраты времени на поиск существенно не зависят от размера таблицы СЧЕТ_ПОСЛЕ_СОБЫТИЯ, то возможно, что атрибут Остаток_на_Счете вообще не нужен в таблице СЧЕТ.

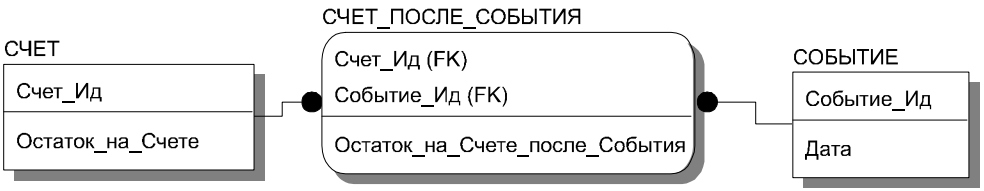


Рис. 1.39. Проектное решение с введением сущности СОБЫТИЕ

При выборе любого метода отслеживания временных изменений можно разбивать сущности на части, зависящие и не зависящие от времени, и отдельно связывать их с событиями.

1.9. Обзор возможностей CASE-средства Erwin

В данном разделе приводятся краткие сведения о работе с таким CASE-средством проектирования реляционной базы данных, как программа Erwin 3.5.2 фирмы PLATINUM. Выбор именно этого CASE-инструмента обусловлен, в первую очередь, его простотой. При этом внимание акцентируется на описании приемов, используемых на самых ранних стадиях проектирования, а именно на работе с концептуальной и логической моделью реляционной базы данных. Поэтому то обстоятельство, что выбрана не самая последняя версия рассматриваемого приложения, не является слишком существенным, потому что в первую очередь будет изучаться вопрос о том, *что* же должно быть изображено в модели реляционной базы данных в тех или иных ситуациях, в меньшей степени подчеркивая то, *как* это сделать.

Разумеется, здесь не могут быть описаны все возможности указанной программы. Практически не рассматриваются вопросы инжиниринга и синхро-

низации физической модели базы данных с целевой СУБД, способы генерации отчетов по моделям, работа с хранилищем моделей и многое другое. Представлено только то, что позволит начать непосредственную практическую работу с программой Erwin как с унифицированным средством моделирования реляционной базы данных.

Программа Erwin имеет два уровня работы с моделью, называемые логическим (**Logical**) и физическим (**Physical**). Как уже говорилось, логический уровень отвечает и концептуальной, и собственно логической модели реляционной базы данных. Переключение между логическим и физическим уровнями модели осуществляется автоматически, путем выбора уровня модели из раскрывающегося списка на главной панели инструментов. При переходе к физической модели необходимо указать целевую СУБД. В нашем случае целевой СУБД будет являться SQL-сервер. Первый раз при запуске программы и переходе в режим физической модели будет задан вопрос о том, какую СУБД считать целевой. Вообще же целевую СУБД можно задать в опции **Server/Target Server** основного меню программы Erwin. При выборе этой опции меню выводится окно с перечнем СУБД, соединение с которыми поддерживается программой Erwin. В нашем случае в этом окне выбран переключатель, соответствующий SQL-серверу, в этом случае все свойства физической модели будут отвечать особенностям SQL-сервера.

Очень важным моментом работы с CASE-средством Erwin является тот факт, что при переключении в режим физической модели эта модель автоматически создается по определенным правилам. Физическая модель, разумеется, в первую очередь отображается в принятой графической нотации. В зависимости от типа связи между сущностями переход от логической модели к физической осуществляется по-разному. В тех случаях, когда эти различия существенны, они будут оговорены особо. Существует возможность указания того, что отдельные объекты следует рассматривать только в контексте логической модели, не включая их в модель физическую, и, следовательно, не передавая в целевую СУБД.

Кроме графического представления физической модели, при переходе к ней генерируется программный код, содержащий операторы создания всех объектов, внесенных в модель базы данных. Этот программный код или сценарий генерации объектов можно просматривать в рамках программы Erwin, копировать в буфер обмена и таким образом переносить и выполнять в контексте целевой СУБД. Для SQL-сервера сценарий генерации объектов целевой СУБД представляет собой набор DDL-операторов на языке T-SQL. Более того, находясь в контексте программы Erwin можно подсоединиться к целевой СУБД и выполнить сценарий генерации объектов непосредственно из Erwin.

1.9.1. Интерфейс Erwin

Интерфейс Erwin выполнен в стиле Windows-приложений, достаточно прост и интуитивно понятен. Как обычно, имеется основное меню пользователя и панели инструментов. Мы не будем описывать все опции меню и все кнопки панелей, а сосредоточим внимание на тех моментах, которые важны именно для моделирования базы данных.

Чтобы начать рисовать модель базы данных, нужно первым делом разобраться в панели инструментов, называемой **ERwin Toolbox**. Эта панель включается/отключается комбинацией клавиш <Ctrl>+<T>. Она содержит восемь кнопок, которые несколько отличаются для логической и физической модели базы данных. Панель инструментов **ERwin Toolbox** приведена на рис. 1.40.

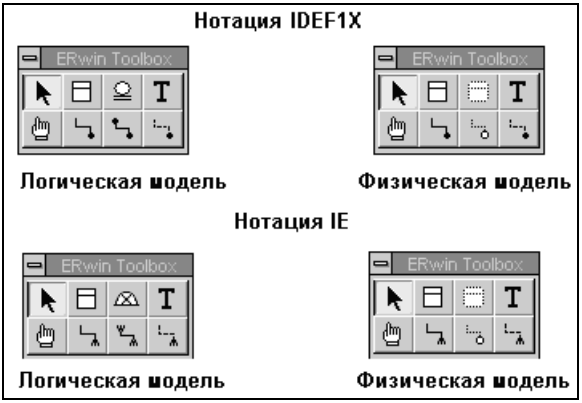


Рис. 1.40. Панель инструментов **ERwin Toolbox**

На *логическом уровне* модели панель инструментов имеет следующие кнопки:

- кнопка указателя (режим мыши — в этом режиме можно установить фокус на каком-либо объекте модели);
- кнопка сущности (для внесения сущности нужно щелкнуть левой кнопкой мыши на кнопке внесения сущности и один раз по свободному пространству на модели; повторный щелчок приведет к внесению в модель еще одной новой сущности; для редактирования сущностей или других объектов модели необходимо перейти в режим указателя);
- кнопка категории — для создания категориальных связей, называемых также связями "супертип-подтип";
- кнопка внесения текстового блока (с ее помощью можно внести текстовый комментарий в любую часть графической модели);

- кнопка перенесения атрибутов внутри сущностей, между ними атрибуты могут быть перемещены способом "переместить и оставить" (drag&drop);
- кнопка создания идентифицирующей связи ;
- кнопка создания связи "много-ко-многим" ;
- кнопка создания неидентифицирующей связи .

На *физическом уровне* панель инструментов имеет следующие отличия:

- вместо кнопки категории — кнопку для создания представлений ;
- вместо кнопки связи "много-ко-многим" — кнопку связей представлений .

Для создания моделей данных в Erwin можно использовать две нотации: IDEF1X и IE. Переключение между нотациями выполняется на вкладке **Methodology** меню **Option/Preferences**.

Уровни отображения модели

Erwin имеет несколько уровней отображения диаграммы (модели): уровень сущностей, уровень атрибутов, уровень определений, уровень первичных ключей и уровень пиктограмм. Переключиться между первыми тремя уровнями можно с использованием кнопок панели инструментов. Переключение на другие уровни отображения осуществляется при помощи контекстного меню, которое появляется, если щелкнуть кнопкой мыши по любому месту диаграммы, не занятому объектами модели. В контекстном меню следует выбрать пункт **Display Level** и затем необходимый уровень отображения, как это показано на рис. 1.41.

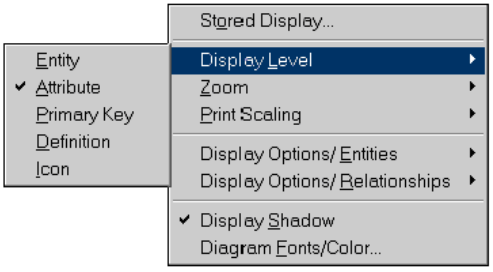


Рис. 1.41. Контекстное меню свободного места диаграммы

Erwin позволяет связать с сущностью большую и малую пиктограммы. При переключении на уровень пиктограмм отображается большая пиктограмма. Для отображения малой пиктограммы следует выбрать в контекстном меню

пункт **Display Options/Entities** и в каскадном меню включить опцию **Entity Icon**. Малая пиктограмма будет показана слева от имени сущности на всех уровнях отображения модели.

Установка цвета и шрифта

Установить шрифт и цвет объектов в Erwin можно несколькими способами. Для установки цвета и шрифта объекта служит панель инструментов **Font and Color Toolbar** (рис. 1.42), которая располагается под основной панелью.

Рис. 1.42. Панель управления шрифтами

Для редактирования шрифта и цвета конкретного объекта следует, щелкнув правой кнопкой мыши по сущности или связи и выбрав из всплывающего меню пункт **Object Font/Color**, вызвать диалоговое окно **Font/Color Editor**, в котором определяются имя, описание и комментарии сущности. Диалоговое окно **Font/Color Editor** имеет три вкладки, на которых можно выбрать шрифт и установить его размер, стиль и цвет (вкладка **Text**), установить цвет заливки (вкладка **Fill**, только для сущностей) и цвет линий (вкладка **Entity Outline**, только для сущностей).

Имеется возможность изменить шрифт и цвет для всех объектов модели или для какой-либо отдельной категории объектов. Для этого служит диалоговое окно **All Default Font/Color Editor** (пункт меню **Option/Default Font/Color**). Каждая вкладка диалогового окна позволяет редактировать шрифт и цвет для определенной категории объектов:

- **All Fonts** — все объекты модели;
- **Entity Name** — имена сущностей и таблиц;
- **Entity Definition** — определение сущностей и таблиц (показываются на уровне определений);
- **Relationship** — связи включая имя и обозначение мощности;
- **Subtype** — иерархия категорий включая дискриминатор категории;
- **Text Block Text** — текстовые блоки;
- **Page Number** — номер страниц при печати диаграммы;
- **Owned Entity Attributes** — атрибуты и колонки за исключением атрибутов и колонок внешних ключей;
- **Foreign Key** — атрибуты и колонки внешних ключей;
- **Background Color** — цвет фона диаграммы;

- **Entity Line** — линии, которыми прорисовываются сущности и таблицы;
- **Entity Fill** — заливка сущностей и таблиц;
- **Subtype Fill** — заливка символов, обозначающих категории.

Замечание

Шрифты, которые используются в Erwin, — это шрифты, установленные в операционной системе. При работе со шрифтами иногда бывает достаточно сложно настроить национальную кодировку, которая доступна не для всех шрифтов. Практический опыт работы показал, что для отображения русских имен сущностей, атрибутов и т. д. подходят шрифты в кодировке Unicode. В частности, все примеры для данной книги выполнены с помощью шрифта Arial Unicode MS.

Подмножества модели и сохраняемые отображения

При создании реальных моделей данных количество сущностей и атрибутов может исчисляться сотнями. Для более удобной работы с большими моделями в Erwin предусмотрены подмножества модели (**Subject Area**), в которые разрешается включить тематически общие сущности. В подмножество модели может входить произвольный набор сущностей, связей и текстовых комментариев. Для создания, удаления или редактирования подмножеств модели нужно вызвать диалоговое окно **Subject Area Editor** меню **Edit/Subject Area**. В этом диалоговом окне указываются имя подмножества модели и входящие в нее сущности. Переключение между подмножествами модели также осуществляется с помощью диалогового окна **Edit/Subject Area**. Все изменения, сделанные в любом подмножестве отображения модели, автоматически влияют на общую модель. Одна и та же сущность может входить в несколько подмножеств отображения модели.

При создании нового подмножества следует в диалоговом окне **Subject Area Editor** указать его имя и список входящих в него объектов. Для переключения между подмножествами используется кнопка на панели инструментов. При этом на диаграмме отображаются только те сущности и связи, которые входят в выбранное подмножество модели.

Существует основное подмножество, называемое **Main Subject Area**, которое всегда содержит все сущности. По умолчанию исходная модель получает имя **Main Subject Area**.

1.9.2. Сущности и атрибуты в Erwin

Разумеется, основными компонентами диаграммы Erwin являются сущности, атрибуты и связи. Каждая сущность представляет собой множество подобных индивидуальных объектов, называемых *экземплярами*. Каждый экземпляр индивидуален и должен отличаться от всех остальных экземпляров. Атрибут

выражает определенное свойство объекта. С точки зрения физической модели, сущности соответствует таблица, экземпляру сущности — строка в таблице, а атрибуту — колонка таблицы.

Для внесения сущности в логическую модель нужно щелкнуть кнопкой сущности на панели инструментов **ERwin Toolbox**, а затем щелкнуть на том месте на диаграмме, где необходимо расположить новую сущность. Щелкнув правой кнопкой мыши на сущности и выбрав из контекстного меню пункт **Entity Editor**, можно вызвать редактор атрибутов, в котором определяются имя, описание и комментарии сущности (рис. 1.43).

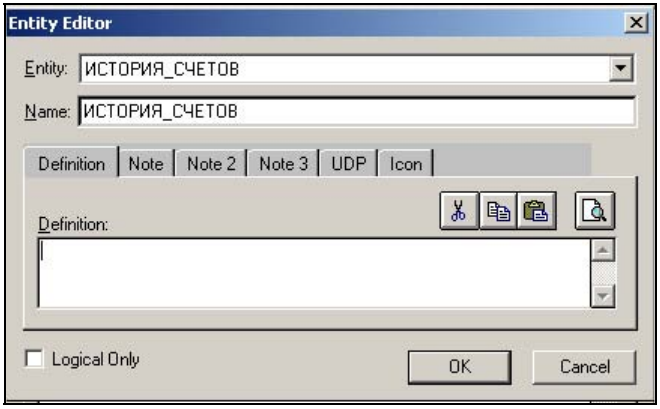


Рис. 1.43. Окно редактора сущности

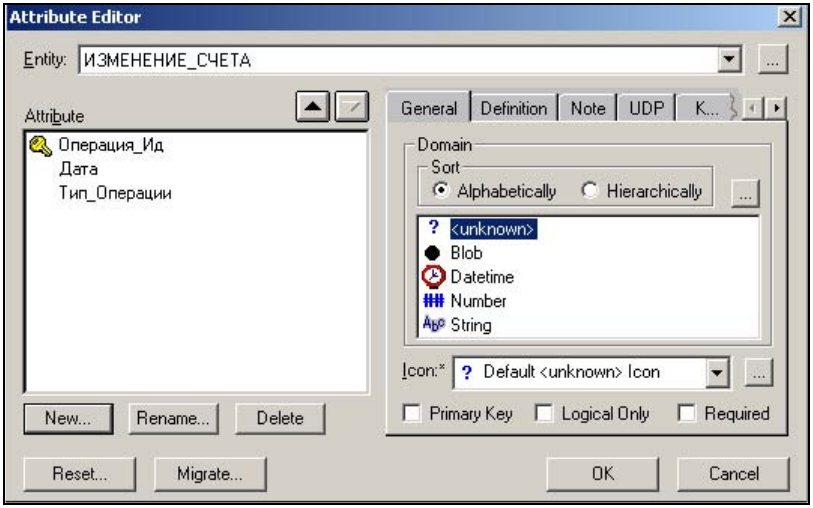


Рис. 1.44. Окно редактора атрибутов

Для описания атрибутов сущности следует, щелкнув правой кнопкой мыши на сущности, выбрать в появившемся меню пункт **Attribute Editor**. Откроется редактор атрибутов (рис. 1.44).

Работа с редактором атрибутов

Редактор атрибутов служит для задания и модификации атрибутов сущности. Если щелкнуть кнопкой **New**, то в появившемся диалоговом окне **New Attribute** можно указать имя атрибута, домен, к которому принадлежит данный атрибут (список **Domain**), а также имя соответствующего в физической модели атрибуту столбца. Домен атрибута будет использоваться при определении типа столбца на уровне физической модели. Каждому домену соответствует стандартная пиктограмма, однако можно импортировать и дополнительные изображения. Кнопка справа от списка выбора **Icon** вызывает диалоговое окно **Erwin Icon Editor**. Щелкнув кнопкой **Import**, можно добавить в список необходимую пиктограмму.

Очень важно дать атрибуту правильное имя. Атрибуты должны именоваться в единственном числе и иметь четкое смысловое значение. Согласно синтаксису IDEF1X имя атрибута должно быть уникально в рамках модели (а не только в рамках сущности!). По умолчанию при попытке внесения уже существующего имени атрибута Erwin переименовывает его. На практике такое переименование не всегда удобно, поэтому существует возможность отменить эту опцию. В диалоговом окне **Unique Name Option** (меню **Option/Unique Name**) можно задать способы именования атрибутов.

Атрибут или группа атрибутов, которые идентифицируют сущность, называются *первичным ключом*. Для атрибутов первичного ключа на вкладке **General** редактора атрибутов необходимо установить флажок выбора **Primary Key**. Если нужен составной первичный ключ, то нужно установить указанный флажок для всех атрибутов, входящих в его состав по очереди. Можно задавать первичный ключ и с помощью кнопки панели инструментов **Erwin ToolBox**, перемещая атрибуты способом "переместить и оставить" в ключевую область.

Флажок **Logical Only** устанавливается для атрибута, если он должен присутствовать только в логической модели. В этом случае атрибут будет отсутствовать в физической модели и не будет создан как столбец соответствующей таблицы. Если установлен флажок **Required**, то столбец на основании соответствующего атрибута будет создан с опцией `NOT NULL`.

Каждый атрибут должен быть описан с помощью текстового описания (вкладка **Definition** редактора атрибутов). При этом следует избегать циклических определений. Иногда определение атрибута легче дать через описание области значений. Например, оценка школьника — это число, принимающее

значения 2, 3, 4 и 5. Такая информация также может быть отображена на вкладке **Definition** редактора атрибутов.

Вкладка **Definition** на уровне физической модели преобразуется во вкладку **SQL Server**. Здесь можно задавать всевозможные свойства и ограничения, допустимые для столбцов таблиц в СУБД SQL-сервер, например, описать для столбца конкретный тип данных, отличный от того, который задается по умолчанию. Кроме того, можно задать для столбца свойство `IDENTITY`, указать значение по умолчанию (`DEFAULT`), а также условие проверки для значений, заносимых в столбец (`CHECK`).

Остальные вкладки редактора атрибутов будут описаны очень кратко. Так, вкладка **Note** позволяет добавлять замечания об одном или нескольких атрибутах сущности, которые не вошли в определения. Вкладка **UDP** служит для задания значений свойств, определяемых пользователем. Предварительно эти свойства должны быть внесены в диалоговое окно **User-Defined Property Editor** как свойства атрибутов.

1.9.3. Связи в Erwin

Как известно, при установлении связей между сущностями атрибуты первичного ключа родительской сущности мигрируют в качестве внешних ключей в дочернюю сущность. Кнопка **Migrate** диалогового окна **Attribute Editor** вызывает диалоговое окно **Migrate Attribute Property**, в котором можно задать свойства, сохраняемые при миграции.

Для создания связей между сущностями в Erwin используются соответствующие кнопки панели **ERwin Toolbox**. По умолчанию имя связи на диаграмме не показывается. Для отображения имени следует в контекстном меню, которое появляется, щелкнуть левой кнопкой мыши на любом месте диаграммы, не занятом объектами модели, выбрать пункт **Display Options/Relationship** и затем включить опцию **Verb Phrase**.

На логическом уровне отображения модели в Erwin можно установить идентифицирующую связь "один-ко-многим", связь "много-ко-многим" и неидентифицирующую связь "один-ко-многим" (соответственно это кнопки панели инструментов).

Кратко напомним суть установления связей между сущностями и терминологию, используемую при этом. Как уже было сказано, связи бывают идентифицирующие и неидентифицирующие. При установлении идентифицирующей связи атрибуты первичного ключа родительской сущности автоматически переносятся в состав первичного ключа дочерней сущности. Эта операция дополнения атрибутов дочерней сущности при создании связи называется миграцией атрибутов. В дочерней сущности новые атрибуты поме-

чаются как внешний ключ с помощью пометки *FK*. При генерации схемы базы атрибуты первичного ключа автоматически получат признак `NOT NULL`.

При установлении неидентифицирующей связи дочерняя сущность остается независимой, а атрибуты первичного ключа родительской сущности мигрируют в состав неключевых атрибутов родительской сущности.

В обеих нотациях модели, применяемых в Erwin, различают *зависимые* и *независимые* сущности. Тип сущности определяется ее связью с другими сущностями. Идентифицирующая связь устанавливается между независимой (родительский конец связи) и зависимой (дочерний конец связи) сущностями. Когда создается идентифицирующая связь, Erwin автоматически преобразует дочернюю сущность в зависимую. Зависимая сущность изображается прямоугольником со скругленными углами.

Для создания новой связи "один-ко-многим" следует:

1. Определить, какой тип связи (идентифицирующая кнопка) или (неидентифицирующая кнопка) будет использован в конкретном случае.
2. Установить курсор на нужной кнопке панели инструментов и нажать левую кнопку мыши.
3. Щелкнуть сначала по родительской, а затем по дочерней сущности.

Для создания новой связи "много-ко-многим" следует:

1. Установить курсор на нужной кнопке панели инструментов и нажать левую кнопку мыши.
2. Щелкнуть сначала по одной, а затем по другой сущности.
3. Если нужно разрешить связь "много-ко-многим" на уровне логической модели, то следует выделить эту связь, вызвать контекстное меню и нажать опцию **Resolve Many To Many**. При переходе на уровень физической модели связь "много-ко-многим" разрешается автоматически через третье отношение и две бинарные связи.

Для создания новой категориальной связи следует:

1. Установить курсор на кнопке в палитре инструментов и нажать левую кнопку мыши.
2. Щелкнуть сначала по родовому предку, а затем по потомку.
3. Для установления второй связи в иерархии категории следует сначала щелкнуть по символу категории, затем по второму потомку и т. д.

Для всех типов связи форму линии связи можно изменять. Для этого нужно захватить с помощью курсора мыши нужную линию связи и перенести ее на другое место.

Для редактирования свойств связи следует щелкнуть правой кнопкой мыши на связи и выбрать из контекстного меню пункт **Relationship Editor**, который активизирует редактор связей (рис. 1.45).

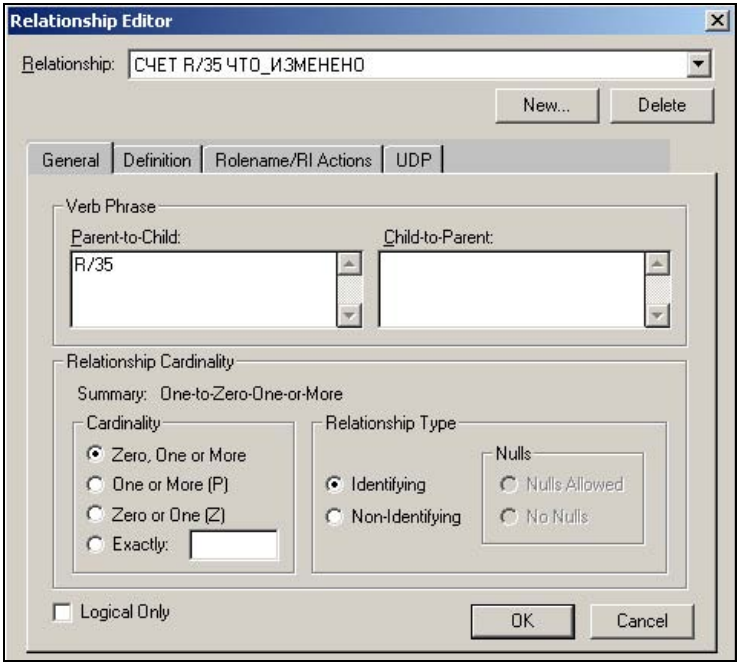


Рис. 1.45. Диалоговое окно **Relationship Editor** для вкладки **General**

Редактор связей, общие параметры бинарной связи

На вкладке **General** открывшегося диалогового окна редактора связи можно задать мощность, имя и тип связи.

Мощность связи (опция **Cardinality**) служит для задания числа, которое показывает, сколько экземпляров дочерней сущности может вступать в связь с одним экземпляром родительской сущности. Как известно, в общей теории реляционных баз данных для бинарных связей имеются такие типы мощности связи как "один-к-одному", "один-ко-многим" и "много-ко-многим". Нотации Erwin позволяют уточнять конкретные параметры связей. При этом некоторым образом учитывается и модальность связи. В Erwin имеется возможность задать четыре значения параметра **Cardinality** для бинарной связи:

1. **Zero, One or More** — это общий случай, когда одному экземпляру родительской сущности соответствуют ноль, один или много экземпляров дочерней сущности. Этот случай не помечается каким-либо символом; это

связи вида 1 : *n*, при этом допускается как обязательная, так и необязательная модальность связей (правила 2 и 6).

2. **One or More (P)** — этот случай помечается символом **P** и имеет место, когда одному экземпляру родительской сущности соответствует один или много экземпляров дочерней сущности; при этом подчеркивается обязательная модальность связи (правило 2).
3. **Zero or One (Z)** — этот случай помечается символом **Z** и имеет место, когда одному экземпляру родительской сущности соответствует ноль или один экземпляр дочерней сущности; таким образом указывается тип связи 1:1 и допускается как обязательная, так и необязательная модальность (правила 1, 4 и 5).
4. **Exactly n,** где цифрой **n** помечается случай точного соответствия, когда одному экземпляру родительской сущности соответствует заранее заданное число экземпляров дочерней сущности.

По умолчанию символ, обозначающий мощность связи, не показывается на диаграмме. Для его отображения следует в редакторе связей включить опцию **Cardinality** путем установки соответствующего переключателя.

Переключателем **RelationType** можно задать тип связи: идентифицирующая связь или неидентифицирующая. Для неидентифицирующей связи можно указать обязательность (переключатель опции **NULLs**). В случае обязательной связи (**No NULLs**) при генерации схемы базы данных атрибут внешнего ключа получит признак `NOT NULL` несмотря на то, что внешний ключ не войдет в состав первичного ключа дочерней сущности. В случае необязательной связи (**NULLs Allowed**) внешний ключ может принимать значение `NULL`. Необязательная связь помечается прозрачным ромбом со стороны родительской сущности в нотации IDEF1X.

Имя связи (**Verb Phrase**) — это фраза, характеризующая отношение между родительской и дочерней сущностями. Для связи "один-ко-многим" достаточно указать имя, определяющее отношение от родительской сущности к дочерней.

На вкладке **Definition** редактора связей можно дать более полное определение связи, чтобы в дальнейшем иметь возможность на него ссылаться.

Анализируя возможности опции **Cardinality** можно сделать вывод о том, что полного соответствия между классификацией бинарных связей по их параметрам и редактором связей Erwin не наблюдается. Более того, поскольку на уровне собственно СУБД нет никаких возможностей контролировать количество строк таблиц (экземпляров сущностей), вступивших или не вступивших в связь, то кардинальность и модальность связи можно задавать только на уровне логической модели. Таким образом, опция **Cardinality** нужна только

для уточнения логической модели, т. к. никаких изменений в SQL DDL-сценарий генерации объектов установка этой опции не вносит.

Замечание

Имеются целевые СУБД, например MS Access, в которых существует возможность использовать значение опции **Cardinality**. Дело в том, что для MS Access генерация объектов целевой СУБД по модели Erwin выполняется не с помощью стандартного SQL DDL-скрипта, а с помощью языка VBA. В этом случае имеется специальный объект, называемый *связь*, для которого можно указать параметр мощности связи, значение которого передается из опции **Cardinality**. В схеме данных СУБД MS Access такие связи отображаются, как показано на рис. 1.46; а в листинге 1.1 представлена часть кода, в которой создается объект "связь" и указывается мощность этой связи.

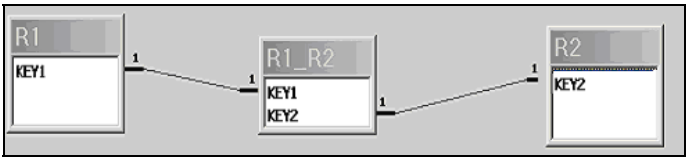


Рис. 1.46. Связи 1:1 в схеме данных СУБД MS Access

Листинг 1.1. Пример создания объекта "связь"

```
' CREATE RELATIONSHIP "R/3"
Set ERwinRelation = _'
ERwinDatabase.CreateRelation("R/3", "СУЩНОСТЬ2", "СУЩНОСТЬ1")
Set ERwinField = ERwinRelation.CreateField("Ключ2")
ERwinField.ForeignName = "Ключ2"
ERwinRelation.Fields.Append ERwinField
ERwinDatabase.Relations.Append ERwinRelation
' CREATE RELATIONSHIP "R/3"
Set ERwinRelation = _'
ERwinDatabase.CreateRelation("R/3", "СУЩНОСТЬ2", "СУЩНОСТЬ1")
Set ERwinField = ERwinRelation.CreateField("Ключ2")
ERwinField.ForeignName = "Ключ2"
ERwinRelation.Fields.Append ERwinField
ERwinRelation.Attributes = ERwinRelation.Attributes + DB_RELATIONUNIQUE
ERwinDatabase.Relations.Append ErwinRelation
```

Редактор связей, ролевые имена атрибутов в бинарной связи

На вкладке **Rolename/RI Actions** редактора связей можно задать *ролевые имена* (функциональные имена атрибутов) и правила ссылочной целостности.

Применение функциональных имен полезно в том случае, когда один и тот же атрибут должен мигрировать в дочернюю сущность два и более раза. То есть имеет место ситуация, когда два мигрирующих внешних ключа происходят из одного и того же отношения, но при этом представляют разные по смыслу связи, следовательно, интеграция внешнего ключа недопустима. С другой стороны, атрибут не может появиться дважды в одной сущности под одним именем, поэтому он обязательно должен получить функциональное имя. Необходимо различать эти атрибуты, которые имеют разный смысл, но ссылаются на одну и ту же сущность.

Другим примером использования функциональных имен являются *рекурсивные связи*. Иногда их называют термином "рыболовный крючок" (fish hook), когда одна и та же сущность является родительской и дочерней одновременно. При задании рекурсивной связи атрибут должен мигрировать в качестве внешнего ключа в состав неключевых атрибутов той же сущности.

В Erwin для задания ролевого имени нужно в окне редактирования связи переключиться на вкладку **Rolename/RI Actions**, выбрать в списке **Migrated Attribute**, и в поле **Rolename** задать ролевое имя. После этого внешний ключ будет по умолчанию отображаться своим ролевым именем, но если включить опцию **Rolename/Attribute** в подменю **Display Options/Entities**, то ролевые имена будут отображаться с добавлением имени первичного ключа, как это показано на рис. 1.47.

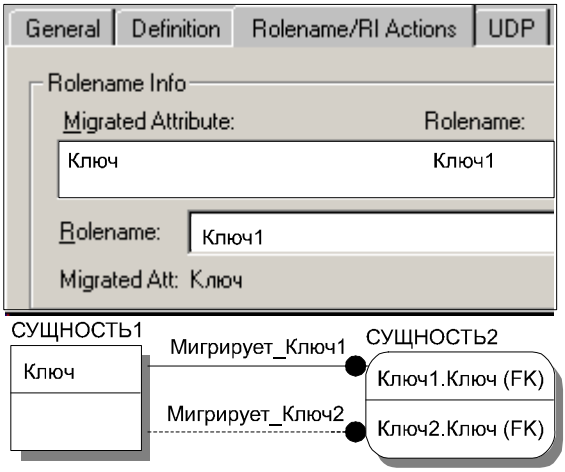


Рис. 1.47. Работа с функциональными именами

Разумеется, при переходе к физической модели никаких ролевых имен не существует. Вкладка редактора связей **Rolename/RI Actions** в физической модели называется **RI actions**, а в целевой СУБД просто будут созданы

столбцы с именами `Ключ1` и `Ключ2` с парой соответствующих ограничений внешнего ключа.

На рис. 1.47 показано, что первичный ключ родительской сущности `СУЩНОСТЬ1` мигрирует в дочернюю сущность `СУЩНОСТЬ2` дважды — один раз с помощью идентифицирующей связи, а второй раз с помощью неидентифицирующей. Показана также вкладка **Rolename/RI Actions** редактора связей, с помощью которой создано ролевое имя `Ключ2` (для связи с именем `Мигрирует_Ключ2`).

Редактор связей, ссылочные действия

Как известно, связь "один-ко-многим" реализуется внешним ключом, наличие которого требует поддержки ссылочной целостности. Эта поддержка сводится к так называемым *ссылочным действиям* (RI actions) — логическим конструкциям, выражающим правила преобразования данных в связанных таблицах при их модификации. Различают:

- *правила удаления* (оператор SQL `DELETE`) — действия при удалении строк;
- *правила вставки* (оператор SQL `INSERT`) — действия при вставке строк в одну из таблиц;
- *правила изменения* (оператор SQL `UPDATE`) — действия при изменении первичного ключа базового отношения или внешнего ключа зависимого отношения.

В Erwin для каждого правила можно задать различные действия, набор которых зависит от обязательности или необязательности связи. К таким действиям относятся:

- *ограничение* (`RESTRICT`), например, запретить добавлять нового студента, если группа для него не задана (`ON CHILD INSERT RESTRICT`);
- *каскадное действие* (`CASCADE`), например, удалить всех студентов данной группы при удалении группы (`ON PARENT DELETE CASCADE`);
- *установка в значение по умолчанию* (`SET DEFAULT`) — например, при удалении группы для всех ее студентов код группы устанавливается в значение по умолчанию (`ON PARENT DELETE SET DEFAULT`);
- *установка в* `NULL` (`SET NULL`) — вместо значения по умолчанию внешний ключ устанавливается в `NULL` .

В Erwin ссылочные действия задаются на той же вкладке, что и ролевые имена. Можно задать отдельно, какие действия производятся при изменении дочерней таблицы (**Child Delete/Insert/Update**) и родительской таблицы (**Parent Delete/Restrict/Update**). Можно задать режим отображения кодов ссылочных действий (например **D:C** для **On Delete Cascade**), если включить опцию **Referential Integrity** в подменю **Display Options/Relationships** контекстного меню правой кнопки мыши.

Связь "много-ко-многим"

Этот тип связи возможен только на уровне концептуальной модели данных. При переходе от концептуальной модели к логической всегда производится преобразование связи "много-ко-многим" в две связи "один-ко-многим" с использованием промежуточной сущности, представляющей эту связь. Такие сущности являются отношениями связи в правилах генерации предварительных отношений и называются *ассоциативными сущностями*.

Атрибуты ассоциативной сущности — это ключевые атрибуты сущностей по обе стороны связи (и, возможно, еще дополнительные атрибуты, характеризующие саму связь). В физической модели, как известно, этот тип связи должен быть разрешен через третье отношение связи *(см. правило 3)*. Для создания связи следует установить курсор на соответствующей кнопке панели инструментов, щелкнуть сначала на одной, а затем на другой сущности. Связь "много-ко-многим" должна именоваться двумя фразами — в обе стороны. При переходе к физическому уровню Erwin автоматически преобразует связь "много-ко-многим", добавляя новую таблицу (ассоциативную сущность) и устанавливая две новые связи "один-ко-многим" от старых к новой таблице. При этом имя новой таблице присваивается автоматически по схеме "`Имя1/Имя2`". Можно связями "много-ко-многим" не пользоваться вовсе, а сразу рисовать модель, используя ассоциативные сущности.

Автоматического решения проблемы связи "много-ко-многим" не всегда оказывается достаточно, т. к. известно, что существуют связи более мощные, чем бинарные. Для этих связей используется правило 7. Отношение связи для этого правила можно строить на основе связи "много-ко-многим", введя в новую таблицу дополнительные атрибуты через связи или непосредственно.

Категориальная связь "супертип/подтип" и иерархия наследования

Иерархия наследования (или иерархия категорий) представляет собой особый тип объединения сущностей, которые разделяют общие характеристики. Чаще всего иерархию наследования создают, когда несколько сущностей имеют общие по смыслу атрибуты, либо когда сущности имеют общие по смыслу связи. Обычно связи этого типа соответствуют ролевым связям ER-диаграмм. Возможны и иные ситуации, продиктованные бизнес-правилами.

Другими словами, сущность может быть разбита на два или более взаимоисключающих *подтипа* таким образом, чтобы у всех этих подтипов были общие атрибуты и/или связи, которые определяются только один раз на верхнем уровне (уровне *предка* или *супертипа*). У подтипов могут быть свои атрибуты и связи. Подтипы могут разбиваться на подтипы и далее. В целом эта концепция соответствует понятию наследования в объектно-ориентиро-

ванном анализе, т. к. сущность-подтип неявно наследует все связи и атрибуты предка.

Для каждой категории можно указать *дискриминатор* — атрибут родового предка, который показывает, как отличить одну категориальную сущность от другой. Иерархии категорий делятся на два типа — полные и неполные. В полной категории одному экземпляру родового предка обязательно соответствует экземпляр в каком-либо потомке. Если в родовом предке могут существовать экземпляры, которые не имеют соответствующих экземпляров в потомках, то такая категория будет неполной. Это в определенной степени соответствует необязательности связи. В нотации IDEF1X полная категория помечается символом , неполная — .

Нотация IE реализует наследование иначе. Здесь различают *взаимно исключающие* (символ) и *перекрывающиеся* (символ) подтипы. Их смысл подобен смыслу рассмотренных ранее обозначений подтипов. При переходе от одной нотации к другой информация о полноте категории может быть потеряна, о чем Erwin и предупреждает пользователя. Пример изображения связи "супертип-подтип" в нотации IDEF1X приведен в левой части на рис. 1.48. В правой части рис. 1.48 показано, как реализовать аналогичную конструкцию с помощью пары обычных идентифицирующих связей и ролевых имен.

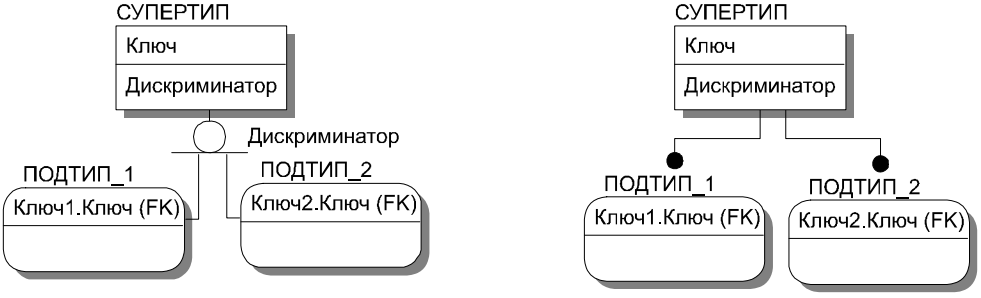


Рис. 1.48. Связь "супертип-подтип" и ее реализация на уровне логической модели

Чтобы реализовать категориальную связь в Erwin, необходимо выбрать соответствующий символ на панели инструментов, щелкнуть кнопкой мыши на предке, потом на одном потомке. Для установления второй связи — щелкнуть кнопкой мыши на символе наследования (окружности), а затем на потомке. Для задания дискриминатора и полноты для данного типа связи нужно дважды щелкнуть кнопкой мыши на символе наследования, в появившемся окне редактора связи **Subtype Relationship Editor** выбрать атрибут дискриминатор (список **Discriminator Attribute Choice**) и указать тип категории — полная/неполная (переключатели **Complete/Incomplete**).

Переход к физической модели и реализация наследования в реляционной базе данных

Особенность наследования заключается в том, что оно напрямую не может быть реализовано в структуре реляционной базы данных, т. е. перед тем, как мы будем генерировать на основе модели физическую структуру базы данных, никакого наследования там быть не должно. При переходе к физической модели категориальные связи, с помощью которых на логическом уровне модели задается иерархия наследования, реализуются просто как набор идентифицирующих связей. При этом сущность-супертип является родительской для всех дочерних сущностей-подтипов. Из сказанного следует, что два способа изображения модели, приведенные на рис. 1.48, дадут эквивалентное содержимое системного каталога целевой СУБД. Однако следует отметить, что выглядеть изображения физических моделей будут все же по-разному (рис. 1.49).

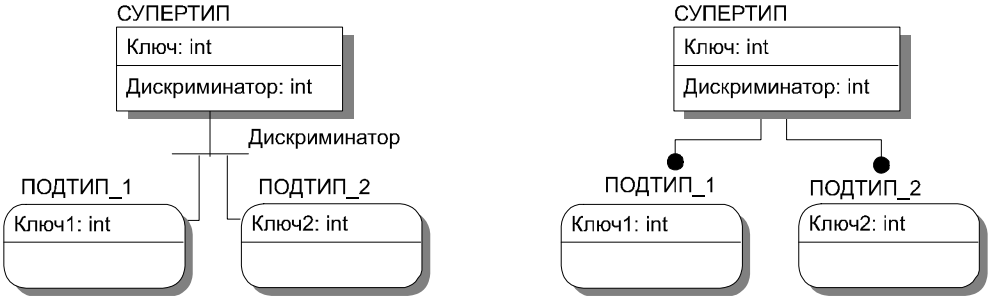


Рис. 1.49. Физическая модель для логической модели, приведенной на рис. 1.48

Из сказанного ранее ясно, что происходит с ключами сущностей, участвующих в категориальных связях, при переходе к физической модели. Но у этих сущностей, разумеется, имеются и неключевые атрибуты, которые, как мы знаем, сущности-подтипы наследуют, с логической точки зрения, от сущности, которая является супертипом. Кроме того, сущности-подтипы способны иметь свои специфические атрибуты. На уровне физической модели мы можем только распределять эти атрибуты по разным сущностям на основе двух основных подходов.

Первый подход состоит в том, что выбирается определенный уровень обобщения для каждой иерархии, т. е. рассматриваются сущности только одного уровня, а информация о наличии остальных сущностей интегрируется в сущности выбранного уровня иерархии путем введения атрибутов типа и обобщения связей. Этот процесс называется *выравниванием модели* (leveling of the model). При выравнивании модели *наследуются* атрибуты супертипов и *сворачиваются* атрибуты подтипов. Свернутые атрибуты становятся необяза-

тельными атрибутами сущности-предка. Само сворачивание можно выполнять указанием опции **Logical Only** для тех объектов, которые сворачиваются, — тогда, присутствуя в логической модели, они не будут переданы в модель физическую. При этом правда, придется иногда изображать такие атрибуты дважды. Можно атрибуты, подлежащие сворачиванию, перемещать из сущностей сворачиваемых уровней в сущности уровня выравнивания модели.

Второй подход состоит в том, что как предки, так и потомки реализуются как отдельные сущности, связанные между собой. При этом атрибуты сущностей распределяются между ними таким образом, что каждый атрибут относится к тому уровню иерархии, на котором он имеет смысл. В этом случае полная информация о сущности-подтипе может быть получена из нескольких таблиц.

Трудно сказать, какой из этих способов лучше, — проблемы имеются в обоих случаях. Например, когда мы выравниваем модель, то необходимо принять решение о том, что делать со связями, в которых участвуют интегрируемые сущности. Подход здесь тот же самый, что и для атрибутов (в конце концов, отношения "один-ко-многим" реализованы именно с помощью атрибутов): мы наследуем связи супертипов и сворачиваем связи подтипов. При сворачивании можно делать связи необязательными, можно обобщить несколько связей в одну. Очевидно, что подобные действия приводят к появлению новых необязательных атрибутов. На уровне СУБД необязательные атрибуты соответствуют столбцам с допустимыми `NULL`-значениями со всеми вытекающими отсюда сложностями. При втором подходе возможностей избежать появления дополнительных столбцов и `NULL`-значений в данных больше. Но в этом случае данные об одной и той же сущности могут оказаться распределенными по нескольким таблицам, которые в запросах придется часто соединять, что тоже требует дополнительной затраты ресурсов.

Рассмотрим оба подхода на примере конкретной схемы. Пусть для модели, приведенной на рис. 1.50, имеются следующие атрибуты, иерархия которых описана в табл. 1.5.

Тогда на основании первого подхода получатся два варианта физической модели в зависимости от выбранного уровня выравнивания, представленные на рис. 1.50. Заметим, что атрибут дискриминатор при выравнивании на уровне потомков не нужен вовсе, т. к. он содержит информацию о том, какому из потомков принадлежит конкретный экземпляр сущности.

На основании второго подхода физическая модель будет выглядеть так, как показано на рис. 1.51.

Проблемы категориальной связи, возникающие вследствие наличия `NULL`-значений, могут до известной степени быть решены, если придерживаться некоторых правил, называемых *правилами обобщения и специализации*.

- Если сущность может быть разбита на подтипы, которые идентифицируются различными ключами, нужно создать сущности для этих подтипов.
- Если сущность может быть разбита на подтипы, у которых разные наборы атрибутов, нужно создать сущности для этих подтипов.
- Если сущность может быть разбита на подтипы, из которых один может участвовать в связи, а другой нет, нужно создать сущности для этих подтипов.

Таблица 1.5. *Атрибуты сущностей в иерархической связи*

Имя атрибута	Принадлежность в иерархии	Первый подход		Второй подход
		Выравнивание на уровне предка	Выравнивание на уровне потомков	
Атр_общ_1	Общие для обоих сущностей-подтипов	NULL-значений нет	NULL-значения имеются	NULL-значений нет нигде
Атр_общ_2				
Атр_общ_3				
Атр_подтип1_1	Относятся только к сущности ПОДТИП1	NULL-значения имеются	NULL-значений нет	
Атр_подтип1_2				
Атр_подтип2	Относится только к сущности ПОДТИП2			

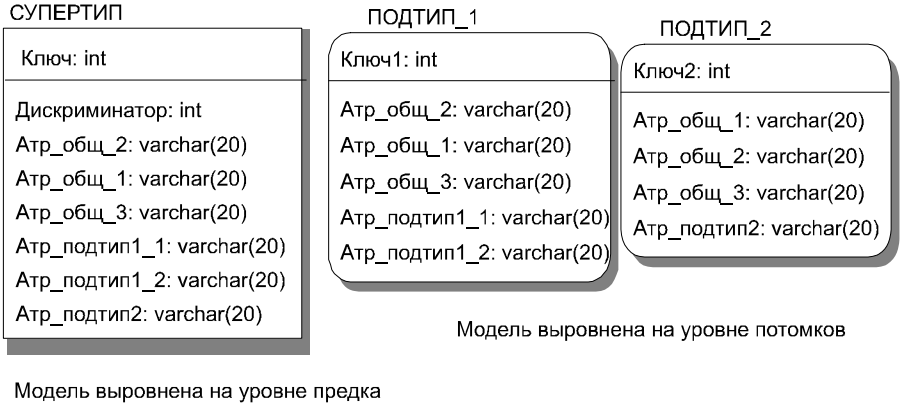


Рис. 1.50. Примеры выравнивания модели

Эти правила не должны касаться всех без исключения атрибутов. Совершенно не обязательно создавать по два подтипа для каждого атрибута, который может принимать NULL-значение, но если у нас есть группа атрибутов, в ко-

торой они все вместе принимают NULL-значение или вместе не равны NULL-значению, то нужно создавать два подтипа: с этой группой атрибутов и без нее.

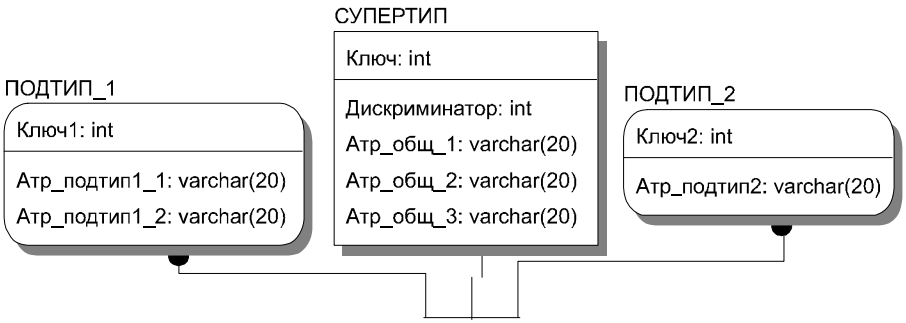


Рис. 1.51. Категориальная связь на уровне физической модели

Не следует путать такие случаи с простыми необязательными связями — если связь иногда справедлива для любого экземпляра, то это необязательная связь, если же она всегда справедлива для некоторого подмножества экземпляров (или никогда не справедлива ни для какого подмножества), то нам нужна специализация.

Продолжим список правил обобщения и специализации.

- Если одни экземпляры сущности участвуют в некоторых процессах, а другие нет, то нужно подумать о создании подтипов. С другой стороны, если сущности участвуют в одном и том же процессе, имеет смысл обратиться к созданию супертипа (поиску общего предка).
- Если сущность может переходить из одного подтипа в другой, то скорее всего такие подтипы не нужны. Они не имеют особого смысла и затрудняют обработку данных.
- Иногда имеет смысл отражать в виде подтипов важные бизнес-правила, например "только менеджеры ходят на собрания менеджеров". Возможно, такое разделение и не сохранится в окончательной модели, потому что оно, как это имеет место в данном случае, скорее всего, нарушает правило запрещения перехода сущностей между подтипами. Но подобный прием позволит на этапе моделирования зафиксировать это правило, чтобы потом его реализовать в виде, например, ограничения.

Один из важных классов связей, которые выражаются в Erwin-модели с помощью категориальной связи, — это так называемые *исключающие связи*, когда две связи одной и той же сущности взаимно исключают друг друга. Например, банковский счет (СЧЕТ) может быть задан для одного и только одного физического лица (ФИЗ_ЛИЦО) или для одного и только одного юридиче-

ского лица (ЮРИД_ЛИЦО). Ясно, что обе роли родительской сущности СЧЕТ совершенно различны — у них и совершенно разные наборы атрибутов. Кроме того, связи, в которые вступают эти роли, полностью разнятся между собой. О таких связях говорят, что они *охвачены дугой* (are in an arc). Это название связано с тем, что в нотации, принятой, например, в Oracle Designer и в нотации Баркера между линиями связей рисуется дуга.

Связи подобного типа достаточно часто реализуются с помощью подтипов. В нашем случае рассмотрим возможность задать общего предка ДЕРЖАТЕЛЬ_СЧЕТА как для роли ФИЗ_ЛИЦО, так и для роли ЮРИД_ЛИЦО. После этого возможны следующие варианты:

- реализовывать сущность ДЕРЖАТЕЛЬ_СЧЕТА как один внешний ключ и поле статуса;
- реализовывать как два внешних ключа, один из которых всегда принимает NULL-значение, что ведет к избыточности;
- занести людей и корпорации в одну таблицу (генерализация);
- разбросать счета по двум таблицам: корпоративные и личные. Возникнут сложности с поиском по всем счетам.

Рекурсивные связи

Рекурсивная связь — это связь, в которой одни и те же сущности участвуют несколько раз или в разных ролях. Классический пример рекурсивной связи — это связь сущности с самой собой. Рекурсивные связи плохо вписываются в стандартную структуру реляционной базы данных, но они встречаются часто и имеют важное значение прежде всего для моделирования иерархических взаимоотношений. У рекурсивной связи есть ряд неформальных названий, происходящих от их обозначения на диаграмме (связь-"чесание затылка" (itch-scratching link), связь-"рыболовный крючок" (fish hook), связь-"свиное ухо" (pig's ear)).

Различают три варианта такой связи:

- *рекурсивная связь "один-к-одному"*, моделирующая *цепочку*;
- *рекурсивная связь "один-ко-многим"* или *иерархическая рекурсивная связь*;
- *рекурсивная связь "много-ко-многим"* или *сетевая рекурсивная связь*.

Первый тип рекурсивной связи имеет место, когда каждый экземпляр сущности связан с одним предыдущим и одним последующим экземплярами, т. е. по сути такая структура данных является связным списком. Цепочку, задающую рекурсивную связь "один-к-одному", можно занести в таблицу, одним из атрибутов которой будет порядковый номер в цепочке. У такого решения есть неудобство — нужно перенумеровывать всю цепочку при вставках или

удалениях в ее середине. Кроме того, нет общепринятого способа изображать такие цепочки, в логической модели просто указываются два атрибута, один из которых будет являться номером.

Наиболее часто используются рекурсивные связи второго и третьего типов. Для их реализации в отношении должны присутствовать два атрибута, принадлежащие одному домену. Организовать такую ситуацию можно с помощью неидентифицирующей связи сущности самой с собой. В случае рекурсивной связи "один-ко-многим" эта связь должна быть необязательной с обоих концов, иначе получится бесконечная иерархия, а у обычного дерева, с помощью которого изображаются иерархические структуры, должен быть корень и листья. Подобная связь в нотации IDEF1X приведена на рис. 1.52. При этом обязательно использование ролевого имени для атрибута связи (в нашем случае `Ключ2`), т. е. на рис. 1.52 правильно построена модель рекурсивной связи для сущности `СУЩНОСТЬ1`.

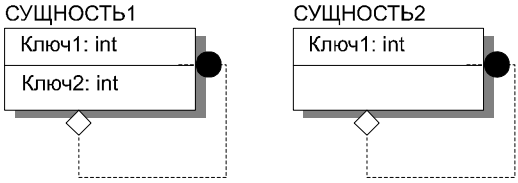


Рис. 1.52. Рекурсивная связь

Теперь в поле `Ключ2` сущности `СУЩНОСТЬ1` можно заносить сведения о родителе того значения, которое занесено в поле первичного ключа `Ключ1`. Чтобы указать родителя для корневого значения, придется в поле `Ключ2` занести `NULL`-значение.

Замечание

Для SQL-сервера сгенерировать таблицы на базе сущности с именем `СУЩНОСТЬ2` вообще невозможно, т. к. ромб на связи указывает на то, что атрибут `Ключ1` является необязательным. Поэтому оператор генерации таблицы для этой сущности выглядит как `CREATE TABLE СУЩНОСТЬ2 (Ключ1 int NULL)`. Для такой таблицы нельзя создать ограничение первичного ключа, т. к. нужный для этого атрибут допускает `NULL`-значения.

Рекурсивная связь "много-ко-многим" должна раскрываться обычно уже на уровне концептуальной модели ассоциативной сущностью и двумя идентифицирующими связями "один-ко-многим". Для того чтобы образовался составной ключ ассоциативной сущности, нужно использовать ролевые имена. На уровне физической модели выглядит все это так, как показано на рис. 1.53.

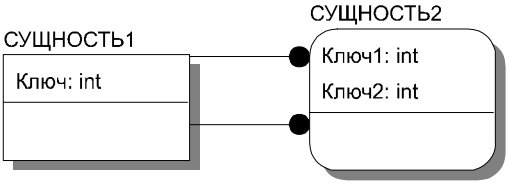


Рис. 1.53. Реализация рекурсии с помощью функциональных имен

Разумеется, для реализации рекурсивной связи "один-ко-многим" или "много-ко-многим" можно просто создать отношение с двумя подходящими столбцами, не входящими в первичный ключ отношения. При этом тип рекурсивной связи не будет уточнен, и ограничения, нужные для каждого из типов такой связи, на уровне сервера также не будут поддерживаться.

1.9.4. Работа с доменами и ключами

Для создания нового домена нужно в меню **Edit** выбрать опцию **Domain Dictionary**. Однажды созданный домен может использоваться и в логической, и в физической модели.

В редакторе словаря доменов нужно нажать **New...**, выбрать в списке домен-предок (из четырех предопределенных и, возможно, заданных ранее), задать имя домена для логической модели в поле **Logical Name** и имя для физической модели в поле **Physical Name**.

Если у нас есть домены, то на их основании можно легко задавать атрибуты с помощью средства **Independent Attribute Browser** (комбинация клавиш — <Ctrl>+<B>). В этом диалоговом окне выбираются домены и по методу "переместить и оставить" перемещаются в сущности. По умолчанию имена получающихся колонок совпадают с именами доменов. В дальнейшем имена столбцов можно изменить.

Мы уже рассмотрели, как работать с первичными и внешними ключами. Помимо этого, Erwin дает возможность работать с такими объектами, как *альтернативные ключи* (alternate keys), которые удовлетворяют определению ключа отношения, но не выбраны в качестве первичных ключей, а также *инверсионные входы*, т. е. неуникальные, но часто используемые для доступа атрибуты (inversion entries).

Чтобы создать альтернативный ключ на уровне логической модели, нужно выбрать в контекстном меню сущности опцию **Key Group Editor...**. В появившемся окне вверху выводится список ключевых групп. Каждая такая группа соответствует какому-либо потенциальному ключу сущности (атомарному или составному). Группы бывают четырех видов: для первичных, внешних, альтернативных ключей и для инверсионных входов. По умолча-

нию ключевые группы для внешних ключей не отображаются, но это можно изменить с помощью переключателя **Show FK Groups**.

Можно задать новую группу выбрав опцию **New...**, после чего появится окно, в котором задается имя для группы и ее тип: альтернативный ключ (уникальная группа) или инверсионный вход (неуникальная группа).

Группа для первичного ключа существует всегда, группы для внешних ключей получаются сами при задании связей. После этого можно включать атрибуты в текущую группу и удалять их оттуда, перемещая их между списками **Available Attributes** и **Key Group Members** на вкладке **Members** (на вкладке **General** можно менять тип текущей группы). При переходе к физической модели альтернативные ключи и инверсные входы автоматически преобразуются в индексы соответствующего типа. В физической модели ключевым группам отвечает опция контекстного меню, называемая редактором индексов. Для SQL-сервера название этой опции — **SQL Server Index...**.

1.10. Примеры проектных решений в Erwin

В данном разделе приводятся и обсуждаются конкретные примеры описания предметных областей и полученные для них проектные решения, представленные в виде логической модели в IDEF1X-нотации CASE-средства Erwin. При этом при подборе примеров преследовалась цель показать наиболее типичные ситуации и проблемы, которые приходится решать при реальном проектировании реляционной базы данных. Разумеется, следует понимать, что нет и не бывает проектных решений, о которых можно было бы сказать, что именно они являются единственно правильными, а все остальные варианты ошибочны, хотя, безусловно, существуют общепринятые способы решать типичные задачи. Разработка базы данных является достаточно неформализованным процессом, особенно на начальных стадиях. Поэтому нужно уметь прогнозировать, к каким последствиям, с точки зрения производительности системы, приведет то или иное изменение в проектном решении.

1.10.1. Использование функциональных (ролевых) имен

Как известно, применение функциональных, или ролевых, имен полезно в том случае, когда реализуется ролевая связь ER-диаграммы. Рассмотрим пример использования ролевых имен, который считается классическим. Пусть в предметной области имеется информация об акте обмена валюты, в котором участвуют две валюты — проданная и купленная. Информация о валютах содержится в сущности `ВАЛЮТА`. Следовательно, сущности `ОБМЕН_ВАЛЮТЫ` и `ВАЛЮТА` должны быть связаны дважды, и первичный ключ сущности

ВАЛЮТА, который называется Валюта_Ид, должен дважды мигрировать в сущность ОБМЕН_ВАЛЮТЫ в качестве внешнего ключа. Необходимо различать атрибуты, которые содержат информацию об идентификаторе проданной и купленной валюты, т. к., с точки зрения бизнес-операции по обмену валюты, они имеют разный смысл, ссылаясь при этом на одну и ту же сущность ВАЛЮТА. Изображение логической модели для этого случая приведено на рис. 1.54.

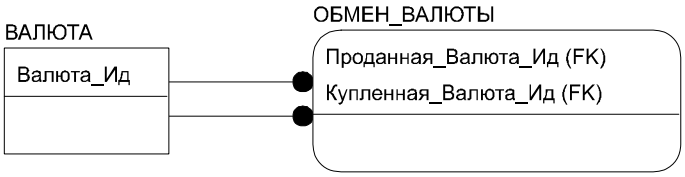


Рис. 1.54. Функциональные имена и их использование

Развивая данное проектное решение, можно рассматривать возможность введения атрибутов сущности ОБМЕН_ВАЛЮТЫ. К ним, например, относится сумма, которую обменяли. Однако рассуждая дальше, нужно сказать, что сущность ОБМЕН_ВАЛЮТЫ порождает серьезную проблему. Действительно, хоть на рис. 1.54 и показано, что она имеет первичный ключ, но на самом деле, с точки зрения семантики данных, трудно признать, что пара атрибутов (Проданная_Валюта_Ид, Купленная_Валюта_Ид) реально обладают ограничением уникальности в ситуации, когда операции по обмену валюты производятся постоянно. Поэтому сущность ОБМЕН_ВАЛЮТЫ подходит для занесения в нее атрибутов, которые описывают какие-то постоянные данные об операциях по обмену валюты одного конкретного вида на другой. Для того чтобы хранить информацию об операциях по обмену валюты, нужно, очевидно, создать специальную сущность, например с именем ОПЕРАЦИЯ_ОБМЕНА. Эта сущность может идентифицироваться автоинкрементным суррогатным ключом — номером сделки. Тогда ключ сущности ВАЛЮТА должен дважды мигрировать в сущность ОПЕРАЦИЯ_ОБМЕНА с помощью неидентифицирующих связей с функциональными именами.

1.10.2. Примеры реализации рекурсии

Другой пример использования функциональных имен — рекурсивные связи, которые, как известно, реализуют иерархическую рекурсию. Классическим примером рекурсивной связи является создание иерархии подчиненности, например для сотрудников организации. Сущность СОТРУДНИК содержит атрибут первичного ключа Сотрудник_Ид. Информация о руководителе сотрудника содержится в той же сущности, поскольку руководитель работает в той же организации. Чтобы сослаться на руководителя сотрудника, следует соз-

дать рекурсивную связь и присвоить имя роли `Начальник_Ид`. Напомним, что подобная рекурсивная связь может быть только неидентифицирующей. В противном случае внешний ключ должен был бы войти в состав первичного ключа и получить при генерации схемы признак `NOT NULL`. Это сделало бы невозможным построение иерархии, поскольку у дерева подчиненности должен быть корень — сотрудник, который никому не подчиняется в рамках данной организации. Приведенная связь позволяет хранить иерархию подчиненности сотрудников, которая соответствует ситуации, когда руководитель может иметь множество подчиненных, но подчиненный имеет только одного руководителя.

Для примера возьмем вариант простейшей иерархии подчиненности, когда в организации имеется директор, ему непосредственно подчиняются начальники отделов, а им уже подчиняются сотрудники отделов. Нужно знать, в каком из отделов работает конкретный сотрудник. Кроме того, нужно хранить информацию о сотрудниках и об отделах. Пример логической модели для описанной ситуации с реализацией иерархии подчиненности приведен на рис. 1.55.

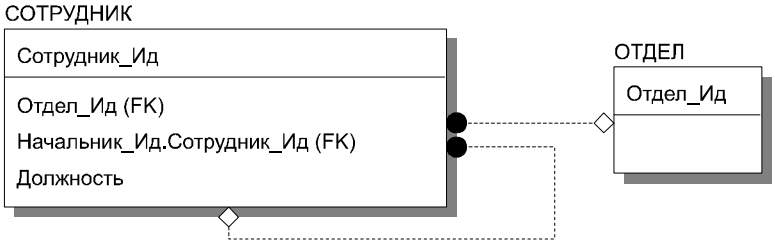


Рис. 1.55. Реализация иерархии подчиненности

Покажем сначала, что данное проектное решение позволяет хранить сведения об иерархии сотрудников. Рассмотрим базовую таблицу (табл. 1.6), созданную на основании сущности `СОТРУДНИК`.

Видно, что столбцы `Сотрудник_Ид` и `Начальник_Ид` позволяют описать имеющуюся иерархию подчиненности между сотрудниками и продолжить ее на столько уровней, на сколько потребуется (в приведенном примере описано три уровня иерархии). Признаком корня дерева является наличие `NULL`-значения в поле `Начальник_Ид`. Ясно, что такая запись может быть только одна.

Однако в приведенном проектном решении имеются и недостатки. Очевидно, что данные из столбца `Отдел_Ид` частично являются избыточными, потому что информацию об отделе, в котором работает сотрудник, можно получить из дерева подчиненности зная начальника этого сотрудника и отдел, которым руководит начальник. То есть номер отдела можно хранить только для на-

чальников отделов. Если же этот номер хранится для всех сотрудников, то появляется необходимость согласования данных о номере отдела, полученном при проходе по дереву подчиненности, с номером отдела, полученным непосредственно из записи таблицы СОТРУДНИК. Подобное согласование может быть выполнено, например, с помощью триггера.

Таблица 1.6. *Возможное заполнение рекурсивными данными базовой таблицы*

Сотрудник_Ид	Отдел_Ид	Начальник_Ид	Должность
1			Директор
2	Отдел 1	1	Начальник отдела
3	Отдел 2	1	Начальник отдела
4	Отдел 3	1	Начальник отдела
5	Отдел 1	2	Рядовой сотрудник
6	Отдел 1	2	Рядовой сотрудник
7	Отдел 2	3	Рядовой сотрудник
8	Отдел 2	3	Рядовой сотрудник
9	Отдел 2	3	Рядовой сотрудник
10	Отдел 3	4	Рядовой сотрудник
11	Отдел 3	4	Рядовой сотрудник
12	Отдел 3	4	Рядовой сотрудник

Если мы откажемся от хранения номеров отделов для рядовых сотрудников, то, во-первых, в таблице СОТРУДНИК будет много NULL-значений, а во-вторых, этот номер будет достаточно трудно получить, потому что нужно будет реализовывать алгоритм прохода по дереву подчиненности, отслеживая его уровни. В текущей версии синтаксиса языка T-SQL не предусмотрено специальных конструкций для работы с иерархическими структурами, поэтому подобная процедура требует достаточно сложной программы.

Подобные рассуждения справедливы и для атрибута Должность, т. к. его значение однозначно определяется уровнем в иерархии подчиненности, на котором находится конкретный экземпляр сущности СОТРУДНИК.

Возможно, что более удачным будет проектное решение, при котором принимается во внимание тот факт, что если ввести в рассмотрение роли с названиями НАЧАЛЬНИК_ОТДЕЛА и РЯДОВОЙ_СОТРУДНИК для сущности СОТРУДНИК, то появится необязательная со стороны роли НАЧАЛЬНИК_ОТДЕЛА связь между

этой ролью и сущностью ОТДЕЛ. Мощность этой связи — "один-к-одному". Тогда, приняв во внимание правила 5 и 8 генерации предварительных отношений, проектное решение может быть изменено, как показано на рис. 1.56.

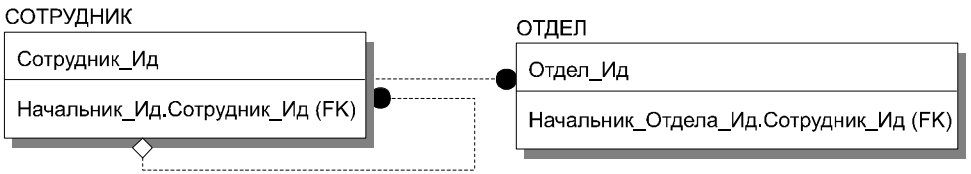


Рис. 1.56. Другое проектное решение для реализации иерархии подчиненности

Для обеих связей, приведенных на рис. 1.56, использованы соответствующие функциональные имена.

Очевидно, что для проектного решения, представленного на рис. 1.56, можно ответить как на вопрос о том, кто в каком отделе работает, так и кто какую должность занимает, хотя для этого всякий раз придется отслеживать уровень, который занимает в иерархии каждый из сотрудников. Безусловно, получить эту информацию из проектного решения, приведенного на рис. 1.55, проще.

При реализации иерархической структуры часто возникают проблемы, связанные с тем, что появляются различные дополнительные связи между экземплярами сущности, участвующей в рекурсии на различных уровнях иерархии. Кроме того, на разных уровнях иерархии экземпляры сущностей могут иметь различные наборы атрибутов, а также могут по-разному вступать в связи с другими сущностями. Все эти возможности необходимо анализировать, выбирая из возможных проектных решений оптимальное.

1.10.3. Сущности, вводимые только на логическом уровне

Пусть предложено следующее описание предметной области. В базе данных должны храниться сведения о счетах клиентов банка, которые являются физическими лицами. При этом возможно, что один счет обслуживает одного клиента, но некоторые клиенты имеют несколько счетов. Счета открываются в разных валютах. Клиенты могут открывать и пополнять свои счета, а также снимать деньги со счета. Банк начисляет проценты на счета в определенные моменты времени. В базе данных должна храниться история финансовых операций с каждым счетом. Кроме того, нужно хранить данные о курсах валют. Предполагается выполнение следующих транзакций: открытие нового счета клиентом, выполнение операций со счетом, начисление процентов на счет банком и задание курсов валют на новый день работы банка.

Проектное решение на уровне логической модели для приведенного описания предметной области может быть таким, как представлено на рис. 1.57, а в табл. 1.7 приведено описание смысла входящих в проектное решение атрибутов.

Таблица 1.7. *Описание атрибутов проектного решения*

Название атрибута	Что в нем хранится
Операция_Клиента_Ид	Номер операции, которую клиенты выполняют со своими счетами в сквозной нумерации
Дата_Операции	Дата выполнения операции со счетом
Деньги	Количество денег, на которое изменился остаток на счете в результате выполнения операции
Тип_операции	Тип операции (например пополнение счета или снятие денег со счета)
Дата	Дата работы банка
Тип_Счета	Идентификатор валюты, в которой открыт счет
Курс	Курс валюты по отношению к базовой национальной валюте
Дата_Открытия	Дата открытия счета
Клиент_Ид	Идентификатор клиента банка, который распоряжается счетом
Остаток_на_Счете	Остаток денежных средств на счете
Операция_Банка_Ид	Номер операции в сквозной нумерации по начислению процентов на счет, которую банк выполняет со счетами клиентов

Проектное решение для рис. 1.57 на уровне физической модели приведено на рис. 1.58.

Прокомментируем данное проектное решение.

Таблица ИСТОРИЯ_ОПЕРАЦИЙ_КЛИЕНТОВ содержит сведения об операциях, которые клиент выполнял со своим счетом. Первичный ключ этой таблицы представляет собой просто числовой идентификатор и может быть автоинкрементным. Этот первичный ключ можно считать суррогатным — клиенту (пользователю) не важно, как нумеруются записи об операциях на уровне базы данных. Заметим, что для сущности ИСТОРИЯ_ОПЕРАЦИЙ_КЛИЕНТОВ достаточно проблематично указать ключ, который бы нес семантическую нагрузку. Действительно, если клиент имеет возможность выполнять нескольких одинаковых по типу операций со своим счетом в течение операционного дня,

например снять со своего счета дважды в день одну и ту же сумму денег, то среди атрибутов сущности `ИСТОРИЯ_ОПЕРАЦИЙ_КЛИЕНТОВ` вообще невозможно найти такую их комбинацию, которая удовлетворяла бы определению ключа. Поэтому единственным выходом является создание суррогатного ключа для указанной сущности.

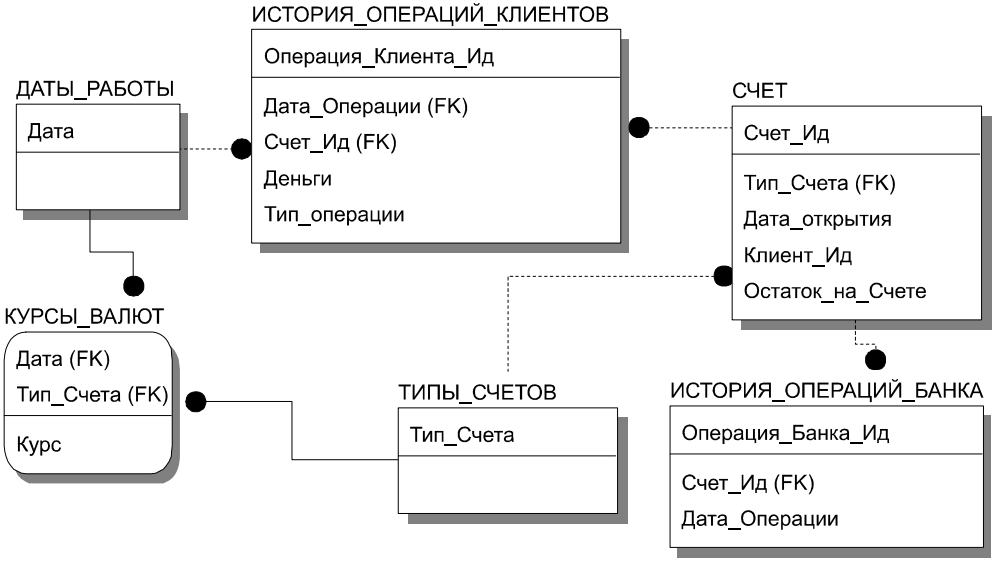


Рис. 1.57. Проектное решение для предметной области, связанной с банковскими операциями

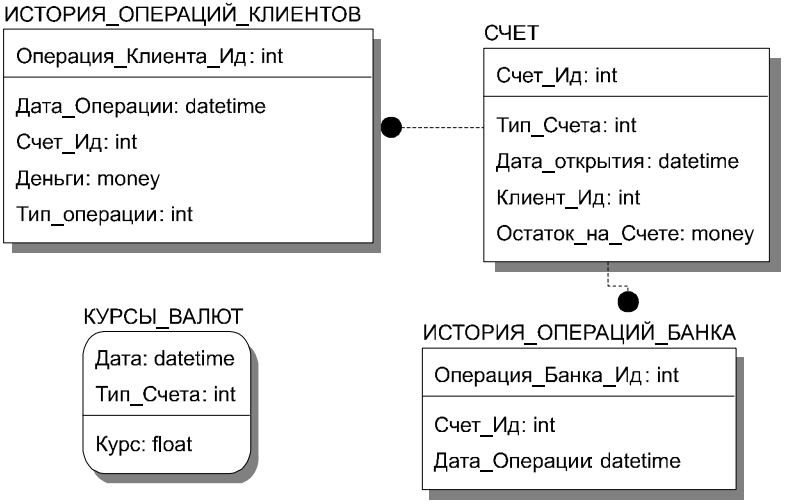


Рис. 1.58. Физический уровень проектного решения предыдущего рисунка

Таблица КУРСЫ_ВАЛЮТ содержит сведения о курсах валют. При этом данная таблица создана как ассоциативная сущность и имеет составной ключ. Сущность ДАТЫ_РАБОТЫ содержит единственный ключевой атрибут под названием Дата. Эта сущность введена в логическую модель для того, чтобы реализовать возможность миграции ее первичного ключа в те таблицы, которые содержат сведения о времени выполнения операций. Ключ этой сущности мигрирует, например, в сущность ИСТОРИЯ_ОПЕРАЦИЙ_КЛИЕНТОВ с использованием функционального имени Дата_Операции. Тем самым в логической модели удалось подчеркнуть тот факт, что операции клиентами выполняются только в те дни, когда банк работает. Разумеется, поддержание подобного ограничения на физическом уровне с помощью ограничения внешнего ключа нелепо. Поэтому сущность ДАТЫ_РАБОТЫ используется только на уровне логической модели (для нее задана опция **Logical Only**). В принципе, нужно связать неидентифицирующей связью сущность ДАТЫ_РАБОТЫ также и с сущностью ИСТОРИЯ_ОПЕРАЦИЙ_БАНКА, если уж поддерживать бизнес-правило о согласовании дат работы и дат выполнения операций до конца. Но эта связь не нарисована, т. к. если и предполагается поддерживать данное ограничение на уровне физической модели, то, скорее всего, это будет сделано просто путем автогенерации дат при выполнении транзакций.

Создание сущности ТИПЫ_СЧЕТОВ служит для аналогичных целей: во-первых, чтобы показать, в какие сущности мигрирует ее первичный ключ, и во-вторых, в предположении о возможности появления атрибутов, зависящих от типа счета. Таким атрибутом, например, может оказаться величина процентов, начисляемых на остаток средств на счете, если эта величина зависит от типа счета. Если подобных атрибутов не обнаружено, то сущность ТИПЫ_СЧЕТОВ также можно отключить на уровне физической модели.

Вообще, создание сущностей, содержащих только атомарные ключи, на уровне логической модели может быть оправдано, если нужно подчеркнуть, что атрибуты разных таблиц, которые вследствие каких-то причин названы в этих таблицах по-разному, получают значения из одного и того же логического домена.

Сущность СЧЕТ не требует особых комментариев. Скажем только, что у этой сущности имеется избыточный атрибут, названный Остаток_на_Счете. Размер этого остатка должен в любой момент быть согласованным с результатами операций по изменению счетов, сведения о которых содержатся в атрибуте Деньги сущности ИСТОРИЯ_ОПЕРАЦИЙ_КЛИЕНТОВ. Кроме того, нужно также принять во внимание установленное банком бизнес-правило по поводу начисления процентов на счета. Введение атрибута Остаток_на_Счете связано с оптимизацией производительности базы данных, и об этом много говорится при обсуждении аналогичного приема в проектном решении для задачи материального учета в *главе 5*.

Самым спорным моментом данного проектного решения является создание сущности и таблицы под названием `ИСТОРИЯ_ОПЕРАЦИЙ_БАНКА`. Эта сущность имеет тот же самый ключ, что и сущность `ИСТОРИЯ_ОПЕРАЦИЙ_КЛИЕНТОВ`, но объединить две указанные сущности по этому ключу проблематично, т. к. они содержат разные наборы атрибутов и, кроме того, сами операции, сведения о которых хранятся в указанных сущностях, имеют различный бизнес-смысл. Более того, возможно, что таблица `ИСТОРИЯ_ОПЕРАЦИЙ_БАНКА` вообще содержит только избыточные данные. Это зависит от принятого банком бизнес-правила по начислению процентов на счета. Пусть данное бизнес-правило формулируется простейшим способом, например, так: "на каждый счет по истечении очередного фиксированного периода начисляется фиксированный процент". В этом случае достаточно программного способа реализации приведенного бизнес-правила, который заключается просто в автоматическом пересчете значения в столбце `Остаток_на_Счете` в нужные моменты времени.

1.10.4. Пример анализа влияния проектного решения на последующие стадии жизненного цикла базы данных

Предположим, что проектируется база данных, связанная с ведением учета выплаты заработной платы сотрудникам организации. Рассмотрим небольшую часть описания предметной области, которая связана с назначением таких денежных выплат, как надбавки к основному окладу.

Пусть о сотруднике нужно знать анкетные данные, табельный номер, должность. Для должностей нужно знать тарифную сетку (размер основного оклада при повременной оплате труда). Предполагается, что один сотрудник занимает одну определенную должность. Размер надбавки задается в процентах от размера основного оклада. Надбавка назначается помесячно и может начисляться несколько месяцев, а потом сниматься. Основанием для назначения надбавки служит приказ, формулировка которого примерно следующая: "Сотруднику такому-то — назначить надбавку указанного типа на указанный диапазон месяцев (указывается первый и последний месяцы диапазона)". В приказе с одним номером допускается назначение разных надбавок одному или нескольким сотрудникам. Разумеется, может быть издан новый приказ, согласно которому определенная надбавка опять назначается определенному сотруднику. Снятие уже назначенной надбавки до истечения ее срока не разрешается.

Далее перечислены бизнес-правила, подлежащие реализации.

- Существует перечень надбавок к основному окладу, которые зависят от должностей, занимаемых сотрудниками. То есть при назначении надбавки

имеются ограничения, связанные с тем, что для каждой должности известен перечень надбавок, которые для нее допустимы. Например, нельзя назначить надбавку "За использование вычислительной техники" сотруднику с должностью "Дворник".

- Сумма выплат по надбавкам за любой месяц не должна превышать основного оклада.
- При назначении надбавки очевидно, что имеется следующее скрытое ограничение: нельзя, чтобы для одного и того же сотрудника одна и та же надбавка дважды была включена в перекрывающиеся диапазоны месяцев. Если указанному ограничению присвоить название "`Без_Перекрытия`", то недопустима ситуация, показанная в табл. 1.8.

Таблица 1.8. *Пример нарушения ограничения "`Без_Перекрытия`"*

Табельный номер	Ф.И.О.	Код надбавки	Месяц начала выплаты надбавки	Месяц окончания выплаты надбавки
1234	Вакин В. В.	33	1	4
1234	Вакин В. В.	33	3	6

Запросы, которые нужно реализовать, связаны с выпиской следующих документов:

- ежемесячная ведомость на выплату зарплаты для всех сотрудников предприятия с учетом надбавок;
- справка о суммарных выплатах денежных средств заданному сотруднику с учетом надбавок за указанный диапазон месяцев.

Работа с данными для хранения информации о времени осуществлена следующим образом. Для каждого месяца введен каким-то из способов его идентификатор. Проще всего в качестве такого идентификатора указать календарную дату, которая соответствует первому числу соответствующего месяца. В этом случае придется выбрать для идентификатора месяца тип данных `Date/Time`. Очевидно, что в рамках данной задачи постоянно придется подсчитывать количество месяцев, попавшее в тот или иной диапазон. При описанном способе введения идентификатора месяца это можно сделать с использованием специальных функций календарной арифметики, которые имеются в любой СУБД. Например, для SQL-сервера функцией, с помощью которой подсчитывается количество календарных периодов, попавших в заданный диапазон между двумя датами, является `DATEDIFF`.

Возможен и другой подход, например, каждому из месяцев, в течение которых функционирует база данных, автоматически присваивается целочисленный автоинкрементный идентификатор. Этот идентификатор заносится в специальную таблицу, с тем чтобы каждому его значению можно было сопоставить календарное значение месяца. В этом случае на уровне базы данных вместо сравнительно медленной календарной арифметики возможно использовать обычную, но на уровне пользователя все равно придется оперировать с обычными датами, потому что ведомость на получение зарплаты, например за двадцать второй месяц, будет выглядеть странно.

Как бы то ни было, в рамках данной задачи считается, что если заданы начальный и конечный месяцы диапазона (`@Мес1` и `@Мес2` соответственно), то разность `(@Мес2 – @Мес1 + 1)` равна количеству месяцев, попавших в диапазон.

Изобразим модель, которая позволяет хранить необходимые данные. Модель нарисована в IDEF1X-нотации логической модели Erwin и представлена на рис. 1.59.

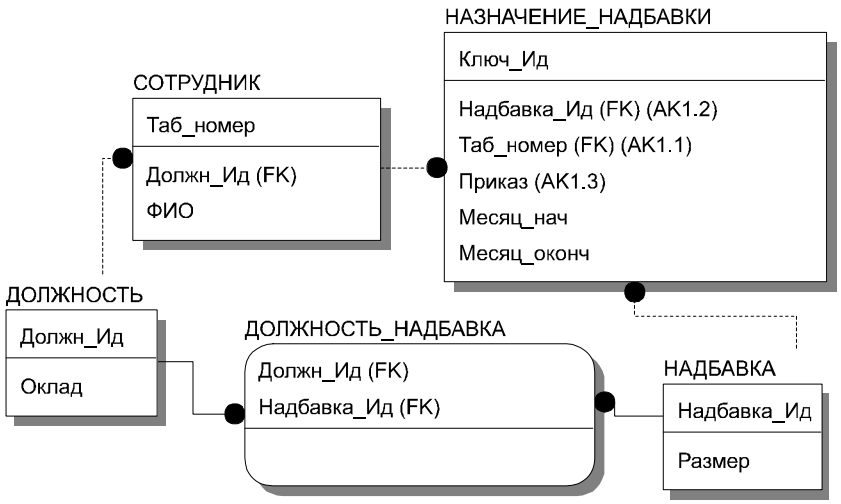


Рис. 1.59. Проектное решение для предметной области, связанной с назначением надбавок

Не вызывают особых проблем сущности `ДОЛЖНОСТЬ`, `СОТРУДНИК` и `НАДБАВКА`. Ассоциативная сущность `ДОЛЖНОСТЬ_НАДБАВКА` служит для хранения сведений о том, какие надбавки допустимы для каких должностей.

Рассмотрим способ хранения данных о назначении надбавки. В данном проектном решении создана сущность `НАЗНАЧЕНИЕ_НАДБАВКИ`, для которой введен суррогатный ключ с именем `Ключ_Ид`. Эта сущность позволяет хранить сведе-

ния о назначении надбавок, причем на каждый факт назначения надбавки, согласно приказу, будет создана одна запись в таблице НАЗНАЧЕНИЕ_НАДБАВКИ.

Попробуем определить потенциальный ключ сущности НАЗНАЧЕНИЕ_НАДБАВКИ. Им, очевидно, считается тройка атрибутов (Надбавка_Ид, Таб_номер, Приказ), потому что в одном и том же приказе не может дважды назначаться для конкретного сотрудника одна и та же надбавка. В модели это обстоятельство показано с помощью обозначения альтернативного ключа для указанной группы атрибутов.

Итак, при использовании проектного решения, приведенного на рис. 1.59, все данные удалось разместить в базе данных.

Далее следует убедиться, что на полученном проектном решении можно реализовать ограничения для бизнес-правил, а также необходимые запросы. Этот момент достаточно сложен — ведь нужно увидеть потенциальную возможность написания определенного программного кода еще не приступая, собственно, к стадии программирования.

В нашем проектном решении, безусловно, допускается возможность проверить нужные ограничения в момент выполнения транзакции по назначению конкретной надбавки конкретному сотруднику, которая выглядит как ввод в клиентскую форму соответствующей строки приказа. Сложнее всего выглядит проверка ограничения "Без_Перекрытия", но идея его проверки будет ясна из разбора способа программирования запросов, т. к. здесь также нужна работа с диапазонами месяцев. Проверка ограничений может быть сделана разными способами как на сервере, так и на клиенте. Возможно создание триггеров, которые будут поддерживать целостность базы данных по сформулированным ограничениям. Может быть принято решение о том, что достаточно выполнять указанные проверки только на клиенте.

Данное проектное решение не дает простой возможности вычисления денежных сумм, необходимых для реализации запросов, несущих основную функциональность системы. Для того чтобы для заданного сотрудника узнать сумму выплат за указанный месяц (ведомость на зарплату), нужно в запросе каким-то образом запрограммировать проверку того, попал ли заданный месяц в диапазон месяцев с назначенной надбавкой.

После того как необходимые таблицы связаны по внешним ключам, считать деньги можно следующим способом. Пусть нужно посчитать для заданного табельным номером сотрудника сумму выплат за диапазон месяцев [@Мес1; @Мес2]. К выплатам по окладу прибавляется сумма надбавок, обозначенная как Сумм_надб, которую нужно выплатить сотруднику за диапазон месяцев [@Мес1; @Мес2]; в том случае, если для заданного сотрудника какой-нибудь из месяцев диапазона [@Мес1; @Мес2] попадает в диапазон месяцев назначения

надбавки [Месяц_нач; Месяц_оконч] из таблицы НАЗНАЧЕНИЕ_НАДБАВКИ. При этом возможны следующие ситуации.

- Диапазон [@Мес1; @Мес2] полностью входит в диапазон [Месяц_нач; Месяц_оконч], т. е. выполняется условие:

  @Мес1 >= Месяц_нач and @Мес2 <= Месяц_оконч.

  Тогда сумма надбавок вычисляется по формуле:

  @Сумм_надб = Размер*(@Мес2 – @Мес1 + 1).

- Диапазон [@Мес1; @Мес2] полностью перекрывает диапазон [Месяц_нач; Месяц_оконч], т. е. выполняется условие:

  @Мес1 <= Месяц_нач and @Мес2 >= Месяц_оконч.

  Тогда сумма надбавок вычисляется по формуле:

  @Сумм_надб = Размер*(Месяц_оконч – Месяц_нач + 1).

- Диапазон [@Мес1; @Мес2] пересекается с диапазоном [Месяц_нач; Месяц_оконч] слева, т. е. выполняется условие:

  @Мес1 < Месяц_нач and @Мес2 < Месяц_оконч.

  Тогда сумма надбавок вычисляется по формуле:

  @Сумм_надб = Размер*(@Мес2 – Месяц_нач + 1).

- Диапазон [@Мес1; @Мес2] пересекается с диапазоном [Месяц_нач; Месяц_оконч] справа, т. е. выполняется условие:

  @Мес1 > Месяц_нач and @Мес2 > Месяц_оконч.

  Тогда сумма надбавок вычисляется по формуле:

  @Сумм_надб = Размер*(Месяц_оконч – @Мес1 + 1).

После этих подсчетов окончательная сумма выплат за заданный диапазон месяцев может быть получена по формуле:

Сумм_выпл = Оклад*(@Мес2 – @Мес1 + 1) + @Сумм_надб.

Эти вычисления необходимо реализовать в рамках операторов SELECT, и задача в целом, таким образом, решается, хотя и достаточно сложно (листинг 1.2).

Листинг 1.2. Вычисление надбавок к окладу

```
DECLARE @Мес2 int,@Мес1 int,@1 int, @Сумм_надб money
SET @Мес1 = -1
SET @Мес2 = 2
SET @1=7
```

```
if exists
       (SELECT Ключ_Ид FROM НАЗНАЧЕНИЕ_НАДБАВКИ
       WHERE @Мес1 >= Месяц_нач and @Мес2 <= Месяц_оконч)
              SELECT @Сумм_надб = Размер*(@Мес2 — @Мес1 + 1)
              --, Размер,Ключ_Ид, 'Случай 1'
              FROM НАЗНАЧЕНИЕ_НАДБАВКИ INNER JOIN НАДБАВКА ON
              НАЗНАЧЕНИЕ_НАДБАВКИ.Надбавка_Ид = НАДБАВКА.Надбавка_Ид
              WHERE (НАЗНАЧЕНИЕ_НАДБАВКИ.Таб_номер = @1)
              AND (@Мес1 >= Месяц_нач and @Мес2 <= Месяц_оконч)
--
if exists
       (SELECT Ключ_Ид FROM НАЗНАЧЕНИЕ_НАДБАВКИ
       WHERE @Мес1 <= Месяц_нач and @Мес2 >= Месяц_оконч)
              SELECT @Сумм_надб = Размер*(Месяц_оконч — Месяц_нач + 1)
              --, Размер,Ключ_Ид,'Случай 2'
              FROM НАЗНАЧЕНИЕ_НАДБАВКИ INNER JOIN НАДБАВКА ON
               НАЗНАЧЕНИЕ_НАДБАВКИ.Надбавка_Ид = НАДБАВКА.Надбавка_Ид
              WHERE (НАЗНАЧЕНИЕ_НАДБАВКИ.Таб_номер = @1)
              AND (@Мес1 <= Месяц_нач and @Мес2 >= Месяц_оконч)
--
if exists
       (SELECT Ключ_Ид FROM НАЗНАЧЕНИЕ_НАДБАВКИ
       WHERE @Мес1 <Месяц_нач and @Мес2< Месяц_оконч)
              SELECT @Сумм_надб = Размер*(@Мес2 — Месяц_нач + 1)
              --, Размер, Ключ_Ид, 'Случай 3'
              FROM НАЗНАЧЕНИЕ_НАДБАВКИ INNER JOIN НАДБАВКА ON
              НАЗНАЧЕНИЕ_НАДБАВКИ.Надбавка_Ид = НАДБАВКА.Надбавка_Ид
              WHERE (НАЗНАЧЕНИЕ_НАДБАВКИ.Таб_номер = @1)AND
              (@Мес1 <Месяц_нач and @Мес2< Месяц_оконч)
--
if exists
       (SELECT Ключ_Ид FROM НАЗНАЧЕНИЕ_НАДБАВКИ
       WHERE @Мес1 >Месяц_нач and @Мес2 > Месяц_оконч)
              SELECT @Сумм_надб = Размер*(Месяц_оконч — @Мес1 + 1)
              --, Размер,Ключ_Ид,'Случай 4'
              FROM НАЗНАЧЕНИЕ_НАДБАВКИ INNER JOIN НАДБАВКА ON
              НАЗНАЧЕНИЕ_НАДБАВКИ.Надбавка_Ид = НАДБАВКА.Надбавка_Ид
              WHERE (НАЗНАЧЕНИЕ_НАДБАВКИ.Таб_номер = @1)AND
              (@Мес1 > Месяц_нач and @Мес2 > Месяц_оконч)
-- Наконец, вычисляется сумма выплат
SELECT Сумм_выпл = Оклад*(@Мес2 — @Мес1 + 1) + @Сумм_надб
FROM Сотрудник INNER JOIN Должность
       ON СОТРУДНИК.Должность_Ид = ДОЛЖНОСТЬ.Должность_Ид
WHERE Таб_номер = @1
```

Рассмотрим, к каким последствиям ведет данное проектное решение при его реализации. Суррогатный первичный ключ `Ключ_Ид` позволит создать простой кластерный индекс для таблицы `НАЗНАЧЕНИЕ_НАДБАВКИ`. Если бы в качестве первичного ключа был выбран альтернативный составной ключ (`Надбавка_Ид`, `Таб_номер`, `Приказ`), то кластерный индекс этой таблицы оказался бы очень громоздким. В указанных запросах не требуется осуществление поиска по кластерному индексу таблицы `НАЗНАЧЕНИЕ_НАДБАВКИ`. Не нужен и поиск по полному альтернативному ключу, поэтому индекс по всем трем атрибутам альтернативного ключа создавать нет необходимости. Для реализации второго запроса требуется организовать поиск по табельному номеру для таблицы `НАЗНАЧЕНИЕ_НАДБАВКИ`, и возможно, соответствующий индекс ускорит этот поиск, хотя очевидно, что селективность индекса по столбцу `Таб_номер` явно недостаточна для того, чтобы он был полезен для поиска.

В случае использования таблицы `Назначение_надбавки` создание индексов по полям `Месяц_нач` и `Месяц_оконч` теоретически может оказаться полезным для ускорения отбора записей из этой таблицы по условиям сравнения, нужным при подсчетах денежных сумм. Но с практической точки зрения, имеется много возражений против создания этих индексов. Индексы явно будут низкоселективными, а сделать их оба кластерными невозможно.

Попробуем найти проектное решение, которое позволило бы проще реализовать подсчеты денег, выплачиваемых сотрудникам. Например, изменим проектное решение следующим образом (рис. 1.60).

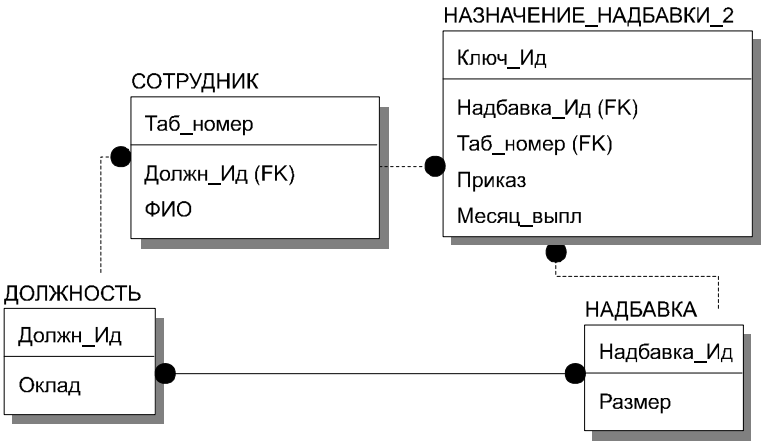


Рис. 1.60. Другое проектное решение для предметной области, связанной с назначением надбавок

Теперь вместо таблицы `НАЗНАЧЕНИЕ_НАДБАВКИ` создана таблица с именем `НАЗНАЧЕНИЕ_НАДБАВКИ_2`, в которой указывается не начало и конец диапазона месяцев назначения надбавки, а каждый месяц, в котором данная надбавка

выплачивается заданному человеку. То есть из одной записи таблицы НАЗНАЧЕНИЕ_НАДБАВКИ получится несколько записей в таблице НАЗНАЧЕНИЕ_НАДБАВКИ_2, а именно столько, сколько месяцев входит в диапазон Месяц_нач ÷ Месяц_оконч. Альтернативным ключом таблицы НАЗНАЧЕНИЕ_НАДБАВКИ_2 станут все ее четыре атрибута (Надбавка_Ид, Таб_номер, Приказ, Месяц_выпл).

Что же получится в этом случае? Для программирования запросов и в том, и в другом случае полезно максимально быстро отобрать только те записи из таблицы НАЗНАЧЕНИЕ_НАДБАВКИ_2, которые нужны для подсчета денег. В случае второго решения этим условием отбора может стать Месяц_выпл between Месяц1 and Месяц2. Для ускорения работы этого условия целесообразно создать индекс по полю Месяц_выпл, но для реального роста быстродействия нужно, чтобы этот индекс был кластерным. Действительно, традиционное назначение кластерного индекса суррогатному первичному ключу в данном случае дает только то преимущество, что кластерный индекс будет наиболее компактным. Если он еще и создан по автоинкрементному полю, то с его помощью наиболее быстро будет происходить добавление данных, т. к. исключается расщепление индексных страниц *(см. разд. 5.3)*. Однако для других целей кластерный индекс по полю первичного ключа практически бесполезен — по нему не происходит соединений таблиц, и он не может быть применен в условии поиска по диапазону, запросы с которым выполняются постоянно. Поэтому стоит хотя бы рассмотреть возможность того, чтобы данные физически хранились упорядоченными по времени. Очевидно, что и в том, и в другом случае данные будут заноситься в таблицу НАЗНАЧЕНИЕ_НАДБАВКИ_2 приблизительно в порядке возрастания значений, идентифицирующих номера месяцев назначения надбавок, что является еще одним аргументом в пользу создания кластерного индекса по полю Месяц_выпл. Очевидно, будут и каким-то образом упорядочены номера приказов, но это несущественно, т. к. в сформулированных запросах нет необходимости выполнять поиск по номерам приказа.

Все приведенные ранее рассуждения подкреплены проведением теста на таблицах с одинаковым количеством записей (порядка 75 000, селективность по всем столбцам из условий поиска очень низкая). Результаты этих тестов сведены в табл. 1.9.

Решение 1. Используется таблица НАЗНАЧЕНИЕ_НАДБАВКИ.

Достоинства решения:

- таблица НАЗНАЧЕНИЕ_НАДБАВКИ имеет существенно меньший размер;
- экономится дисковое пространство;
- данные о назначении надбавки хранятся в виде, наиболее соответствующем первичному документу (приказу), на основании которого они вводятся в базу данных.

Таблица 1.9. *Сравнение результатов тестирования*

Таблица	Условие поиска	Индексы	Стоимость отбора записей
НАЗНАЧЕНИЕ_НАДБАВКИ	Таб_номер =1 and @@Мес1 >= Месяц_нач and @@Мес2 <= Месяц_оконч	Кластерный индекс по столбцу Ключ_Ид, других полезных индексов подобрать не удалось	~0,5
НАЗНАЧЕНИЕ_НАДБАВКИ_2	Таб_номер =1 and Месяц_выпл between @@Мес1 and @@Мес2	Кластерный индекс по столбцу Ключ_Ид	~0,5
НАЗНАЧЕНИЕ_НАДБАВКИ_2		Кластерный индекс по столбцу Месяц_выпл, по первичному ключу — уникальный некластерный индекс	~0,05

Недостатки решения:

- сложнее запрограммировать выполнение запросов;
- сложнее поддерживать ограничение целостности данных, так, для поддержания ограничения "Без_Перекрытия" нужен достаточно сложный триггер;
- не удалось найти способа оптимизации быстродействия запросов с использованием индексов.

Решение 2. Используется таблица НАЗНАЧЕНИЕ_НАДБАВКИ_2.

Достоинства решения:

- запросы программируются очень просто — достаточно внутреннего соединения таблиц и группировок с агрегатными функциями;
- ограничение "Без_Перекрытия" можно поддерживать с помощью уникального составного индекса по четырем полям (Надбавка_Ид, Таб_номер, Приказ, Месяц_выпл).

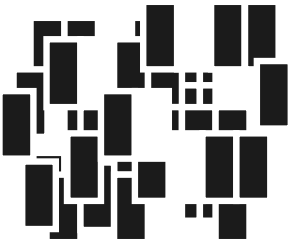
Недостатки решения:

- таблица НАЗНАЧЕНИЕ_НАДБАВКИ_2 большая, и проблема состоит не только в повышении затрат на ее хранение. Ведь запросы, которые в этом случае хоть и запрограммированы весьма прозрачно, вынуждены выполняться на больших таблицах, поэтому общее повышение быстродействия остается под вопросом;

- индекс для поддержания ограничения "Без_Перекрытия" очень громоздкий. Сложно сказать, является ли его использование достоинством по сравнению с триггером, который, разумеется, можно создать и для таблицы НАЗНАЧЕНИЕ_НАДБАВКИ_2;
- увеличивается объем базы данных.

Вывод представляется следующим — если запросы не порождают проблем, связанных с недостаточным быстродействием, то предпочтительнее использование таблицы НАЗНАЧЕНИЕ_НАДБАВКИ, т. к. на первое место выступают соображения уменьшения размера базы данных и наглядности. Если же оптимизация быстродействия необходима, то этой проблемой нужно заниматься отдельно, возможно принимая во внимание второе проектное решение.

ГЛАВА 2



Логическое проектирование и оптимизация

Целью разработки любой базы данных является хранение и использование информации о какой-либо предметной области. Если речь идет о реляционной базе данных, то для реализации этой цели имеются следующие инструменты: реляционная модель данных и язык SQL. Как уже говорилось в *главе 1*, первоначальным этапом реализации базы данных является логическое моделирование, которое чаще всего выполняется по методологии "сущность-связь".

Методика выделения сущностей и связей недостаточно формализована, чтобы для одного и того же описания предметной области, сколь угодно детализованного, разные разработчики пришли бы к одному и тому же решению. Но существуют некоторые общие стратегические принципы, которые будут присутствовать в реальных жизнеспособных проектных решениях, кто бы их ни принимал.

Очевидно, что для одной и той же предметной области существует несколько вариантов выделения сущностей, и, следовательно, реляционные отношения можно создать множеством различных способов. Более того, как мы видели в *главе 1*, даже для одинакового набора выявленных сущностей и связей реляционные отношения бывают разными. Например, можно спроектировать несколько отношений с большим количеством атрибутов или, наоборот, разнести все атрибуты по большому числу мелких отношений. Для ключей сущностей и атрибутов имеются альтернативы при выборе абстрактных доменов, не говоря уже о реальных типах данных конкретной СУБД.

Как же определить, по каким признакам нужно помещать атрибуты в те или иные отношения? Далее рассматриваются способы "хорошего", или "правильного", проектирования реляционных отношений, с точки зрения характера бизнес-среды и критериев нормальных форм отношений *(см. разд. 2.3)*.

Более подробно, чем в *главе 1*, разбирается проблема внесения избыточности в проектное решение.

2.1. Бизнес-среда: OLTP- и OLAP-системы

С точки зрения основных особенностей бизнес-среды и напрямую зависящего от них уровня нормализации данных, можно выделить два основных класса систем, основанных на реляционном подходе к представлению данных.

Сильно нормализованные модели данных хорошо подходят для так называемых *OLTP-приложений*. Типичными примерами OLTP-приложений являются системы складского учета, системы заказов билетов, банковские системы, выполняющие операции по переводу денег, и т. п. Основная функция подобных систем заключается в одновременном выполнении большого количества коротких *транзакций*. Полное определение последнего термина приведено в *главе 6*, которая посвящена подробному изучению транзакций как неотъемлемой составляющей реляционной базы данных. На данном этапе достаточно понимание сути транзакций на уровне их примеров.

Сами транзакции выглядят относительно просто, например, "снять сумму денег со счета *A* и добавить эту сумму на счет *B*", "оформить закупку партии товаров", "продать пассажиру билет на заданный поезд на заданное место на определенную дату".

Перечислим основные особенности OLTP-приложений.

1. Транзакций очень много.
2. Транзакции выполняются одновременно (к системе может быть подключено несколько тысяч пользователей, работающих в одно и то же время).
3. При возникновении ошибки транзакция должна целиком откатиться и вернуть систему к состоянию, которое было до начала транзакции (не должно быть ситуации, когда деньги сняты со счета *A*, но не поступили на счет *B*).
4. Практически все запросы к базе данных, которые должны выполняться в реальном времени, состоят из команд вставки, обновления, удаления.

Запросы на выборку в OLTP-системе в основном предназначены для предоставления пользователям возможности выбора из различных справочников, и большая часть этих запросов известна заранее еще на этапе проектирования системы. Сложные аналитические запросы если и встречаются, то время их выполнения не является существенным для работоспособности системы. Таким образом, критическими для OLTP-приложений является скорость и надежность выполнения коротких операций обновления данных. Чем выше уровень нормализации данных в OLTP-приложении, тем оно, как правило,

быстрее и надежнее. Отступления от этого правила могут происходить тогда, когда уже на этапе разработки известны некоторые часто возникающие запросы, требующие соединения отношений, от скорости выполнения которых существенно зависит работа приложения. В этом случае можно пожертвовать нормализацией для ускорения выполнения подобных запросов.

Другим типом приложений являются так называемые *OLAP-приложения* (On-Line Analitical Processing, оперативная аналитическая обработка данных). Это обобщенный термин, характеризующий принципы построения *систем поддержки принятия решений* (DSS, Decision Support System), а также *хранилищ данных* (data warehouse) и *систем интеллектуального анализа данных* (data mining). Такие системы предназначены для установления зависимостей между данными (например, можно попытаться определить, как связан объем продаж товаров с характеристиками потенциальных покупателей) или для проведения анализа, отвечающего на вопросы "что если...". OLAP-приложения оперируют большими массивами данных, уже накопленными на протяжении длительного времени с помощью OLTP-приложений, электронных таблиц или других источников данных.

OLAP-системы характеризуются следующими признаками:

- добавление в систему новых данных происходит относительно редко крупными блоками;
- данные, добавленные в систему, обычно никогда не удаляются;
- перед загрузкой данные проходят различные процедуры "очистки", связанные с тем, что в одну систему могут поступать данные из многих источников, имеющих различные форматы представления для одних и тех же понятий, данные могут быть некорректны, ошибочны;
- запросы к системе являются нерегламентированными и, как правило, достаточно сложными. Очень часто новый запрос формулируется аналитиком для уточнения результата, полученного при выполнении предыдущего запроса;
- скорость выполнения запросов важна, но не критична.

Данные OLAP-приложений обычно представлены в виде одного или нескольких *гиперкубов*, измерения которых представляют собой справочные данные, а в ячейках самого гиперкуба хранятся собственно данные. Например, можно построить гиперкуб, измерениями которого являются время (в кварталах, годах), тип товара и отделения компании, а в ячейках хранятся объемы продаж. Такой гиперкуб будет содержать данные о продажах различных типов товаров по кварталам и подразделениям. Основываясь на этих данных, можно отвечать на вопросы, сформулированные примерно следующим образом: "Каковы тенденции продаж отделений Харьковской области в текущем году по сравнению с предыдущим годом?".

Физически гиперкуб может быть построен на основе специальной *многомерной модели данных* (MOLAP, Multidimensional OLAP) или средствами реляционной модели данных (ROLAP, Relational OLAP). Таким образом, можно сказать, что в системах ROLAP данные целесообразно хранить в виде слабо нормализованных отношений, содержащих заранее вычисленные основные итоговые данные. Большая избыточность и связанные с ней проблемы тут не страшны, т. к. обновление происходит только в момент загрузки новой порции данных. При этом происходит как добавление новых данных, так и пересчет итогов.

2.2. Уровни моделирования реляционной базы данных

При разработке базы данных обычно выделяется несколько уровней моделирования, при помощи которых происходит переход от предметной области к реализации базы данных средствами конкретной СУБД. Исходным понятием в любом случае является понятие *предметной области* как части реального мира, данные о которой необходимо отразить в базе данных.

Часто упоминается концепция трехуровневого моделирования, называемая *архитектурой ANSI—SPARC* (American National Standards Institute—Systems Planning and Requirements Committee, Американский национальный институт стандартов—Комитет по системному планированию и выработке требований). В этой архитектуре выделяются три уровня моделирования:

- *внешний* — т. е. уровень представления базы данных с точки зрения пользователя;
- *концептуальный* — этот уровень описывает, какие данные хранятся в базе данных, а также какие связи имеются между этими данными;
- *внутренний* — этот уровень описывает физическое представление базы данных в компьютере, т. е. отвечает на вопрос, как информация хранится в базе данных.

Вводятся следующие понятия:

- *модель предметной области*;
- *логическая (концептуальная) модель данных*;
- *физическая модель данных*;
- *собственно база данных и приложения*.

Модель предметной области — это наши знания о предметной области, описанные с помощью некоторого формального общепринятого способа. В качестве модели могут выступать текстовые описания предметной области, набо-

ры должностных инструкций, правила ведения дел в компании и т. п. Опыт показывает, что текстовый способ представления модели предметной области крайне неэффективен. Гораздо более информативными и полезными при разработке баз данных являются описания предметной области, выполненные при помощи специализированных графических нотаций, имеющихся в достаточно большом количестве. Из наиболее известных можно назвать методику структурного анализа SADT (Structured Analysis and Design Technique, метод структурного анализа и проектирования) и основанную на ней нотацию IDEF0, диаграммы потоков данных Гейна-Сарсона, методику объектно-ориентированного анализа UML (Unified Modeling Language, универсальный язык моделирования) и др. В любом случае модель предметной области описывает процессы, происходящие в предметной области, и данные, используемые этими процессами. От того, насколько правильно смоделирована предметная область, зависит успех дальнейшей разработки приложений.

Логическая модель данных, являясь органической составляющей модели предметной области, описывает понятия предметной области в *реляционных терминах данных.* Логическая модель является начальным прототипом будущей базы данных и строится в терминах информационных единиц, но *без привязки к конкретной СУБД*. При разработке логической модели данных возникают вопросы: хорошо ли спроектированы отношения? Правильно ли они отражают предметную область? Учтены ли все ограничения целостности, налагаемые на данные? Логическая модель возникает как результат семантического моделирования *(см. главу 1)*. Основным средством разработки логической модели данных в настоящий момент являются различные варианты *ER-диаграмм*. Различают три уровня логической модели, отличающихся по глубине представления информации о данных:

- диаграмма "сущность-связь" (ERD, Entity Relationship Diagram);
- модель данных, основанная на ключах (KB, Key Based model);
- полная атрибутивная модель (FA, Fully Attributed model).

Диаграмма "сущность-связь" представляет собой модель данных верхнего уровня, отражающую основные бизнес-правила предметной области. Такая диаграмма не слишком детализирована, в нее включаются основные сущности и связи между ними, которые удовлетворяют главным требованиям, предъявляемым к базе данных. Как правило, такие модели используются для презентаций и обсуждения структуры данных с экспертами предметной области. Для представления логических моделей на этом уровне используется нотация Баркера или Чена.

Модель данных, основанная на ключах, — это более подробное представление данных. Она включает описание всех сущностей и их первичных ключей, а также всех взаимосвязей между сущностями.

Полная атрибутивная модель является наиболее детальным представлением структуры данных, включая все сущности, атрибуты и связи. Чаще всего именно этот уровень реализуется в CASE-средствах. Основные нотации для логической модели данных этого уровня — это уже известные нам IE- и IDEFX1-нотации.

Физическая модель данных. На еще более низком уровне находится физическая модель данных. Она описывает данные средствами конкретной реляционной СУБД, например MS SQL-сервер, Oracle или Access. Отношения, разработанные на стадии логического моделирования, преобразуются в таблицы, атрибуты становятся столбцами таблиц, для ключевых атрибутов создаются уникальные индексы, домены преображаются в типы данных, принятые в конкретной СУБД. Ограничения, имеющиеся в логической модели данных, реализуются различными средствами СУБД, например при помощи индексов, декларативных ограничений целостности, триггеров, хранимых процедур. Многое тут зависит от конкретной СУБД. Отношения из логической модели данных должны быть преобразованы в таблицы, но для каждой таблицы можно дополнительно объявить различные индексы, повышающие скорость обращения к данным, которые никак не обусловлены логической моделью. При разработке физической модели данных возникают вопросы: хорошо ли спроектированы таблицы? правильно ли выбраны индексы? как много программного кода в виде триггеров и хранимых процедур необходимо разработать для поддержания целостности данных?

Последним этапом является *собственно база данных и приложения*, реализованные на конкретной программно-аппаратной основе. Выбор этой основы существенно влияет на скорость работы с базой данных. Например, можно выбирать различные типы компьютеров, менять количество процессоров, объем оперативной памяти, дисковые подсистемы и т. п. Очень большое значение имеет также настройка СУБД в рамках выбранной программно-аппаратной платформы.

Очевидно, что решения, принятые на каждом этапе моделирования и разработки базы данных, будут сказываться на дальнейших этапах. Поэтому особую роль играет принятие правильных решений *на ранних этапах моделирования*.

Описанные выше уровни моделирования соотносятся с архитектурой ANSI-SPARC так, как показано на рис. 2.1.

Замечание

Вся приведенная выше иерархия уровней архитектуры и моделей весьма условна. Например, Microsoft считает достаточным введение всего двух уровней моделирования: логического и физического, не выделяя представление базы данных с точки зрения пользователя в отдельный уровень модели.

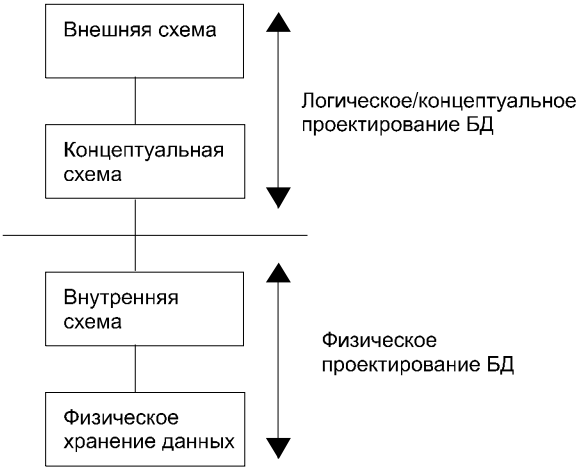


Рис. 2.1. Соответствие этапов моделирования данных и архитектуры ANSI-SPARC

2.3. Критерии оценки качества логической модели

С практической точки зрения, первым уровнем модели базы данных, с которым начинают непосредственную работу, является логическая модель. Критериев ее качества может быть очень много, и их выбор в достаточной степени произволен, но существуют критерии, безусловно важные с точки зрения получения качественной базы данных:

1. Адекватность базы данных предметной области.
2. Скорость выполнения операций обновления данных (вставка, обновление, удаление кортежей).
3. Скорость выполнения операций выборки данных.
4. Легкость разработки и сопровождения базы данных.
5. Отсутствие неоправданной избыточности данных.

В данном разделе будут рассмотрены первые четыре критерия качества логической модели базы данных. Критерий избыточности более сложен, его учет связан с постоянным выбором оптимального компромисса между противоречивыми требованиями ускорения работы запросов на выборку или транзакций, поэтому особенности влияния избыточности данных на принимаемые проектные решения будут рассматриваться подробнее в *разд. 2.4—2.6.*

2.3.1. Адекватность базы данных предметной области

Этот критерий означает, что состояние базы данных в каждый момент времени должно соответствовать состоянию предметной области. Изменение состояния предметной области обязательно приводит к соответствующему изменению состояния базы данных. Ограничения предметной области, представленные в ее модели, должны некоторым образом отражаться и учитываться в базе данных. Критерий адекватности является абсолютно необходимым. Никому не нужна база данных, которая неверно отображает предметную область.

2.3.2. Скорость операций обновления данных

Операциями, изменяющими состояние базы данных, являются вставка, обновление и удаление записей в базовых таблицах. В базах данных, требующих постоянных изменений, производительность определяется скоростью выполнения большого количества небольших операций вставки, обновления и удаления. Такие базы данных называются OLTP-системами *(см. разд. 2.5)*, и для них критерий быстродействия операций модификации данных является основным после адекватности базы данных предметной области.

На уровне логического моделирования определяются отношения и атрибуты этих отношений. Единственное, чем можно здесь управлять, — это распределение атрибутов по различным отношениям. Можно описать мало отношений с большим количеством атрибутов или много отношений, каждое из которых содержит мало атрибутов. Таким образом, необходимо попытаться ответить на вопрос: влияет ли количество отношений и атрибутов в отношениях на скорость выполнения операций обновления данных при *одинаковых подходах к физическому моделированию*.

Вставка записи производится в одну из свободных страниц памяти, выделенной для данной таблицы. СУБД постоянно хранит информацию о наличии и расположении свободных страниц. Если для таблицы не созданы индексы, то операция вставки выполняется фактически с одинаковой скоростью независимо от размера и количества атрибутов таблицы. Однако наличие индексов существенно усложняет этот процесс. Подробнее эта проблема рассматривается в *разд. 4.2.* Здесь мы ограничимся только выводом, что *чем больше атрибутов имеют отношения*, *тем медленнее будут выполняться операции обновления данных* за счет затрат времени на перестройку большего количества индексов.

2.3.3. Скорость операций выборки данных

Одно из основных назначений базы данных — предоставление информации пользователям с помощью оператора выборки `SELECT`. Известно, что наиболее дорогостоящей операцией при выполнении этого оператора является операция соединения таблиц. Таким образом, чем больше взаимосвязанных отношений было создано в ходе логического моделирования, тем больше вероятность того, что при выполнении запросов эти отношения будут соединяться, и, следовательно, тем медленнее будут выполняться запросы; поэтому увеличение количества отношений приводит к замедлению выполнения операций выборки данных. Критерий скорости выполнения операций выборки данных является доминирующим для так называемых OLAP-систем. Для таких приложений в большей степени характерны операции по выполнению сложных запросов в реальном режиме времени, а обновление данных может выполняться в периодически оговоренные моменты без работы в режиме реального времени.

2.3.4. Легкость разработки и сопровождения базы данных

Практически любая база данных содержит некоторое количество программного кода в виде триггеров и хранимых процедур.

Хранимые процедуры — это процедуры, хранящиеся непосредственно в базе данных в откомпилированном виде.

Триггеры — это те же хранимые процедуры, связанные с событиями, происходящими во время работы базы данных, такими как операции вставки, обновления и удаления строк таблиц. Если в базе данных определен некоторый триггер, то он всегда запускается *автоматически* при возникновении события, с которым этот триггер связан. Очень важным является то, что пользователь не может обойти триггер. Таким образом, основное назначение триггеров — автоматическое поддержание целостности базы данных. Триггеры могут быть как достаточно простыми, например, поддерживающими ссылочную целостность, так и довольно сложными, реализующими какие-либо сложные ограничения предметной области, затрагивающие несколько таблиц. Так, с операцией вставки нового товара в расходную накладную может быть связан триггер, который выполняет следующие действия: проверяет, есть ли необходимое количество товара, при наличии товара добавляет его в накладную и уменьшает данные о наличии товара на складе, а при отсутствии товара формирует заказ на поставку недостающего товара и тут же посылает заказ по электронной почте поставщику. Очевидно, что чем больше программного кода в виде триггеров и хранимых процедур содержит база данных, тем сложнее ее разработка и дальнейшее сопровождение.

2.4. Избыточность данных

Формальное определение термина *избыточность* базы данных дать невозможно. Но существуют общепринятые трактовки стандартных ситуаций, связанных с избыточностью. Во-первых, считается, что не рационально (избыточно) идентифицировать сущности первичными ключами из доменов, содержащих текстовые данные, даже если это соответствует логике модели. По поводу этого типа избыточности много говорилось в *главе 1*, и больше он рассматриваться не будет.

Вторая ситуация, которою трактуют как типичную ситуацию, связанную с избыточностью данных, возникает, *когда одни и те же факты можно многократно получить из разных объектов базы данных.*

Наличие избыточности может принести как пользу, так и являться недостатком, с точки зрения общей производительности системы. Основная польза от внесения избыточности в базу данных состоит в возможности ускорения выполнения запросов, т. к. часть нужной информации уже сохранена в готовом виде, и не нужно разыскивать ее по разным таблицам, осуществляя иногда достаточно сложные соединения таблиц и вычисления. Но за все нужно платить. Основной проблемой, связанной с наличием избыточности любого типа, является то, что *избыточность должна быть контролируемой.* То есть необходима программная реализация проверок того, что избыточные и базовые данные адекватно согласованы между собой. В OLTP-приложении обычно процедуру контроля избыточности выносят в транзакции, что, разумеется, замедляет их выполнение. Но с другой стороны, часто избыточные атрибуты нужно вычислять по базовым атрибутам в рамках той же самой транзакции, потому что, например, этих значений требуют входные и выходные формы транзакции.

2.4.1. Избыточность, связанная с вычислимыми атрибутами

Примером избыточности данного типа является наличие в базе данных вычислимых атрибутов и тесно связанное с ним использование ограничений. *Вычислимыми* (или *производными*) называются атрибуты, значения которых могут быть вычислены на основании других значений, хранящихся в базе данных, которые в свою очередь называются *базовыми*.

С точки зрения способов реализации, важно, вычисляются ли производные атрибуты по данным одной записи одной таблицы, или для их вычисления нужно использовать данные из разных строк или таблиц. В первом случае

можно в таблице создать вычислимый столбец, а во втором случае нужен триггер.

Например, если в таблице имеется столбец с датой рождения, то для того чтобы узнать текущий возраст человека (количество полных лет), можно создать вычислимый столбец. Если же в накладной нужно установить взаимосвязь атрибутов "`Цена*Количество = Сумма`", то скорее всего понадобится триггер, т. к. значение цены вряд ли хранится в той же самой таблице, где хранятся данные о товарных позициях накладной.

Все сказанное верно, если значения производных атрибутов вычисляются на сервере. Но их можно вычислять и на клиенте, что тоже порождает свои особенности. Далее мы рассмотрим некоторые ситуации, связанные с вычислимыми атрибутами, в предположении, что бизнес-средой является OLTP-приложение.

2.4.2. Примеры вычислимых атрибутов в простой и сложной транзакции

Пример 1. Вычислимый атрибут в простой транзакции

Классическим примером избыточности данного типа является вычисление итоговой суммы по финансовому документу. Для нашего примера базы данных для задачи материального учета *(см. ее описание в главе 5)* таким атрибутом является итоговая стоимость накладных. То есть для приходной накладной, содержащей одну товарную позицию, можно в таблицу `ПРИХОДЫ` добавить атрибут `Сумма_накл`, который равен

`Сумма_накл = Σ Кол_прих * Цена_Учетн.` (2.1)

Этот атрибут является явно избыточным, т. к. его значение просто выводится из значений других атрибутов. Реляционная теория, стремясь свести избыточность к минимуму, утверждает, что лучше хранить только два базовых атрибута `Кол_прих` и `Цена_Учетн`, а сумму вычислять во время выполнения запросов по мере необходимости. На практике однако дело обстоит сложнее. Базовыми, т. е. требующими ввода данных для приходной накладной, являются всего три атрибута: `N_прих`, `Товар` и `Кол_прих`. Кроме того, базовым для вычисления значения `Сумма_накл` является атрибут `Цена_Учетн` таблицы `ТОВАРЫ`.

Вопрос, на который нужно дать ответ, с точки зрения оптимизации производительности системы, следующий: нужно ли хранить в отношениях только базовые атрибуты или желательно хранить все атрибуты, пересчитывая значения вычислимых атрибутов каждый раз при изменении базовых.

Решение 1

Пусть в отношениях решено хранить только базовые атрибуты.

Достоинства решения:

- структура отношения полностью неизбыточна;
- не нужно дополнительного программного кода для поддержания целостности базы данных, в связи с чем транзакции по добавлению новых данных будут выполняться быстрее;
- экономится дисковое пространство, т. к. базовые таблицы содержат меньшее количество столбцов.

Недостатки решения:

- в выходных формах, на основании которых, например, печатаются документы, используются, как правило, не базовые, а вычислимые атрибуты. Одни и те же формулы применяются во многих местах, поэтому все операторы отбора данных для форм на клиентской стороне приложения будут содержать одинаковые фрагменты кода с одними и теми же формулами, в связи с чем имеется риск в разных местах вычислять одни и те же значения по *разным формулам*;
- при модификации формул для вычислений производных атрибутов необходимо изменить одни и те же фрагменты кода *везде*, где они встречаются. Это сильно затрудняет модификацию приложений;
- если возникает нерегламентированный запрос, то пользователь, формулирующий этот запрос, должен *помнить* все эти формулы.

Решение 2

В отношениях решено хранить *все атрибуты*, в том числе и вычислимые.

Достоинства решения:

- код, поддерживающий целостность кортежа и содержащий формулы для вычислимых атрибутов, хранится в одном месте, например, в триггере, связанном с данным отношением;
- при изменении логики вычислений изменения в формулы требуется внести только в одном месте, а именно, — в указанном триггере;
- запросы к базе данных содержат меньше формул и поэтому более просты;
- легче формулировать нерегламентированные запросы, т. к. в них используются атрибуты, имеющие для бухгалтера конкретный смысл.

Недостатки решения:

- при изменении логики расчета необходимость в некоторых атрибутах может исчезнуть, зато появится потребность в новых атрибутах. Это повле-

чет за собой *перестройку структуры* отношения, что является весьма болезненной операцией для работающей системы;

- структура отношения становится более сложной и запутанной;
- увеличивается объем базы данных.

Кроме перечисленных достоинств и недостатков имеется еще одно соображение, являющееся ключевым, с точки зрения производительности, а именно учет влияния наличия или отсутствия вычислимых атрибутов на трафик сети. К сожалению, нельзя однозначно определить, в каком из двух рассмотренных случаев загрузка сети уменьшится.

Очевидно, что оба решения имеют как свои достоинства, так и недостатки. Важно то, что программный код, содержащий формулы для вычисления производных атрибутов, не исчезает ни в каком из этих решений (да и куда он денется, раз такова предметная область!). Только в одном случае код хранится в одном месте, а в другом может быть "распределен" по всему приложению.

На практике вычислимые атрибуты всегда используются в той или иной степени. Не последним вопросом в связи с этим является выбор места хранения кода для вычислимых атрибутов. Этим местом может быть как сервер, так и клиент, что в каждом конкретном случае определяет свои особенности.

Если выбрано решение с использованием вычислимых полей таблицы, то формулы для них хранятся и выполняются только на сервере. Например, в таблицу `ПРИХОДЫ` можно добавить вычислимый столбец, который содержит значение суммы накладной. Рассмотрим, как в этом случае будет выполняться транзакция по формированию приходной накладной. Эта транзакция не требует никаких проверок, порождающих возможность отката. Просто на клиенте вводится название товара (или его идентификатор), а также его количество, после чего определяется цена товара. Определение этой цены невозможно без запроса к серверу, потому что нужно проверить ограничение внешнего ключа для таблиц `ПРИХОДЫ` и `ТОВАРЫ`. После этого на клиенте уже есть все необходимые данные для того, чтобы в форме, с которой работает пользователь, вывести значение суммы накладной. Если этого избыточного атрибута нет вообще, то транзакция фиксируется, на клиенте в локальную переменную вычисляется сумма накладной и печатается документ. Если же избыточный атрибут с суммой накладной введен и это значение решено вычислять на клиенте, то значение суммы накладной участвует в транзакции, транзакция фиксируется и печатается документ. Ограничения целостности базы данных, связанного с формулой (2.1), не возникает, в базе данных могут оказаться и рассогласованные данные.

Но если вычисление значения `Сумма_накл` происходит по формуле, привязанной к столбцу базовой таблицы, то для его получения необходимо сначала

зафиксировать транзакцию, чтобы сработал триггер, вычисляющий значение суммы накладной, после чего можно послать к серверу простой запрос на извлечение значения вычислимого поля из вновь вставленной строки, а затем напечатать форму. В последнем случае в базе данных поддерживается ограничение целостности, описанное формулой (2.1).

Какое из решений лучше, сказать сложно. Прежде всего нужно понимать, для чего используется значение `Сумма_накл`, — если только для выписки накладной, то, разумеется, его вводить как столбец таблицы необязательно. В основном проектном решении для задачи материального учета *(см. главу 5)* столбцы для вычисления сумм накладных не введены.

Пример 2. Вычислимые атрибуты в сложной транзакции

Теперь рассмотрим ситуацию, когда выполняется транзакция, потенциально порождающая возможность отката, а запрос, вычисляющий избыточные данные, очень ресурсоемок. В нашей базе данных эта ситуация характерна для транзакции формирования расходной накладной, при которой возможен откат транзакции в случае попытки продать товары в количестве, большем, чем их остаток, не говоря уже о попытке продажи несуществующего товара (*FK*-ограничение).

Без наличия поля, содержащего остатки товара, в данном случае не обойтись. В торговой OLTP-системе выписка расходной накладной — это основная бизнес-операция. Посмотрим, что получится, если поля `Остаток` в таблице `ТОВАРЫ` нет вообще. На клиенте заполняется форма, содержащая товарные позиции расходной накладной. При этом вводятся базовые атрибуты `N_расх`, `Товар` и `Кол_расх`. На сервер посылается запрос на проверку ограничения внешнего ключа. Считаем, что эта проверка успешно пройдена, и теперь нужно выполнить проверку соответствия значений `Кол_расх` остаткам товаров. Если в базе данных нет готового значения для этих остатков, то, чтобы их получить, нужно выполнять запрос из листинга 5.31 либо 5.32. Эти запросы очень ресурсоемки, хотя и создано индексное представление для ускорения их выполнения. Кроме того, время, нужное для выполнения этих запросов, напрямую зависит от размера базовых таблиц. Очевидно, что выполнение подобных запросов в рамках основной транзакции OLTP-приложения нереально. Поэтому поле `Остаток` в таблице `ТОВАРЫ` обязательно должно существовать. В этом случае запрос к серверу на получение остатков введенных товаров будет максимально простым и практически не будет зависеть от размеров базовых таблиц.

Итак, пусть на клиенте введены данные о товарных позициях расходной накладной, получены значения остатков введенных товаров из соответствующих записей таблицы `ТОВАРЫ`, после чего стартовала транзакция по формиро-

ванию расходной накладной. Теперь рассмотрим, где лучше вычислять значение для поля остатка — в триггере или на клиенте. Возможны следующие варианты действий:

1. Происходит попытка зафиксировать транзакцию без всяких вычислений на клиенте. В этом случае должен быть создан триггер, который выполнит все необходимые проверки и в случае успеха изменит значения остатка каждого проданного товара в соответствии со значением `Кол_расх`. После успешной фиксации транзакции можно выводить значения стоимости расходной накладной отдельно по товарным позициям и в целом, а затем печатать документ. Если же проверки в триггере не сработали, то из этого триггера происходит откат транзакции, а возникшая исключительная ситуация должна быть обработана на клиенте. Примерный код триггера для этого случая приведен в листинге 5.39.
2. На клиенте проверяется соответствие остатков товаров введенным значениям `Кол_расх`. Если эта проверка не завершилась успехом, то на клиенте формируется соответствующее сообщение и транзакция откатывается. При успешной проверке транзакция фиксируется, выводятся значения стоимости расходной накладной и печатается документ. При фиксации транзакции все равно нужно изменять значения остатков проданных товаров. Это можно сделать из триггера, но теперь в нем уже нет необходимости проверять возможность отката транзакции. Можно обойтись и без триггера, воспользовавшись вычисленными на клиенте значениями остатков и включив запрос на изменение таблицы `ТОВАРЫ` в соответствии с этими значениями в транзакцию, которая должна быть зафиксирована только после выполнения этого запроса.

Замечание

Наличие триггера для второго варианта позволит поддерживать целостность базы данных по согласованию остатков товаров с их отпуском. Этот триггер прост — он может быть получен из триггера, приведенного в листинге 5.39, удалением логической ветви, связанной с проверкой возможности отката.

2.5. Критерий нормализации

Нормализация в самом обобщенном понимании — это метод создания набора отношений с заданными свойствами на основании требований к данным, установленным в некоторой предметной области. Понятие нормализации введено Э. Ф. Коддом в начале 1970-х гг., и в настоящее время под нормализацией реляционной базы данных понимают формальный процесс удаления избыточных данных, основанный на определенных математических понятиях реляционной теории.

Обычно нормализация выполняется в виде последовательности тестов для некоторого отношения с целью его проверки на соответствие формальным требованиям той или иной нормальной формы.

Далее рассматривается проблема избыточности реляционной базы данных, связанная с ее недостаточной нормализацией, и пути преодоления указанной проблемы для OLTP-бизнес среды.

2.5.1. Избыточность, связанная с нормализацией

Начнем с примера. Нам уже известно проектное решение для задачи материального учета, описанное в *главе 5*. Приведем небольшую его часть:

```
ПРИХОДЫ(N_прих, Товар_Ид, Кол_прих, Дата_прих)
ТОВАРЫ(Товар_Ид, Товар, Цена_учетн, Остаток)
```

Чтобы понять, почему выбраны именно такие отношения, попробуем изменить проектное решение определенным образом, и посмотрим, к чему это приведет. Например, уменьшим количество отношений, выполнив *естественное соединение (см. разд. 5.5)* отношений ПРИХОДЫ и ТОВАРЫ с помощью оператора SELECT по их общему полю Товар_Ид. В результате получим отношение:

```
ПРИХОДЫ_ТОВАРЫ(N_прих, Товар_Ид, Кол_прих, Дата_прих, Цена_учетн).
```

Очевидно, что в отношении ПРИХОДЫ_ТОВАРЫ есть место для размещения *всех* данных из отношений ПРИХОДЫ и ТОВАРЫ. Кроме того, ясно, что в отношении ПРИХОДЫ_ТОВАРЫ наблюдается избыточность по атрибуту Цена_учетн, т. к. *одно и то же* значение учетной цены можно получить из *нескольких записей* отношения ПРИХОДЫ_ТОВАРЫ. Рассмотрим, к каким проблемам ведет создание базовой таблицы на основе отношения ПРИХОДЫ_ТОВАРЫ вместо пары отношений ПРИХОДЫ и ТОВАРЫ.

- *Обновление*. Пусть, например, необходимо изменить цену какого-либо товара, заданного с помощью его идентификатора. Для этого в таблице ПРИХОДЫ_ТОВАРЫ нужно найти все записи о приходных накладных с этим товаром и в каждой из них *согласованным образом изменить цену товара*.
- *Вставка*. Если нужно вставить в таблицу ПРИХОДЫ_ТОВАРЫ данные о цене нового товара, который не занесен ни в одну из приходных накладных, то придется создавать запись с NULL-значениями в полях N_прих, Кол_прих и Дата_прих.
- *Удаление*. Если удалить записи о каких-то приходных накладных, например, по истечении срока хранения этих документов, то цена товара может

потеряться в том случае, когда этого товара не окажется ни в одной из приходных накладных, не подлежащих удалению.

Описанные проблемы связаны с плохой нормализацией отношения ПРИХОДЫ_ТОВАРЫ. Исторически эти проблемы получили название *аномалии модификации данных*. Попытки дать строгое определение понятия аномалии этого типа не являются вполне удовлетворительными без использования терминологии нормальных форм. Будем придерживаться интуитивного понятия аномалии как неадекватности модели данных предметной области, что говорит на самом деле о том, что логическая модель данных либо попросту неверна, либо необходимы дополнительные усилия для реализации всех ограничений, определенных в предметной области. Так, для поддержания отношения ПРИХОДЫ_ТОВАРЫ в целостном состоянии придется затратить дополнительные усилия на создание триггеров, расходовать дисковое пространство на хранение избыточных данных, неизбежно вводить NULL-значения.

Все эти проблемы могут быть решены еще на стадии логического моделирования, если применять специальные методики нормализации реляционной базы данных. Для устранения аномалий обновления (а на самом деле *для правильного проектирования модели данных*!) применяется метод нормализации отношений, основанный на понятиях *функциональной зависимости* (ФЗ) и *нормальных форм* (НФ).

2.5.2. Нормальные формы отношений

Первая нормальная форма (1 НФ) — это обычное отношение, удовлетворяющее определению, но с одной оговоркой. В определении отношения подчеркивается, что домены, на которых определены атрибуты отношения, не обязательно различны. С последним утверждением и связана упомянутая оговорка. Первая нормальная форма имеет место, когда каждый из атрибутов отношения является атомарным (неделимым, скалярным) и не содержит повторяющихся групп.

Атомарность атрибутов, с практической точки зрения, следует понимать таким образом, что не должно присутствовать запросов, которые требуют выдачи информации из части поля. Если необходимы, например, сведения о номере факультета, на котором обучается студент, то этот номер нужно не извлекать из номера группы, а ввести для номера факультета отдельное поле.

Повторяющиеся группы — это атрибуты, определенные на одних и тех же доменах. Бывает, что избежать этого нельзя, например для рекурсивных (ролевых) связей *(см. главу 1)*. Суть возникновения избыточности данных, обусловленной повторяющимися группами, связана с понятием первой нормальной формы. Пример избыточности этого типа приведен в *разд. 2.6*.

2.5.3. Функциональные зависимости

Пусть R — отношение. Множество атрибутов Y ($Y \in R$) *функционально зависимо* от множества атрибутов X ($X \in R$) тогда и только тогда, когда *для любого состояния* отношения R во всех кортежах, имеющих одинаковые значения атрибутов X, значения атрибутов Y также совпадают.

Символически функциональная зависимость записывается как $X \rightarrow Y$. Множество атрибутов X называется *аргументом функциональной зависимости*, а множество атрибутов Y называется *зависимой частью*.

Другими словами это определение можно дать так.

Функциональная зависимость — это такая зависимость между двумя атрибутами (множествами атрибутов) A и B, когда в любой момент времени для любого значения атрибута A (аргумента ФЗ) существует одно и только одно значение атрибута B (функции). Атрибут A (аргумент ФЗ) может быть как атомарным, так и составным.

Графическое изображение ФЗ, принятое в реляционной алгебре, выглядит, как показано на рис. 2.2.

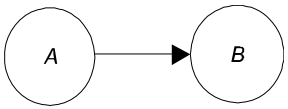


Рис. 2.2. Графическое изображение ФЗ

Если существует ФЗ $A \rightarrow B$ и B не зависит функционально ни от какого подмножества A, то говорят, что A представляет собой *детерминант* для B. Чаще всего понятия аргумент ФЗ и детерминант ФЗ совпадают.

Замечание

ФЗ не могут быть *выведены* из внешнего вида отношения в конкретный момент времени. Они отражают взаимосвязи, обнаруженные между объектами предметной области, и являются дополнительными ограничениями, определяемыми предметной областью. Таким образом, ФЗ — это *семантическое понятие*. Она возникает, когда по значениям одних данных в предметной области можно однозначно определить значения других данных. ФЗ *задает дополнительные ограничения* на данные, которые могут храниться в отношениях.

ФЗ между атрибутами отношения напоминает понятие функциональной зависимости в математике. Но это не одно и то же. Для сравнения напомним математическое понятие функциональной зависимости.

В математике под функциональной зависимостью (функцией) понимают тройку объектов (X, Y, f), где X — множество *(область определения)*, Y — множество *(множество значений)*, f — правило, согласно которому каждому

элементу $x \in X$ ставится в соответствие один и только один элемент $y \in Y$ *(правило функциональной зависимости)*.

Определения ФЗ для атрибутов отношения тоже *напоминает* это определение. Отличие же от математического определения ФЗ состоит в том, что если рассматривать математическое понятие функции, то *для фиксированного значения* $x \in X$ соответствующее значение функции $y = f(x)$ *всегда одно и то же*. Например, если задана функция $y = x^2$, то для значения $x = 2$ соответствующее значение y *всегда* будет равно 4. В противоположность этому в отношениях значение зависимого атрибута может быть *различным* в различных состояниях базы данных.

Таким образом, понятие ФЗ атрибутов нельзя считать полностью эквивалентным математическому понятию функциональной зависимости, т. к. значения этой зависимости различны при разных состояниях отношения и, самое главное, эти значения могут меняться *непредсказуемо*. ФЗ атрибутов утверждает лишь то, что для каждого конкретного состояния базы данных по значению одного атрибута (аргумента) можно однозначно определить значение другого атрибута (зависимой части). Но конкретные значения зависимой части могут быть различны в разных состояниях базы данных.

Как уже говорилось, ФЗ — это *семантическое понятие.* Однако имеются средства проверки того обстоятельства, что определенные данные в конкретных столбцах таблицы SQL-сервера находятся в состоянии, удовлетворяющем определению ФЗ. В листинге 2.1 приведен оператор `SELECT`, который для таблицы с именем `TABLE_` проверяет, удовлетворяют ли данные в ее столбцах `COLUMN1` и `COLUMN2` определению ФЗ.

Листинг 2.1. Проверка наличия ФЗ COLUMN1 → COLUMN2 с помощью языка SQL

```
SELECT COUNT(DISTINCT COLUMN1), COUNT(DISTINCT COLUMN2)
FROM TABLE_
GROUP BY COLUMN1
HAVING COUNT(DISTINCT COLUMN1)<>COUNT (DISTINCT COLUMN2)
```

Запрос из этого листинга возвращает те строки, которые нарушают определение ФЗ. Соответственно, если этот запрос не выдал никаких строк, то это означает, что определение ФЗ выполняется.

2.5.4. Вторая нормальная форма

Отношение находится *во второй нормальной форме* (2 НФ) тогда и только тогда, когда отношение находится в 1 НФ и в отношении нет *неключевых ат-*

рибутов, функционально зависящих от части сложного потенциального ключа отношения.

Неключевой атрибут — это атрибут, не входящий в состав никакого потенциального ключа. Иными словами, 2 НФ имеет место, когда отношение находится в 1 НФ и все его неключевые атрибуты функционально зависят от ключа отношения.

Отношение `ПРИХОДЫ_ТОВАРЫ` находится в 2 НФ по тривиальной причине — оно имеет атомарный ключ, однако в нем имеются аномалии модификации данных. Отсюда следует вывод о том, что 2 НФ недостаточно для адекватного проектирования реляционной базы данных.

2.5.5. Третья нормальная форма (НФ Бойса — Кодда)

Отношение находится в *третьей нормальной форме* (3 НФ) тогда и только тогда, когда оно находится в 2 НФ и *все его неключевые атрибуты взаимно функционально независимы,* т. е. ни один из них не является функционально зависимым от другого.

Нормальная форма Бойса — Кодда (НФБК) вводится в рассмотрение, когда у отношения имеется более одного потенциального ключа, и имеет место в отношении, в котором наблюдается полное совпадение возможных ключей и детерминантов. На рис. 2.3 в графическом виде приведено различие между 2 НФ и 3 НФ.

Замечание

Если отношение находится в НФБК, то оно автоматически находится и в 3 НФ, как это сразу следует из ее определения.

Отношение `ПРИХОДЫ_ТОВАРЫ` не находится в 3 НФ, т. к. существует ФЗ (`Товар` → `Цена_учетн`), а атрибут `Товар` не является ключом отношения `ПРИХОДЫ_ТОВАРЫ`.

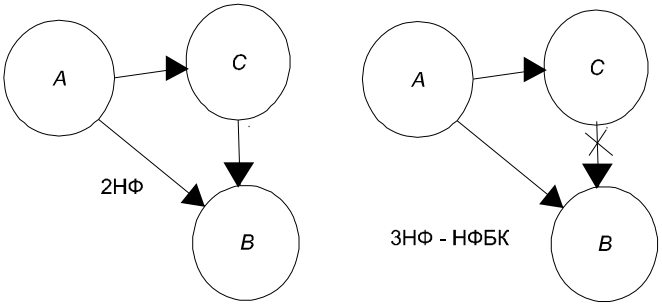


Рис. 2.3. Различие между 2 НФ и 3 НФ

2.5.6. Правила вывода ФЗ. Минимальное покрытие

Одна из проблем, связанная с применением ФЗ, заключается в том, что обычно при использовании формального определения удается выявить гораздо больше ФЗ, чем нужно для корректной нормализации. По этой причине оставляют минимально возможный набор ФЗ, называемый *минимальным покрытием*, который обеспечивает корректное проектирование. Выбор ФЗ для минимального покрытия основан на использовании такого аппарата реляционной алгебры, как *правила вывода ФЗ*. Рассмотрим простейшие правила вывода, позволяющие удалять *избыточные* ФЗ.

1. Транзитивность ФЗ — если $A \to B$, $B \to C$, то $A \to C$.
2. Добавление ФЗ — если $A \to B$, то A, $C \to B$.
3. Объединение ФЗ — если $A \to B$ и $A \to C$, то $A \to B$, C.

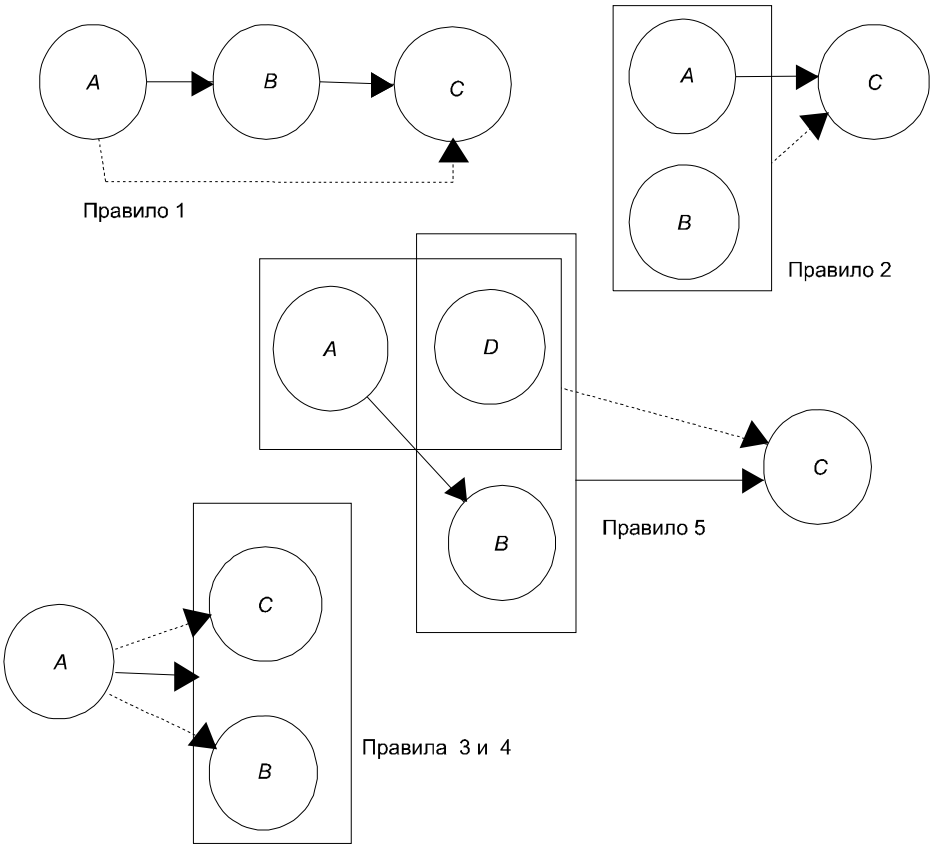


Рис. 2.4. Иллюстрация правил вывода ФЗ

4. Декомпозиция ФЗ — если $A \to B, C$, то $A \to B$ и $A \to C$.
5. Псевдотранзитивность ФЗ — если $A \to B$ и $B, D \to C$, то $A, D \to C$.

Графически эти правила вывода проиллюстрированы на рис. 2.4. Для правил 1, 2 и 5 пунктиром показаны избыточные ФЗ, подлежащие удалению. Правила 3 и 4 являются взаимно обратными и позволяют выбрать одну из комбинаций ФЗ (сплошную стрелку либо две пунктирные).

Обычно в OLTP-приложениях довольствуются приведением реляционной базы данных к 3 НФ. Далее на примерах эта процедура будет подробно рассмотрена. Однако в теории существуют еще четвертая и пятая нормальная формы, минимальные сведения о которых будут рассмотрены на примерах без строгих определений.

2.5.7. Четвертая нормальная форма

Четвертая нормальная форма (4 НФ), являясь достаточно сложной, не считается обязательной на практике во всех ситуациях, поэтому обойдемся при ее рассмотрении без строгих математических определений. Рассмотрим следующий пример. Пусть требуется учитывать данные об абитуриентах, поступающих в ВУЗ. При анализе предметной области были выделены следующие требования:

1. Каждый абитуриент имеет право сдавать экзамены на несколько факультетов одновременно.
2. Каждый факультет имеет свой список сдаваемых предметов.
3. Один и тот же предмет может сдаваться на нескольких факультетах и при этом перезачет оценок по одноименным предметам на разных факультетах не допускается. То есть абитуриент обязан сдавать все предметы, указанные для факультета, на который он поступает, несмотря на то, что он, может быть, уже сдавал такие же предметы на другом факультете.

Предположим, что нам требуется хранить данные о том, какие предметы должен сдавать каждый абитуриент. Попытаемся хранить данные в одном отношении ЭКЗАМЕНЫ (табл. 2.1).

Таблица 2.1. *Отношение* ЭКЗАМЕНЫ

Абитуриент	**Факультет**	**Предмет**
Иванов	Математический	Математика
Иванов	Математический	Информатика

Таблица 2.1 *(окончание)*

Абитуриент	**Факультет**	**Предмет**
Иванов	Физический	Математика
Иванов	Физический	Физика
Петров	Математический	Математика
Петров	Математический	Информатика

Ключ отношения ЭКЗАМЕНЫ содержит все три его атрибута.

В данный момент в отношении хранится информация о том, что абитуриент Иванов поступает на два факультета (математический и физический), а абитуриент Петров — только на математический факультет. Кроме того, можно сделать вывод о том, что на математическом факультете нужно сдавать математику и информатику, а на физическом — математику и физику.

В отношении ЭКЗАМЕНЫ имеются аномалии обновления, вставки и удаления, которые заключаются в том, что ни одну из этих операций нельзя выполнить для единственного кортежа отношения без учета других его кортежей.

- *Аномалия вставки*. При попытке добавить в это отношение новый кортеж, например (Сидоров, Математический, Математика), необходимо добавить и кортеж (Сидоров, Математический, Информатика), т. к. все абитуриенты математического факультета *обязаны* иметь один и тот же список сдаваемых предметов.
- *Аномалия удаления.* При попытке удалить кортеж (Иванов, Математический, Математика) необходимо удалить и кортеж (Иванов, Математический, Информатика) по той же самой причине. Кроме того, если удалить кортеж (Иванов, Физический, Математика), а вместе с ним и кортеж (Иванов, Физический, Физика), то будет потеряна информация о предметах, которые должны сдаваться на физическом факультете.
- *Аномалия обновления*. При попытке изменить список экзаменов, сдаваемых на факультете, нужно это делать для всех абитуриентов.

Таким образом, вставка и удаление кортежей не могут быть выполнены независимо от других кортежей отношения.

Однако для нормализации отношения ЭКЗАМЕНЫ нельзя использовать нормальные формы, основанные на ФЗ, т. к. это отношение, являясь полностью ключевым, *не содержит никаких ФЗ*. Но ясно, что какая-то связь между его атрибутами все-таки имеется. Эта взаимосвязь описывается термином *многозначной зависимости,* на понятии которой основано определение 4 НФ.

Отношение R находится в 4 НФ тогда и только тогда, когда отношение находится в НФБК и не содержит *нетривиальных многозначных зависимостей*, т. е. таких, что для многозначной зависимости, обозначаемой как $(X \rightarrow\rightarrow Y|Z)$, не существует функциональных зависимостей $(X \rightarrow Y)$ и $(X \rightarrow Z)$.

Отношение ЭКЗАМЕНЫ находится в НФБК, но не в 4 НФ, и его можно без потерь декомпозировать на два отношения, используя понятие многозначных зависимостей. Эти отношения следующие:

КТО_КУДА_ПОСТУПАЕТ(Факультет, Абитуриент)
ГДЕ_ЧТО_СДАВАТЬ(Факультет, Предмет)

В полученных отношениях устранены аномалии вставки и удаления, характерные для отношения ЭКЗАМЕНЫ. При этом полученные отношения находятся в 4 НФ, оставаясь полностью ключевыми, и в них по-прежнему нет функциональных зависимостей.

Отношения с нетривиальными многозначными зависимостями возникают, как правило, в результате естественного соединения двух отношений по общему полю, которое *не является ключевым ни в одном из отношений*. Фактически это приводит к попытке хранить в одном отношении информацию о двух *независимых* сущностях. В нашем случае список экзаменов, которые сдаются на разных факультетах, никак не зависит от того, какие абитуриенты куда поступают.

Замечание

Если окажется необходимым хранить оценки, полученные при сдаче экзаменов абитуриентами, то отношение ЭКЗАМЕНЫ придется оставить в исходном состоянии, добавив к нему поле Оценка, потому что больше эту оценку размещать негде. Чтобы получить отношение для хранения оценок, нужно выполнить естественное соединение отношений КТО_КУДА_ПОСТУПАЕТ и ГДЕ_ЧТО_СДАВАТЬ по полю Факультет, вернувшись к отношению ЭКЗАМЕНЫ, после чего появится возможность заносить оценки, которые функционально зависят от составного ключа отношения ЭКЗАМЕНЫ.

В качестве еще одного примера 4 НФ можно привести ситуацию, когда сотрудник может иметь много мест работы и много детей. Хранение информации о местах работы и детях в одном отношении приводит к возникновению нетривиальной многозначной зависимости (Работник $\rightarrow\rightarrow$ Работа|Дети).

2.5.8. Пятая нормальная форма

Функциональные и многозначные зависимости позволяют произвести декомпозицию исходного отношения без потерь на *две* проекции. Можно, однако, привести примеры отношений, которые нельзя декомпозировать без потерь

ни на какие две проекции. Следует отметить, что придумать пример, связанный с реальной предметной областью, для 5 НФ крайне сложно. Однако для полноты картины рассмотрим следующий абстрактный пример для отношения *R* (табл. 2.2—2.4).

Таблица 2.2. *Отношение R*

X	Y	Z
1	1	2
1	2	1
2	1	1
1	1	1

Всевозможные проекции отношения *R*, включающие по два атрибута, имеют следующий вид (табл. 2.3).

Таблица 2.3. *Проекции отношения R*

Проекция *R*1=*R*[*X*,*Y*]	
X	**Y**
1	1
1	2
2	1

Проекция *R*2=*R*[*X*,*Z*]	
X	**Z**
1	2
1	1
2	1

Проекция *R*3=*R*[*Y*,*Z*]	
Y	**Z**
1	2
2	1
1	1

Таблица 2.4. *Соединение отношений R1 и R2*

***R*1 JOIN *R*2**		
X	***Y***	***Z***
1	1	2
1	1	1
1	***2***	***2***
1	2	1
2	1	1

Как легко заметить, отношение *R* не восстанавливается ни по одному из *попарных естественных* соединений *R*1 JOIN *R*2, *R*1 JOIN *R*3 или *R*2 JOIN *R*3.

Действительно, соединение $R1$ JOIN $R2$ имеет вид, показанный в табл. 2.4. Курсивом выделен лишний кортеж, отсутствующий в отношении R. Аналогично (в силу соображений симметрии) и другие попарные соединения не восстанавливают отношения R.

Однако отношение R восстанавливается соединением *всех трех* проекций $R1$ JOIN $R2$ JOIN $R3 = R$. Это говорит о том, что между атрибутами этого отношения также имеется некоторая зависимость, но она не является ни функциональной, ни многозначной зависимостью. Зависимости подобного типа называются *зависимостями соединения* и являются обобщением понятия многозначной зависимости.

Другими словами, под зависимостью соединения понимают такую зависимость, которая при обратном естественном соединении зависимых отношений порождает ложные строки, т. е. является нетривиальной.

Говорят, что отношение R находится в *пятой нормальной форме* (5 НФ) тогда и только тогда, когда *любая имеющаяся зависимость соединения является тривиальной* или, наоборот, отношение R *не находится в 5 НФ*, если в отношении *найдется нетривиальная зависимость соединения*.

Если вернуться к примеру, становится понятно, что не зная ничего о том, какие потенциальные ключи имеются в отношении и как взаимосвязаны атрибуты, нельзя сделать выводы о том, находится ли данное отношение в 5 НФ (как, впрочем, и в других НФ). По данному конкретному примеру можно только предположить, что отношение не находится в 5 НФ. Допустим, что анализ предметной области позволил выявить следующие зависимости атрибутов в отношении R. Отношение R является полностью ключевым. Имеется следующая зависимость (довольно странная, с практической точки зрения): если в отношении R содержатся кортежи $r_1 = (x, y, z_1)$, $r_2 = (x, y_1, z)$ и $r_3 = (x_1, y, z)$, то отсюда следует, что в отношении R содержится также и кортеж $r_4 = (x, y, z)$.

С учетом приведенных ранее утверждений отношение из рассмотренного примера находится в 4 НФ, но не в 5 НФ.

2.5.9. Алгоритм декомпозиции отношения с целью его нормализации

Декомпозиция — это процесс разбиения отношения, которое не находится в НФ нужного порядка, на несколько отношений, каждое из которых находится в нужной НФ, путем применения к исходному отношению оператора реляционной алгебры ПРОЕКЦИЯ.

Оператор реляционной алгебры ПРОЕКЦИЯ применяется к одному существующему отношению для получения нового отношения путем выбора (проециро-

вания) определенных столбцов из текущего отношения. Если результат выполнения операции ПРОЕКЦИЯ содержит повторяющиеся кортежи (строки), то оставляется только один кортеж из повторяющейся группы.

Использование языка SQL позволяет выполнить оператор реляционной алгебры ПРОЕКЦИЯ следующим образом (листинг 2.2).

Листинг 2.2. Оператор реляционной алгебры ПРОЕКЦИЯ на SQL

```
SELECT DISTINCT COLUMN1, COLUMN2 INTO TABLE_2
FROM TABLE_1
ALTER TABLE TABLE_1 DROP COLUMN COLUMN2
-- иначе
SELECT COLUMN1, COLUMN2 INTO TABLE_2
FROM TABLE_1
GROUP BY COLUMN1, COLUMN2
ALTER TABLE TABLE_1 DROP COLUMN COLUMN2
```

Оператор ПРОЕКЦИЯ выполняется над таблицей TABLE_1 по ее столбцам COLUMN1 и COLUMN2. Обязательно указание либо опции DISTINCT, либо опции GROUP BY для автоматического исключения дублирования кортежей в результирующем отношении TABLE_2. После выполнения проекции необходимо определить ключ отношения TABLE_2 и задать его.

В реляционной базе данных для OLTP-приложений обычно выполняется декомпозиция до 3 НФ, алгоритм которой включает в себя следующие шаги.

1. *Приведение к 1 НФ*. Отношения, включенные в модель, анализируются на предмет соответствия 1 НФ. Из отношений удаляются повторяющиеся группы, уточняется перечень доменов базы данных. Проверяется скалярность атрибутов, с точки зрения основанных на них запросов. Нескалярные атрибуты представляются в виде нескольких скалярных атрибутов.
2. *Выявление ФЗ и построение минимального покрытия.* Строится и обосновывается минимальное покрытие для найденных ФЗ между атрибутами.
3. *Приведение ко 2 НФ*. Если в некоторых отношениях обнаружена функциональная зависимость атрибутов от части сложного ключа, то проводится декомпозиция этих отношений на несколько отношений следующим образом: те атрибуты, которые зависят от части сложного ключа, выносятся в отдельное отношение вместе с этой частью ключа. В исходном отношении остаются все ключевые атрибуты. Например, пусть исходное отношение $R(K_1, K_2, A_1, ..., A_n, B_1, ..., B_m)$ имеет сложный ключ (K_1, K_2), а также содержит две ФЗ: $(K_1, K_2) \rightarrow (A_1, ..., A_n, B_1, ..., B_m)$ и $(K_1) \rightarrow (A_1, ..., A_n)$. Тогда де-

композированные отношения имеют вид: $R1(\underline{K_1, K_2}, B_1, ..., B_m)$ с ключом (K_1, K_2) и $R2(\underline{K_1}, A_1, ..., A_n)$ с ключом K_1.

4. *Приведение к 3 НФ*. Если в каких-то отношениях обнаружена зависимость некоторых неключевых атрибутов от других неключевых атрибутов, то проводим декомпозицию этих отношений следующим образом: те неключевые атрибуты, которые зависят от других неключевых атрибутов, выносятся в отдельное отношение. В новом отношении ключом становится детерминант функциональной зависимости. То есть пусть исходное отношение $R(\underline{K}, A_1, ..., A_n, B_1, ..., B_m)$ имеет ключ K и две ФЗ $K \rightarrow (A_1, ..., A_n, B_1, ..., B_m)$ — зависимость всех атрибутов от ключа отношения, а также $(A_1, ..., A_n) \rightarrow (B_1, ..., B_m)$ — зависимость некоторых неключевых атрибутов от других неключевых атрибутов. Декомпозированные отношения: $R1(\underline{K}, A_1, ..., A_n)$ — остаток от исходного отношения с ключом K и $R2(\underline{A_1, ..., A_n}, B_1, ..., B_m)$ с ключом $(A_1, ..., A_n)$, куда вынесены атрибуты из исходного отношения вместе с детерминантом ФЗ.

При необходимости алгоритм нормализации можно продолжить и выполнить следующие шаги:

1. *Приведение к 4 НФ*. Если в отношениях обнаружены нетривиальные многозначные зависимости, то необходимо провести декомпозицию для исключения таких зависимостей.

2. *Приведение к 5 НФ*. Если в отношениях обнаружены нетривиальные зависимости соединения, то необходимо провести декомпозицию для исключения и таких зависимостей.

Согласно этому алгоритму, обычно декомпозицию начинают с графического изображения всех ФЗ, которые удалось выявить в предметной области. При этом стрелки следует рисовать от наборов атрибутов, образующих детерминанты ФЗ, тем самым сразу учитывая правило вывода 2. Следующий этап декомпозиции — выделение минимального покрытия на основании правил вывода. Декомпозиция считается корректной, если после ее выполнения проектное решение оказалось, во-первых, декомпозированным на отношения, находящиеся в НФБК, и, во-вторых, ни одна из ФЗ минимального покрытия не потеряна и не появляется более чем в одном результирующем отношении.

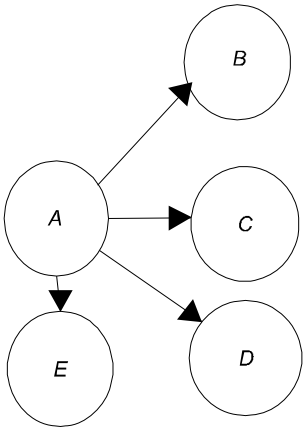
Начинать декомпозицию следует с "краев" рисунка. При выполнении декомпозиции целесообразно руководствоваться следующими правилами.

Правило "цепочки" — если отношение $R(\underline{A}, B, C)$ содержит цепочки ФЗ вида $(A \rightarrow B)$, $(B \rightarrow C)$, $(A \rightarrow C)$, то $R(\underline{A}, B, C)$ следует декомпозировать на два отношения $R1(\underline{A}, B)$ и $R2(\underline{B}, C)$. В реляционной алгебре декомпозиция по правилу "цепочки" реализуется оператором проекции отношения $R(A, B, C)$.

Обобщая правило "цепочек" на более сложные случаи, когда явных "цепочек" нет, можно выполнять декомпозицию, помещая все ФЗ с разными детерминантами в отдельные отношения.

Нельзя для декомпозиции выбирать ФЗ, зависимая часть которой сама — целиком или частично — является детерминантом другой ФЗ.

ФЗ с одинаковым детерминантом нужно естественным образом помещать в одно отношение. Однако совершенно корректно решение (см. правила вывода 3 и 4), когда такие ФЗ помещены в разные отношения. Атрибуты, образующие детерминант, должны присутствовать в каждом отношении. Каждый из зависимых атрибутов следует поместить только в одно отношение, сгруппировав их удобным образом. Пусть, например, имеются наборы ФЗ (рис. 2.5). Корректные наборы отношений для этого случая приведены на рис. 2.5. При этом все отношения находятся в НФБК.



R(*A*, *B*, *C*, *D*, *E*)

R1(*A*, *B*, *C*) и R2(*A*, *D*, *E*)

R1(*A*, *B*), R2(*A*, *C*) и R3(*A*, *D*, *E*)

И так далее...

Рис. 2.5. Применение правил вывода 3 и 4 для декомпозиции

Процедура декомпозиции зачастую является неоднозначной, т. е. разные наборы отношений могут дать корректное проектное решение, и окончательный выбор зависит от способа физической реализации проектного решения в конкретной СУБД.

Если при декомпозиции удалось достичь состояния, когда дальнейшие ее шаги, не влекущие за собой потерь ФЗ, становятся невозможными, то нужно или пересмотреть порядок выполнения предыдущих шагов декомпозиции, или уточнить и перестроить рисунок. В крайнем случае для отдельных отношений допустим отказ от НФБК, но такая ситуация считается аномальной и ее причины должны быть обоснованы.

Замечание

На практике при создании логической модели данных, как правило, не следуют прямо приведенному алгоритму нормализации. Опытные разработчики обычно сразу строят отношения в 3 НФ. ER-диаграммы сразу позволяют создавать отношения в 3 НФ, если пользоваться специальными правилами, которые приведены в *главе 1*.

Тем не менее приведенный алгоритм важен по двум причинам. Во-первых, этот алгоритм показывает, какие проблемы возникают при разработке слабо нормализованных отношений. Во-вторых, обычно модель предметной области никогда не бывает правильно разработана с первого шага. Эксперт предметной области может забыть о чем-либо упомянуть, разработчик может неправильно понять эксперта, во время разработки могут измениться правила, принятые в предметной области, и т. д. Все это приведет к появлению новых зависимостей, которые отсутствовали в первоначальной модели предметной области. В данных ситуациях как раз и необходимо использовать алгоритм нормализации хотя бы для того, чтобы убедиться, что отношения остались в 3 НФ и логическая модель не ухудшилась.

Как было показано ранее, алгоритм нормализации состоит в выявлении функциональных зависимостей, имеющихся в предметной области, и соответствующей декомпозиции отношений. При этом возникают естественные вопросы: что произойдет с уже накопленными данными? не будут ли данные потеряны в ходе декомпозиции? можно ли вернуться обратно к исходным отношениям, если будет принято решение отказаться от декомпозиции? восстановятся ли при этом данные?

При декомпозиции из одного отношения получают два или более отношений, каждое из которых содержит часть атрибутов исходного отношения. Из полученных новых отношений необходимо удалить дубликаты строк, если таковые возникли. Это в точности означает, что декомпозиция отношения есть не что иное, как взятие одной или нескольких проекций исходного отношения так, чтобы эти проекции в совокупности содержали (возможно, с повторениями) *все* атрибуты исходного отношения, т. е. при декомпозиции *не должны теряться атрибуты отношений*. Но при декомпозиции также не должны потеряться и сами данные, что имеет место в том случае, если возможна обратная операция, — по декомпозированным отношениям можно восстановить исходное отношение *в точности в прежнем виде*.

Операцией, обратной операции проекции, является операция соединения отношений. Имеется большое количество видов операций соединения, но т. к. при восстановлении исходного отношения путем соединения проекций не должны появиться новые атрибуты, то необходимо использовать *естественное соединение*.

Проекция $R(X)$ отношения R на множество атрибутов X называется *собственной*, если множество атрибутов X является *собственным подмножеством* множества атрибутов отношения R, т. е. множество атрибутов X не совпадает с множеством *всех* атрибутов отношения R.

Собственные проекции $R1$ и $R2$ отношения R называются *декомпозицией без потерь*, если *отношение R точно восстанавливается* из них при помощи естественного соединения *для любого состояния* отношения R, т. е. $R1$ JOIN $R2 = R$.

Установлено, что *без дополнительных ограничений* на отношение R нельзя говорить о декомпозиции без потерь. Такими дополнительными ограничениями как раз и являются функциональные зависимости. Существует так называемая теорема Хеза, которая обосновывает корректность процедуры нормализации, основанной на декомпозиции без потерь.

Теорема Хеза. Пусть R(A, B, C) является отношением и A, B, C — атрибуты или множества атрибутов этого отношения. Если имеется функциональная зависимость ($A \rightarrow B$), то проекции $R1 = R(A, B)$ и $R2=R(A, C)$ образуют декомпозицию без потерь и, следовательно, алгоритм нормализации является корректным, т. е. в ходе нормализации не происходит потери информации.

Замечание

В процессе доказательства теоремы Хеза наличие ФЗ *не используется* при доказательстве включения $R1$ JOIN $R2 \supseteq R$. Это означает, что при выполнении декомпозиции и последующем восстановлении отношения при помощи естественного соединения кортежи исходного отношения *не будут потеряны*. Основной смысл теоремы Хеза заключается в доказательстве того, что при этом *не появятся новые кортежи*, отсутствовавшие в исходном отношении.

2.5.10. Анализ критериев оптимальности реляционной базы данных

Соберем воедино результаты влияния уровня нормализации отношений на качество логического моделирования и на производительность базы данных. Это сделано в табл. 2.5.

Общий вывод состоит в том, что сильно нормализованные отношения оказываются лучше спроектированными, адекватнее соответствуют предметной области, легче в разработке, для них быстрее выполняются операции модификации базы данных. Правда, это достигается ценой некоторого замедления выполнения операций выборки данных. У слабо нормализованных отношений единственное преимущество — при обращении к базе данных только с запросами на выборку они выполняются быстрее. Это связано с тем, что в таких отношениях уже как бы произведено соединение отношений и на это

не затрачивается время при выборке данных. Таким образом, выбор степени нормализации отношений зависит от характера запросов, с которыми чаще всего обращаются к базе данных.

Таблица 2.5. *Сравнение критериев оптимальности логической модели*

Критерий	**Отношения слабо нормализованы (1 НФ, 2 НФ)**	**Отношения сильно нормализованы (3 НФ, 4 НФ)**
Адекватность базы данных предметной области	ХУЖЕ	ЛУЧШЕ
Легкость разработки и сопровождения базы данных	СЛОЖНЕЕ	ЛЕГЧЕ
Скорость выполнения вставки, обновления, удаления	МЕДЛЕННЕЕ	БЫСТРЕЕ
Скорость выполнения выборки данных	БЫСТРЕЕ	МЕДЛЕННЕЕ
Общая производительность OLTP-приложения	НИЖЕ	ВЫШЕ
Общая производительность OLAP-приложения	ВЫШЕ	НИЖЕ

2.6. Примеры нормализации, основанной на НФ

Рассмотрим несколько примеров практического использования теории, изложенной в *разд. 2.5.*

2.6.1. Приведение к 1 НФ

Рассмотрим возможные способы хранения контактных телефонов для абстрактного круга людей без привязки к конкретной предметной области. Пусть принято решение идентифицировать человека с помощью Ф.И.О. Предполагается, что у одного человека есть несколько контактных телефонов, возможно разного типа (рабочий, домашний, мобильный и т. д.). Для внесения информации о телефонах создадим отношение `КОНТАКТЫ` с ключом `ФИО`. Для внесения данных о нескольких телефонах создадим в этом отношении достаточное количество столбцов. Тогда таблица `КОНТАКТЫ` будет примерно такой.

Таблица 2.6. *Ненормализованное отношение* `КОНТАКТЫ`

ФИО	**Телефон_1**	**Телефон_2**	**Телефон_3**	**...**	**Телефон_*n***
Комаров Н. Л.	465432	8-050-5664556			
Азофов А. А.	764544				
Бимс В. В.	346784				
Дышкин Е. Е.	893840	7855567	8-050-55556755		
Герасимов Ф. Ф.	765544				
Иванов И. И.	321434	676876786	88666543		8-095-68767868
Васильев О. П.	654534				
Неромов В. Н.	332444				
Ретинов Р. Р.	645654	789787879	666786875		

Выглядит таблица `КОНТАКТЫ` не слишком хорошо. В ней неизбежно будут присутствовать в большом количестве `NULL`-значения, а куда заносить тип телефона, вообще не ясно. Но самое главное, невозможно сказать, сколько же вообще нужно столбцов для хранения телефонов.

Очевидная проблема возникла из-за того, что атрибуты отношения `КОНТАКТЫ` `Телефон_1` — `Телефон_n` образуют повторяющуюся группу. Решить эту проблему можно так. Перестраиваем отношение `КОНТАКТЫ`, введя в него атрибут `Тип_телефона`, а вместо повторяющейся группы атрибутов `Телефон_1` — `Телефон_n` вводим единственный атрибут `Телефон` и вносим его в ключ перестроенного отношения: `КОНТАКТЫ` (`ФИО`, `Телефон`, `Тип_телефона`). Очевидно, что перенести данные о телефонах в перестроенное отношение `КОНТАКТЫ` несложно (табл. 2.7). Проблема повторяющихся групп таким образом решена.

Таблица 2.7. *Нормализованное отношение* `КОНТАКТЫ` *без повторяющихся групп*

ФИО	**Телефон_1**	**Тип_телефона**
Комаров Н. Л.	465432	дом
Азофов А. А.	764544	дом
Бимс В. В.	346784	дом
Дышкин Е. Е.	893840	дом
Герасимов Ф. Ф.	765544	дом
Иванов И. И.	321434	дом

Таблица 2.7 *(окончание)*

ФИО	Телефон_1	Тип_телефона
Васильев О. П.	654534	дом
Неромов В. Н.	332444	дом
Ретинов Р. Р.	645654	дом
Комаров Н. Л.	8-050-5664556	моб
Дышкин Е. Е.	7855567	раб
Дышкин Е. Е.	8-050-55556755	моб
Иванов И. И.	676876786	раб
Иванов И. И.	88666543	раб
Иванов И. И.	8-095-68767868	межг
Ретинов Р. Р.	789787879	раб
Ретинов Р. Р.	666786875	раб

Подумаем теперь, как повлияет устранение повторяющихся групп из отношения КОНТАКТЫ на скорость поиска данных о телефонах по Ф.И.О. человека, т. к. очевидно, что именно эта операция поиска будет наиболее частой для отношения КОНТАКТЫ. Разница для поиска по табл. 2.6 и 2.7 состоит в том, что в первом случае вся информация о телефонах может быть получена из одной записи, в то время как во втором случае нужно найти некоторый диапазон. Однако в обоих случаях Ф.И.О. является первым атрибутом логического ограничения первичного ключа, по которому в соответствии со стандартной рекомендацией будет построен кластерный индекс. Для поиска по кластерному индексу в данном случае нет существенного различия в быстродействии.

Однако атрибут Ф.И.О., как мы знаем *(см. главу 1)*, является плохим кандидатом на роль первичного ключа, хоть атомарного, хоть составного, с точки зрения физической реализации базы данных, т. к. он имеет строковый тип данных и, следовательно, не удовлетворяет требованию компактности первичного ключа. Если поступить в соответствии со стандартной рекомендацией и заменить в первичном ключе и соответственно в кластерном индексе атрибут, идентифицирующий человека, на целочисленный идентификатор ФИО_Ид, то индексный поиск организовать по Ф.И.О. не удастся без дополнительных индексов. В первом случае такой индекс по полю ФИО может быть создан с ограничением уникальности, а во втором нет, что может повлиять на скорость поиска. С другой стороны, в нормализованном отношении КОНТАКТЫ

для поиска телефонов по фамилии можно создать составной индекс по паре полей (ФИО, Телефон). Тогда информация о телефонах может быть получена из покрывающего индекса *(см. разд. 5.3)*, а это очень эффективный прием. Очевидно, что создание покрывающего индекса для ненормализованной таблицы КОНТАКТЫ бессмысленно, т. к. он будет очень широким.

Замечание

На практике все же можно оставить в таблице поля, образующие повторяющиеся группы, в ситуации, когда заранее известно их предельное количество и оно невелико (до четырех). Кроме того, данные в этих полях не должны содержать много NULL-значений.

По поводу второго пункта в определении 1 НФ, а именно скалярного смысла атрибутов отношения, можно сказать следующее. Разумеется, телефоны конструируются по понятным правилам, и если записан номер телефона, то можно легко понять, присутствует ли в нем код города, код страны или код оператора мобильной связи. Появление нескалярных атрибутов в данном случае возможно, если в запросах, например, нужно из номера телефона выделять код города или код оператора мобильной связи.

Рассмотрим ситуацию с междугородними телефонами. Если запросы к таблице КОНТАКТЫ выполняются только с целью увидеть информацию для осуществления телефонного звонка данному человеку, то тогда лучше всего увидеть весь номер целиком, и атрибут Телефон представляется скалярным. Но если мы затеем выделение кодов городов из номеров телефонов, например с помощью строковых функций, для использования этих кодов в условиях поиска, или еще хуже, для соединения таблиц, — тогда атрибут Телефон перестает быть скалярным, и для его составной части, содержащей код города, нужно создавать отдельное поле. При этом если нормализацию доводить до предельного уровня, тогда код города придется убрать из поля, содержащего номер телефона.

Замечание

Отношение КОНТАКТЫ, приведенное в табл. 2.7, находится только лишь в 1 НФ. Даже 2 НФ для него не выполняется, т. к. скорее всего в нем имеется ФЗ вида Телефон → Тип_телефона.

2.6.2. Построение диаграмм ФЗ и минимальных покрытий

В этом разделе будут рассмотрены примеры описания предметных областей для различных типов бизнес-среды. На основании этих описаний будут построены диаграммы ФЗ, после чего будут показаны минимальные покрытия и построены удовлетворяющие им проектные решения.

Пример 1. Построение простейшей диаграммы ФЗ

Пусть имеется предметная область, связанная с функционированием автопредприятия, и нужно спроектировать отношения для хранения следующей информации. На автопредприятии есть парк автомобилей, которые выезжают в рейсы. Каждый рейс уникально идентифицируется номером. Каждый рейс выполняет один из водителей, работающих на автопредприятии. Каждый водитель закреплен за автомобилем, т. е. водитель ездит все время на одном и том же автомобиле, но за одним автомобилем может быть закреплено несколько водителей. О каждом водителе известны определенные сведения. О каждом автомобиле также известны некоторые данные, например модель и цвет. Понятно, что на автопредприятии может существовать несколько однотипных по модели и цвету автомобилей. Диаграмма ФЗ для этого примера изображена на рис. 2.6. Она достаточно проста, хотя и содержит больше ФЗ, чем будет включено в минимальное покрытие. Лишние ФЗ показаны пунктиром, все они не входят в минимальное покрытие по правилу цепочек вывода ФЗ.

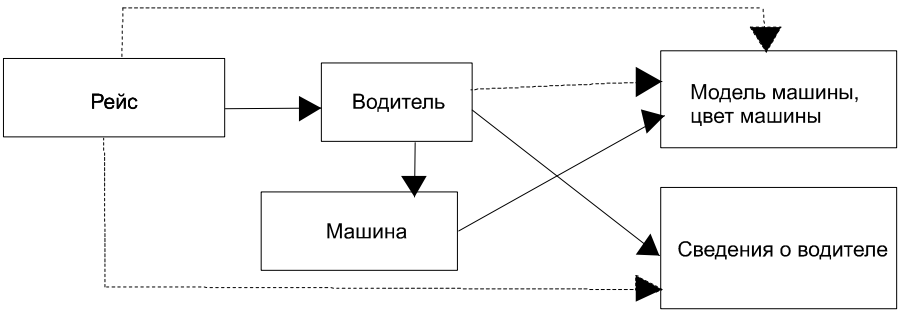


Рис. 2.6. Диаграмма ФЗ и минимальное покрытие для примера про автопредприятие

Если ввести атрибуты для идентификации сущностей, совпадающие для простоты с их именами, тогда на основании минимального покрытия, изображенного на рис. 2.6, может быть получено следующее проектное решение, нормализованное до 3 НФ:

```
R1(Рейс, Водитель)
R2(Водитель, Машина, Сведения_о_водителе)
R3(Машина, Модель, Цвет)
```

Пример 2. Построение диаграммы ФЗ с псевдотранзитивными зависимостями

Опишем следующую ситуацию. Пусть имеются студенты, объединенные в группы. Каждый студент может обучаться только в одной группе. Для каж-

дой группы известно название специальности, по которой обучаются все студенты данной группы. Групп с одинаковым названием специальности может быть несколько. Для каждой специальности определен перечень предметов, которые изучают соответствующие студенты в течение, например, определенного семестра. По изучаемым предметам у каждой группы имеется преподаватель, читающий лекции. При этом один и тот же лектор должен читать свой предмет всем группам определенной специальности. То есть группы с одинаковой специальностью объединены для чтения лекций в потоки. С другой стороны, любой лектор может читать свой предмет нескольким потокам. Лекторы читают лекции более чем по одному предмету. Описанная ситуация действует в течение одного определенного семестра. Диаграмма ФЗ для этого примера изображена на рис. 2.7.

Данный пример сложнее предыдущего, т. к. в нем неизбежно присутствуют ФЗ с составными детерминантами. Кроме очевидных ФЗ (1), (2) и (3) с атомарными детерминантами существует еще ФЗ (4), поскольку утверждается, что один и тот же лектор должен читать лекции по определенному предмету всем группам одной специальности. Кроме того, ясно, что для любой комбинации предмета и группы однозначно определяется лектор, который его читает, — так возникает ФЗ (5). Аналогично возникает и ФЗ (`Студент, Предмет` → `Лектор`), но она не показана на рис. 2.7.

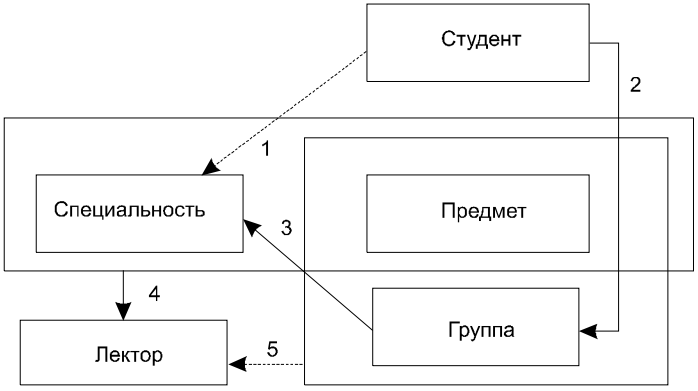


Рис. 2.7. Диаграмма ФЗ и минимальное покрытие для примера про студентов

Как и в предыдущем примере, пунктиром на рис. 2.7 показаны ФЗ, которые не входят в минимальное покрытие. Так, ФЗ (1) не входит в минимальное покрытие по правилу цепочки, а ФЗ (5) является псевдотранзитивной *(см. правило 5 вывода ФЗ)*. Итак, в минимальном покрытии оставлено три ФЗ,

изображенных сплошными стрелками, которые дают следующее проектное решение, нормализованное до 3 НФ:

R1(Студент, Группа)
R2(Группа, Специальность)
R3(Специальность, Предмет, Лектор)

Сможем ли мы на основании приведенного проектного решения определить по номеру группы и названию предмета, кто из лекторов читает данный предмет? То есть необходимо убедиться, что в полученном проектном решении не потеряна ФЗ (5). Для этого выполним естественное соединение отношений R2 и R3 по атрибуту Специальность. В результате получим отношение

R2_3(Специальность, Предмет, Группа, Лектор)

В этом отношении содержатся все комбинации предметов и групп, которые только возможны для групп данной специальности. Пара атрибутов (Предмет, Группа) является новым потенциальным ключом отношения R2_3. Отсюда следует, во-первых, вывод о том, что отношение R2_3 содержит ФЗ (5) вида (Предмет, Группа → Лектор), а во-вторых, отношение R2_3 не удовлетворяет определению 2 НФ, т. к. в нем имеется атрибут, функционально зависящий от части ключа отношения, а именно имеется ФЗ (Группа → Специальность). Поэтому с точки зрения нормализации нельзя оставить в проектном решении отношение R2_3 вместо двух отношений R2 и R3.

2.7. Построение нормализованных проектных решений для задачи материального учета

Опишем некоторые ситуации, характерные для задачи материального учета *(см. главу 1)* с целью разбора примеров построения диаграмм ФЗ и минимальных покрытий. Введем следующие сущности: ТОВАР, ПРИХОДНАЯ_НАКЛАДНАЯ, РАСХОДНАЯ_НАКЛАДНАЯ. Имена для ключей сущностей для простоты выберем совпадающие с именами сущностей. ER-диаграмма для данной задачи представлена на рис. 2.8. Подразумевается, что во всех ситуациях о товарах нужно хранить определенные сведения, зависящие от идентификатора товара, и эта связь не показана на диаграммах ФЗ для экономии места, но отношение для хранения сведений о товаре присутствует во всех проектных решениях. Связь Связь3, показанная пунктиром, в примере 1 не учитывается.

Предварительные отношения для данной ER-диаграммы приведены в табл. 2.8.

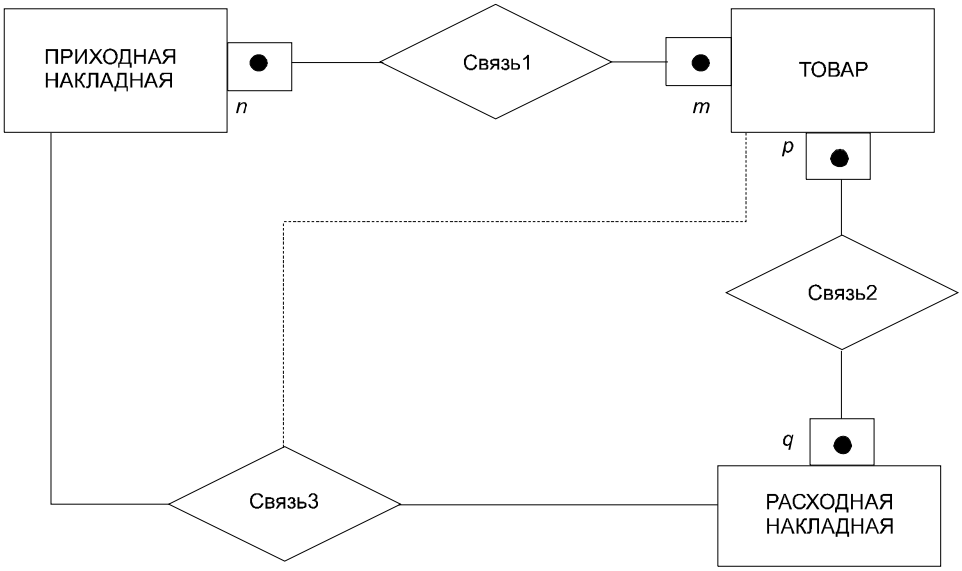


Рис. 2.8. Общая ER-диаграмма задач материального учета

Таблица 2.8. *Предварительные отношения для ER-диаграммы с рис. 2.8.*

Без учета связи `Связь3`	**С учетом связи** `Связь3` **вместо связи** `Связь2`
`R1(N_прих, ...)`	`R1(N_прих, ...)`
`R2(N_прих, Товар, ...)`	`R2(N_прих, Товар, ...)`
`R3(Товар, Сведения о товаре)`	`R3(Товар, Сведения о товаре)`
`R4(N_расх, Товар, ...)`	`R4(N_прих, N_расх, Товар, ...)`
`R5(N_расх, ...)`	`R5(N_расх, ...)`

2.7.1. Пример 1. Произвольные цены без фиксированной прибыли

Итак, пусть поставщики поставляют любые товары по известным ценам, которые изменяются произвольно, и для конкретного товара привязаны только к конкретной приходной накладной. На поставку товара оформляется приходная накладная с несколькими товарными позициями, в которую включаются разнообразные партии товаров от конкретного поставщика. За один день от одного поставщика могут приходить несколько произвольно сформированных приходных накладных. Товары отпускаются по расходным накладным, которые формируются также произвольно, т. е. за одну дату любому получателю может быть отправлено любое количество расходных накладных

с разными товарами по любой отпускной цене. Диаграмма ФЗ для этого случая приведена на рис. 2.9.

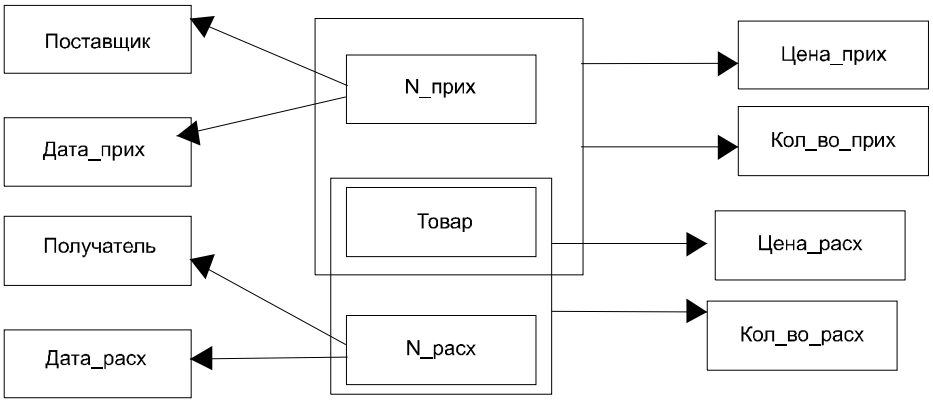


Рис. 2.9. Диаграмма ФЗ примера 1

Диаграмма ФЗ, приведенная на рис. 2.9, является готовым минимальным покрытием, поэтому ее можно использовать для построения проектного решения. На основании правила о помещении в одно отношение всех ФЗ с одинаковым детерминантом, проектное решение имеет вид:

```
R1(N_прих, Дата_прих, Поставщик)
R2(N_прих, Товар, Кол_во_прих, Цена_прих)
R3(Товар, Сведения о товаре)
R4(N_расх, Товар, Кол_во_расх, Цена_расх)
R5(N_расх, Дата_расх, Получатель)
```

Данное проектное решение дает все отношения, находящиеся в НФБК. Этот вывод можно сделать, если сравнить ключи отношений (они подчеркнуты) и детерминанты ФЗ (рис. 2.9).

Однако приведенное проектное решение не позволяет контролировать процесс ценообразования. В данном случае отпускная цена назначается произвольно, и может возникнуть ситуация, когда прибыль будет меньше запланированной или ее не будет вообще. Для контроля ценообразования нужно при описании предметной области сформулировать и использовать правило ценообразования.

2.7.2. Пример 2. Произвольные цены с фиксированной прибылью

Для обеспечения фиксированной прибыли можно принять следующее бизнес-правило, которое состоит в том, что каждая единица товара должна быть

продана на определенный процент дороже, чем она была приобретена. В ситуации, когда имеются одни и те же товары, закупленные по разным приходным ценам, вычислять отпускную цену по проектному решению примера 1 невозможно. Но если мы будем знать, из какой приходной накладной сколько единиц каждого товара включено в данную расходную накладную, то тем самым его отпускная цена определится однозначно и будет функционально зависеть от трех атрибутов. То же самое можно теперь сказать и про атрибут `Кол_во_расх`. Изменение в диаграмме ФЗ показано на рис. 2.10. В ER-диаграмме тоже нельзя обойтись бинарными связями — вместо бинарной связи придется использовать связь-триплет с именем `Связь3` (пунктирные линии на рис. 2.8).

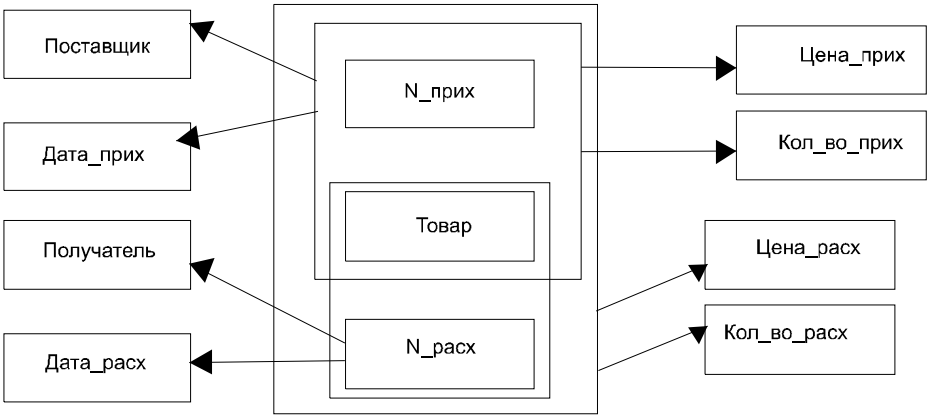


Рис. 2.10. Диаграмма ФЗ примера 2

Новое проектное решение, полученное аналогично предыдущему примеру, находится в НФБК:

```
R1(N_прих, Дата_прих, Поставщик)
R2(N_прих, Товар, Кол_во_прих, Цена_прих)
R3(Товар, Сведения о товаре)
R4(N_расх, Товар, N_прих, Кол_во_расх, Цена_расх)
R5(N_расх, Дата_расх, Получатель)
```

Теперь атрибут `Цена_расх` не может быть введен как базовый независимо, а должен стать вычислимым. Значение атрибута `Цена_расх` может быть вычислено путем связывания отношений `R2` и `R4` по паре полей `Товар`, `N_прих`, после чего для каждой товарной позиции нужно умножить значения поля `Цена_прих` на процент прибыли.

Данное проектное решение полностью реализует бизнес-правило о продаже любой единицы товара дороже на заданный процент прибыли, но в целом не является удачным.

1. Отношение `R4` имеет первичный ключ, составленный из трех атрибутов. Физическая реализация такого широкого первичного ключа весьма накладна, т. к. требует громоздкого индекса со всеми вытекающими отсюда негативными последствиями.
2. Данное проектное решение порождает составное (по двум полям) ограничение внешнего ключа между таблицами `R2` и `R4`, которое очень трудно поддерживать как с точки зрения размеров индексов, нужных для этого, так и с точки зрения написания кода приложения, например для подсчета остатков товаров.
3. В одну расходную накладную могут оказаться включенными две товарные позиции с одноименным товаром, но с разными ценами, что является странным, с точки зрения привычного вида выходных форм. Можно, конечно, в выходных документах попытаться скрыть наличие одноименных товаров с разными ценами, выставляя в выходной форме на любой товар только одну товарную позицию и некоторую усредненную цену, из каких бы приходных накладных не включались товары в данную расходную накладную. Однако все это требует дополнительных усилий при реализации.

В свете вышесказанного более удачным, обеспечивающим фиксированную прибыль, представляется проектное решение из следующего примера и его развитие, приведенное в *главе 5.*

2.7.3. Пример 3. Постоянные цены и история цен

Пусть поставщики на достаточно продолжительное время зафиксировали цены на поставляемые товары, т. е. образовалась новая ФЗ (1) (рис. 2.11). Пунктиром изображена ФЗ из предыдущего примера. После появления ФЗ (1) пунктирная ФЗ должна быть исключена из минимального покрытия как добавленная *(см. правило 5 вывода ФЗ)*.

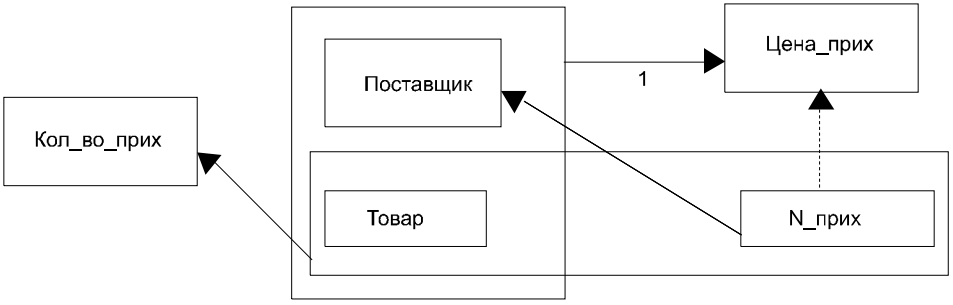


Рис. 2.11. Диаграмма ФЗ примера 3

Если ER-диаграмму оставить без изменения (см. рис. 2.8), то в предварительных отношениях невозможно разместить дополнительные атрибуты так, чтобы сохранить ФЗ (1), ввиду того, что не будет предварительного отношения с ключом, совпадающим с ее детерминантом (см. табл. 2.8). Из этой ситуации можно выйти путем введения новой сущности с именем ПОСТАВЩИК, выбрав атрибут Поставщик в качестве ее ключа.

Но можно поступить и по-другому. Пусть предварительные отношения будут прежними, причем связь Связь3 не используется. Большинство дополнительных атрибутов останутся на своих местах. Поместив атрибуты Поставщик и Цена_прих в отношение R2, получим отношение

R2(N_прих, Товар, Поставщик, Цена_прих, Кол_во_прих)

Отношение R2 уже не будет находиться в НФБК, т. к. в нем имеется ФЗ (Товар, Поставщик → Цена_прих). Выполнив декомпозицию отношения R2 по правилу, согласно которому для каждой ФЗ выделяется свое отношение, получим проектное решение, находящееся в НФБК:

R2′(N_прих, Поставщик) + R1(N_прих, Дата_прих)
R2′′(N_прих, Товар, Кол_во_прих)
R2′′′(Товар, Поставщик, Цена_прих)
R3(Товар, Сведения о товаре)
R4(Товар, N_расх, Кол_во_расх, Цена_расх)
R5(N_расх, Дата_расх, Получатель)

Так как отношения R2′ и R1 имеют одинаковый ключ для сущностей с обязательным классом принадлежности, то их можно объединить в одно отношение.

Последнее проектное решение также не позволяет однозначно вычислять значение атрибута Цена_расх без знания, из какой приходной накладной продается товар. Внесение в R4 атрибута N_прих, как это сделано в примере 2, в данном случае решит проблему, но это решение будет еще более громоздким. Теперь для определения приходной цены конкретного товара, чтобы при формировании расходной накладной умножить эту цену на процент прибыли, придется соединять уже не два отношения, как в предыдущем примере, а четыре:

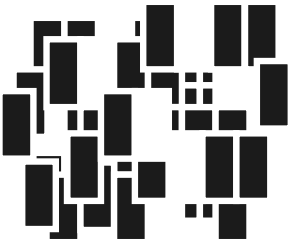
R1(N_прих, Дата_прих, Поставщик)
R2′′(N_прих, Товар, Кол_во_прих)
R2′′′(Товар, Поставщик, Цена_прих)
R4(N_прих, Товар, N_расх, Кол_во_расх, Цена_расх)

С точки зрения простоты и скорости реализации, очень бы пригодилось наличие ФЗ вида (Товар → Цена_прих), но этой ФЗ не возникает ни из каких

бизнес-правил, т. к. очевидно, что цены на товары не могут оставаться неизменными с течением времени. Поэтому сильной натяжкой можно считать и наличие ФЗ (Товар, Поставщик → Цена_прих), с рассмотрения которой начался разбор последнего примера.

Способ удобной имитации введения ФЗ (Товар → Цена_прих) предложен в основном проектном решении для базы данных МАТ_УЧЕТ. Он заключается во введении так называемой учетной цены товара и образовании ФЗ (Товар → Цена_учетн) на какой-то промежуток времени. Атрибут Цена_учетн размещается в отношении с ключом Товар, и учетной ценой пользуются при различных финансовых вычислениях, выполняемых в оперативном режиме. В частности, она является основным базовым атрибутом, от которого рассчитывается отпускная цена товаров при самых разных способах ценообразования, которые могут быть весьма сложными, т. к. реализуют учет прибыли, учет оптовых скидок и розничных наценок и т. д. Так как учетная цена может изменяться, то создается таблица для ведения истории изменения цен, в которой хранится ФЗ вида (Товар, Дата → Цена_учетн), где атрибут Дата используется для занесения дат изменения учетной цены на определенный товар. История цен применяется для проведения необходимых аналитических исследований, которые не обязательно нужно выполнять в реальном режиме времени. Проектное решение, использующее этот прием, как уже было сказано, реализовано и подробно разобрано в базе данных МАТ_УЧЕТ *(см. главу 5)*.

ГЛАВА 3



Физическое представление базы данных

Как известно, фаза, предшествующая физическому проектированию, называется *фазой логического проектирования*. Результаты ее выполнения в значительной степени независимы от особенностей реализации проекта в среде конкретной *целевой СУБД*. Можно сказать, что при логическом проектировании разработчик сосредоточен на том, что нужно сделать, тогда как при физическом проектировании решается вопрос о том, как это сделать.

Разумеется, первым делом необходимо определить саму целевую СУБД. Соображения, которые нужно учесть при этом, достаточно сложны и многоплановы, но, разумеется, какое-то решение в конечном итоге будет принято. В нашем случае целевой СУБД является MS SQL-сервер 2000. Исходной информацией для перехода к этапу непосредственной физической реализации базы данных является адекватная *физическая модель базы данных*, созданная с помощью какого-либо CASE-средства. При такой постановке задачи, казалось бы, само физическое проектирование уже выполнено, и остается только сгенерировать схему базы данных на основании ее физической модели, что зачастую можно сделать автоматически в рамках выбранного CASE-средства. Эта последовательность действий обычно вполне приемлема, однако разработчику почти всегда приходится учитывать какие-то нюансы физического моделирования вне рамок CASE-средства.

Кроме того, необходимо иметь представление о способах хранения, доступа и обработки информации на самом низком уровне. Понимание этих основ позволяет обоснованно принимать решения, связанные с проектированием файловой структуры базы данных, выбором системных ресурсов, адекватными настройками сервера и т. д.

Кратко перечислим те проблемы, которые обязательно приходится решать при проектировании физического представления базы данных.

- Определение требований к *системным ресурсам*.
- Выбор *файловой структуры* и определение групп файлов.
- Анализ *запросов и транзакций* с целью корректного размещения базовых таблиц и индексов по группам файлов.
- Определение *вторичных индексов* и размещение их в группах файлов.
- Анализ адекватности базы данных, с точки зрения возможности выполнения всех заданных *транзакций при допустимом уровне их конфликтности* и приемлемом совокупном быстродействии.
- Анализ необходимости введения *контролируемой избыточности данных* и средств ее реализации.
- Определение необходимости применения специфических *настроек сервера для конкретной системной конфигурации*.
- Разработка механизмов защиты, к которым относятся:
  - пользовательские представления;
  - определение пользователей и прав доступа;
  - шифрование данных.

В данной главе дается общее представление о способах хранения и доступа к данным, рассматривается материал, связанный с вопросами выбора оптимальной файловой структуры базы данных и размещения объектов базы данных в ее файлах. Здесь же в минимальном объеме приведены сведения по поводу настройки сервера и связанные с ними практические рекомендации. Вопросы введения контролируемой избыточности были подробно рассмотрены в *главе 2*. Проблемы, связанные с проектированием и выполнением транзакций, отдельно вынесены в *главу 6*.

Что же касается вторичных индексов, под которыми понимаются индексы, необходимые для повышения быстродействия отдельных критических запросов, — то в *главе 5* эта проблема подробно анализируется.

Замечание

Если до сих пор основная работа выполнялась в рамках выбранного CASE-средства, то теперь происходит переход к работе непосредственно на сервере СУБД. Неизбежно будут иметь место какие-либо действия, которые изменяют физическую модель базы данных, например, добавятся вторичные индексы, определятся файлы и их группы, и многое другое. Необходимо не забывать отслеживать эти изменения и периодически заносить их в физическую модель базы данных, выполняя синхронизацию этой модели с текущим состоянием системного каталога базы данных.

3.1. Понятие о системных ресурсах

Чтобы достичь высокой производительности системы, разработчик физического проекта базы данных должен решить, каким образом четыре основных компонента оборудования, называемые *системными ресурсами*, взаимодействуют между собой. К системным ресурсам относятся следующие компоненты.

- *Оперативная память*. Доступ к данным в оперативной памяти осуществляется во много раз быстрее (в десятки или сотни тысяч раз). В общем случае, чем больше объем доступной СУБД оперативной памяти, тем быстрее будут работать приложения. Рекомендуется поддерживать в системе такой режим, при котором около 5 % ее оперативной памяти остается свободной. Если системе не хватает основной оперативной памяти для удовлетворения потребностей всех процессов, то часть страниц может быть вытолкнута из оперативной памяти на диск, а потом они снова будут считываться по мере надобности. Установить наличие проблем с объемом доступной оперативной памяти можно по высокому уровню страничного обмена в системе.
- *Процессор.* Устанавливает задачи для других системных ресурсов и выполняет пользовательские процессы. Важнейшим условием эффективной работы процессора является отсутствие конкуренции за него между процессами. Обычно подобная конкуренция ведет к резкому возрастанию страничного обмена.
- *Дисковый ввод/вывод.* В любой достаточно мощной СУБД процессы сохранения и выборки данных связаны с выполнением множества операций ввода/вывода. Обычно изготовители дисковых устройств указывают рекомендуемое количество операций ввода/вывода в секунду. Если реальный показатель превышает это значение, то дисковая подсистема становится узким местом. Иногда полезно отключить устройство кэширования данных дисковым контроллером в том случае, если это устройство не создавалось специально для серверов баз данных.
- *Сеть.* Сеть может стать узким местом всей системы при чрезмерном возрастании сетевого трафика или большом количестве сетевых коллизий.

3.2. Общие сведения о файлах SQL-сервера

База данных состоит из двух и более файлов, каждый из которых может использоваться лишь одной базой данных. У файлов существует два имени — логическое и физическое. Логическое имя подчиняется стандартным правилам выбора имен объектов SQL-сервера. Физическое имя — это имя файла с

точки зрения операционной системы, под управлением которой функционирует SQL-сервер. Имеются следующие типы файлов.

- *Первичный файл данных*, который используется для хранения данных. Этот файл всегда существует в единственном экземпляре, создаваясь по умолчанию командой создания базы данных. Стандартное расширение — mdf.
- *Вторичные файлы данных* — это вспомогательные файлы данных, которые используются для распределения данных по нескольким физическим дискам операционной системы. Стандартное расширение — ndf.
- *Файлы журналов транзакций* — служат для хранения сведений о процессе выполнения транзакций в базе данных *(см. главу 6)*. База данных всегда содержит по крайней мере один файл журнала транзакций, но при необходимости можно создать несколько таких файлов. Стандартное расширение — ldf.

Замечание

Не рекомендуется изменять стандартные расширения файлов базы данных, хотя вполне возможно при выполнении команды `CREATE DATABASE` задать для файла базы данных какое-нибудь свое расширение.

MS SQL-сервер содержит все необходимые средства и команды работы с файлами базы данных, поэтому обычно нет необходимости использовать для этих целей средства операционной системы. Более того, зачастую и невозможно средствами операционной системы выполнить какие-либо действия, потому что SQL-сервер блокирует файлы активной базы данных и не позволяет, например, произвольно переименовывать или удалять их.

3.2.1. Понятие группы файлов

Файлы данных можно объединять в группы. Под *группой файлов* понимается несколько файлов, объединенных общим логическим именем. В одну и ту же группу файлов можно включать как файлы, расположенные на одном физическом устройстве, так и на разных. Создать пользовательскую группу файлов можно с помощью команды `CREATE (ALTER) DATABASE`.

У каждой базы данных существует по крайней мере одна группа файлов, которая содержит первичный файл данных. Имя этой группы файлов — `PRIMARY`. Файлы журналов транзакций не включаются ни в какую из групп файлов.

Любой пользовательский объект, будь то, например, таблица или индекс, можно создать с указанием того, в какой группе файлов он будет храниться. Если же эту информацию явно не указать, то пользовательский объект созда-

ется в первичной группе файлов с именем `PRIMARY`. Очевидно, что может иметь место ситуация, когда таблица хранится в одной группе файлов, а ее индексы в другой. Так как файлы из разных групп обычно располагаются на разных физических устройствах, то указанным приемом можно добиться того, что при выполнении сложного запроса к таблице, использующего доступ по индексу, будут задействованы два и более физических диска. Поэтому данные таблицы и ее индекса будут считываться одновременно (параллельно) разными контроллерами, что может сильно повысить производительность.

Если же в одну группу файлов включены файлы, расположенные на разных физических устройствах, то SQL-сервер автоматически определяет, в каком из файлов данной группы будет храниться определенная таблица или индекс. При этом все равно весьма вероятно, что данные из одного запроса будут сохраняться на различных физических устройствах, что дает возможность параллельного доступа к ним при чтении для выполнения запроса.

При определении групп файлов и размещении объектов в этих группах полезно придерживаться следующих рекомендаций:

- если принято решение работать с несколькими файлами, то в первичном файле следует хранить только системную информацию и метаданные, а для пользовательских данных лучше создавать отдельные файлы;
- нужно стараться распределять файлы и группы файлов по максимально возможному числу локальных физических устройств;
- объекты, занимающие много места, доступ к которым с большой степенью вероятности будет происходить одновременно, следует включать в разные группы файлов;
- не следует размещать файлы базы данных на сетевых дисках, т. к. это сильно снижает быстродействие;
- громоздкие таблицы, используемые в одном и том же критическом запросе, следует помещать в разные группы файлов;
- сами таблицы и их некластерные индексы лучше помещать в разные группы файлов;
- следует помнить, что создание кластерного индекса для таблицы в другой группе файлов перемещает в эту группу файлов саму таблицу, т. к. кластерный индекс на уровне листа совпадает с данными из самой таблицы;
- файлы журналов транзакций как с точки зрения повышения быстродействия, так и для обеспечения большей надежности следует располагать на дисках, отличных от дисков, на которых хранятся файлы данных;
- по возможности не следует располагать на одном и том же диске файлы баз данных и файлы операционной системы. Задать каталог, в котором бу-

дут по умолчанию располагаться файлы баз данных, можно при инсталляции SQL-сервера.

3.2.2. Создание базы данных

Создание базы данных выполняется с помощью команды `CREATE DATABASE`, сокращенный синтаксис которой с некоторыми параметрами приведен в листинге 3.1.

Листинг 3.1. Пример создания базы данных

```
USE master
GO
-- Создаем базу данных DB1
CREATE DATABASE DB1
ON
/* Включаем в первичную группу файлов базы данных файл с логическим
именем file1, начальным размером 10 Мбайт,
максимальным размером 200 Мбайт и приращением 20 % */
PRIMARY (NAME = file1,
 FILENAME = 'e:\SQL data files\filedat1.mdf',
 SIZE = 10MB, MAXSIZE = 200, FILEGROWTH = 20),
/* Создаем в базе данных пользовательскую группу файлов с именем Group1
и включаем в нее два файла с именами file2 и file3, начальный размер
этих файлов по умолчанию выбран равным 1 Мбайт, приращение первого
файла по умолчанию равняется 10 % и этот файл может расти неограниченно;
приращение второго файла запрещено */
FILEGROUP Group1
(NAME = file2,
 FILENAME = 'e:\SQL data files\filedat2.ndf'),
(NAME = file3,
 FILENAME = 'e:\SQL data files\filedat3.ndf',
       FILEGROWTH = 0)
--Создаем два файла журнала транзакций на другом физическом устройстве
LOG ON
 (NAME = filelog1,
 FILENAME = 'd:\SQL LOG files\filelog1.ldf',
 SIZE = 100MB, MAXSIZE = 200, FILEGROWTH = 20),
(NAME = filelog2,
 FILENAME = 'd:\SQL LOG files\filelog2.ldf',
 SIZE = 100MB, MAXSIZE = 200, FILEGROWTH = 20)
GO
```

3.3. Управление объемом базы данных

Одной из основных функций системного администратора является планирование объема базы данных и контроль этого объема. По умолчанию файлы базы данных автоматически увеличиваются на величину `FILEGROWTH` по мере необходимости (задание `FILEGROWTH = 0` запрещает прирост размера файла). Разумеется, может наступить ситуация, когда диск, на котором находится определенная группа файлов, исчерпал все свое свободное пространство, и тогда станет невозможным выполнение изменений для объектов базы данных. В этом случае необходимо принятие каких-то мер, которые могут быть самыми различными, исходя из конкретной ситуации. Вообще полезно предусмотреть автоматическую выдачу предупреждающих сообщений в ситуации, когда свободный объем дискового пространства становится меньше заданного.

Очевидно, что объем файлов базы данных должен быть достаточным для сохранения всех необходимых данных с каким-то запасом. Однако этот запас должен определяться на основании некоторых разумных критериев, потому что размеры физических устройств конечны, и задание слишком большого значения параметра `SIZE` приведет к тому, что значительная часть дискового пространства, которую могли бы использовать другие файлы, просто будет потеряна.

При определении предполагаемого объема базы данных следует принять во внимание размеры как системных, так и пользовательских объектов.

При создании базы данных первоначально ее содержимое переносится из системной базы данных `MODEL`, инициализируя таким образом системные таблицы пользовательской базы данных. По мере создания пользовательских объектов и заполнения их данными будет расти объем системной и пользовательской части базы данных.

При расчете предполагаемого объема системной части базы данных следует иметь в виду, что в момент создания любого пользовательского объекта в одну или несколько системных таблиц в зависимости от типа пользовательского объекта добавляется по одной записи на каждый из созданных пользовательских объектов. Обычно нет необходимости вникать в подробности заполнения информацией системных таблиц. Рекомендуется включать системные таблицы в первичный файл базы данных (файл MDF), а сам этот файл как раз и образует группу файлов `PRIMARY`. Размеры файла, содержащего системную информацию, должны быть достаточными для сохранения первоначального размера базы данных `MODEL`, и эти размеры вряд ли будут слишком большими в сравнении с файлами, содержащими пользовательские объекты. Поэтому задание незначительного автоматического приращения, принятого

по умолчанию (опция `FILEGROWTH` опущена) и равного 10 % от размера файла, обычно достаточно.

Пользовательские объекты рекомендуется размещать в отдельных файлах, объединенных в группы, отличные от группы файлов `PRIMARY`. При этом разместить новые данные невозможно, если все файлы группы заполнены до предела.

Для точного расчета размера пользовательских таблиц или индексов нужно принять во внимание, как эти объекты хранятся на диске. Для примера, подробный расчет размера таблицы кучи приведен в *разд. 3.4.* Расчет размера индексов или кластерных таблиц выполняется сложнее, т. к. нужно учесть размер служебной информации, необходимой для создания индекса, а также структуру индекса как бинарного дерева. При желании эти данные можно найти в BOL (SQL server Books OnLine, электронное справочное руководство по SQL-серверу) (поиск по "Estimating the Size of a Table").

Однако приблизительно прикинуть размер, например для таблицы, не очень сложно. Базовыми параметрами для его расчета, разумеется, являются предполагаемое количество записей, которое будет храниться в таблице, и средняя длина записи. С определением второго значения особых проблем нет — при заданной структуре таблицы известен максимально возможный размер каждого из ее полей. Обычно для полей переменной длины выбирают некоторое среднее значение, но можно для перестраховки взять и максимальное значение длины поля. Количество же записей в таблице может быть спрогнозировано только на основании анализа бизнес-среды, и эта процедура является внешней по отношению к базе данных. После определения количества записей в таблице нужно подсчитать количество записей, которое может поместиться на восьми страницах данных, называемых *экстентом*, разделив размер экстента на среднюю длину записи и округлив результат до ближайшего целого. Далее прогнозируемое количество записей следует разделить на размер экстента и округлить до целого в сторону увеличения — таким образом будет вычислено количество экстентов, занимаемых таблицей.

Расчет размера индекса выполнить сложнее, т. к. он зависит не только от количества индексируемых строк и размера ключа индексации, т. е. размера тех полей, по которым построен индекс. Количество дисковой памяти, необходимой для хранения индекса, зависит также от типа индекса (кластерный/некластерный) и степени заполнения индексных страниц.

Существенное значение на размер базы данных оказывает размер файлов журналов транзакций. В ярко выраженной OLTP-среде он может быть сопоставим с размерами файлов данных. Подробно о работе механизма транзакций SQL-сервера и необходимых для этого ресурсах рассказывается в *главе 6*. Здесь можно только ограничиться рекомендацией использовать для OLTP-

среды первоначальное значение размера файла журнала транзакций, равное 10—25 % от размера базы данных. В OLAP-среде это значение может быть уменьшено.

Если файлы данных созданы слишком большими или в процессе эксплуатации из них удалено значительное количество данных, то их размер можно уменьшить, а неиспользованное дисковое пространство вернуть операционной системе. Для выполнения этих операций служат команды средства DBCC (DataBase Console Commands, команды консоли базы данных), такие как DBCC SHRINKDATABASE и DBCC SHRINKFILE. Краткий синтаксис этих команд следующий.

```
DBCC SHRINKDATABASE( database_name [, target_percent ]
[, { NOTRUNCATE | TRUNCATEONLY } ])
```

и

```
DBCC SHRINKFILE ( { file_name | file_id }
 { [, target_size ]
| [, { EMPTYFILE | NOTRUNCATE | TRUNCATEONLY } ] })
```

Команда DBCC SHRINKDATABASE выполняет сжатие файлов базы данных с именем database_name таким образом, что в базе данных остаются все данные и может быть оставлено свободное пространство в объеме, указанном параметром target_percent. При этом часть данных может быть перенесена с целью уменьшения размера файла.

Команда DBCC SHRINKFILE выполняет сжатие конкретного файла базы данных, имя которого задано параметром file_name, до указанного в параметре target_size размера.

Если параметры размера в приведенных выше командах не заданы, то сжатие выполняется настолько, насколько это возможно. Разумеется, сжать базу данных нельзя сильнее, чем позволяют находящиеся в ней данные.

Параметр NOTRUNCATE указывает, что освобожденное файловое пространство должно остаться в распоряжении файлов базы данных. По умолчанию оно возвращается операционной системе.

Параметр TRUNCATEONLY указывает, что все неиспользуемое пространство в файлах данных должно быть освобождено в пользу операционной системы. При этом файлы сокращаются до границы последнего выделенного экстента без попытки перемещения каких-либо данных в нераспределенные страницы.

Параметр EMPTYFILE переносит все данные из файла в другие файлы соответствующей группы файлов, и больше в этом файле данные размещаться не будут. Такой файл является кандидатом на удаление с помощью команды ALTER DATABASE ... DROP FILE.

Команды сжатия базы данных и ее файлов выполняются в так называемом отложенном режиме, не влияя на работу пользователей, подключенных к базе данных. Поэтому результаты работы этих команд ощущаются не сразу.

Ранее указана возможность уменьшения размера файлов данных и команды, которые для этого необходимы. Разумеется, объем базы данных можно и увеличивать. Этот объем автоматически увеличивается на величину параметра `FILEGROWTH` по мере необходимости, если последний не равен нулю. Однако можно увеличить базу данных и путем добавления в нее нового файла. Это действие выполняется с помощью команды `ALTER DATABASE ... ADD FILE`. Далее приведен листинг 3.2, который содержит несколько примеров выполнения команд, изменяющих базу данных `DB1` из листинга 3.1.

Листинг 3.2. Примеры изменения размеров базы данных

```
USE master
GO
-- Выполняем сжатие базы данных DB1, оставив в ней 10 % свободного места
DBCC SHRINKDATABASE (DB1, 10)
GO
--Добавляем в базу данных DB1 группу файлов с именем Group3
ALTER DATABASE DB1 ADD FILEGROUP Group3
GO
-- Добавляем в базу данных DB1 файл с именем file3 и включаем его
-- в группу файлов Group3
ALTER DATABASE DB1 ADD FILE
(NAME = file3,
 FILENAME = 'e:\Data\SQL data files\filedat3.ndf',
 SIZE = 5MB, MAXSIZE = 100MB, FILEGROWTH = 5MB)
TO FILEGROUP Group3
--Делаем группу файлов Group3 группой файлов по умолчанию
ALTER DATABASE DB1 MODIFY FILEGROUP Group3 DEFAULT
GO
```

3.4. Представление о хранении данных на SQL-сервере

С точки зрения пользователя, данные в реляционной базе данных хранятся в таблицах, которые в свою очередь состоят из строк или записей определенной структуры. Последовательность строк таблицы, с логической точки зрения, представляется непрерывной, хотя с точки зрения реляционной модели, сами по себе записи в отношении никак не упорядочены. Однако, глядя на

таблицу, пользователь интуитивно воспринимает такие понятия как текущая, предыдущая или последующая строка.

С точки зрения физического хранения данных, очевидно, что они должны быть как-то упорядочены и привязаны к конкретным областям диска, размер которых должен быть фиксирован, а сами эти области должны каким-то образом идентифицироваться в адресном пространстве диска. Для серверов баз данных такие элементарные области хранения данных носят название страниц, т. е. говорят, что данные на диске хранятся в страницах.

Страница — это блок фиксированной длины непрерывных виртуальных адресов памяти, который участвует в операциях чтения и записи как единое целое, т. е. требует только одного перемещения головки диска для своего осуществления. В SQL-сервере 2000 размер страницы данных равен 8 Кбайт (8 192 байт). Страницы объединяются в экстенты.

Экстент — это единица памяти, которую сервер выделяет как единое целое при размещении данных на диске по мере необходимости. В SQL-сервере 2000 размер экстента равен 8 страницам, а в SQL-сервере предыдущих версий размер экстента совпадает с размером страницы и равен 8 Кбайт.

В зависимости от способа хранения различают два типа таблиц: *таблицы кучи* и *кластерные таблицы*. Тип, к которому принадлежит таблица, зависит от того, имеется ли у этой таблицы кластерный индекс или нет.

3.4.1. Способ хранения кластерных таблиц

Кластерные таблицы — это таблицы, которые имеют кластерный индекс. Кластерный индекс, логически организованный в виде структуры сбалансированного дерева (B-дерева), — это такой индекс, в котором логический порядок ключевых значений определяет физический порядок соответствующих строк в таблице. Кластерный индекс у таблицы может быть только один, и его наличие приводит к тому, что строки таблицы хранятся на диске в строго определенном порядке.

Физически это организовано за счет того, что в кластерном индексе страницы данных, содержащие записи таблицы, совпадают со страницами индекса на листовом уровне. Страницы на каждом уровне индекса, включая страницы данных на уровне листьев, связаны в двусвязный список, но переход от одного уровня индекса к другому выполняется с использованием ключевых значений.

Каждая страница данных имеет специальный заголовок размером 96 байт, содержащий системную информацию, позволяющую серверу управлять доступом к страницам и данным. Этот заголовок включает в себя идентификатор таблицы, которой принадлежит страница, а также указатели на следующую и

предыдущую страницы. Строки данных заполняют остальную часть страницы. Для уровня листа кластерной таблицы выглядит это примерно так, как показано на рис. 3.1.

Рис. 3.1. Организация страниц данных кластерной таблицы на SQL-сервере

При вставке новой строки в кластерную таблицу, если в нужной странице нет свободного места, происходит процесс, который называется *расщеплением страниц*. Суть этого процесса состоит в перемещении половины строк или индексных входов из полностью заполненной страницы во вновь выделенную страницу таким образом, что вместо одной полностью заполненной страницы появляются две, заполненные наполовину. Так появляется место для новой строки или индексного входа с сохранением физического порядка сортировки. Доступ к конкретной строке кластерной таблицы осуществляется по алгоритму поиска в сбалансированном дереве по значению ключа, уникальному для каждой строки таблицы.

3.4.2. Работа с кучей

Таблицы кучи — это таблицы, расположенные в так называемой куче, которая представляет собой набор страниц данных, выделяемых по мере надобности для таблиц, не имеющих кластерного индекса, а также для самих индексов. При отсутствии кластерного индекса строки таблицы не имеют какого-либо конкретного порядка, и страницы данных, принадлежащие одной таблице, также никак не упорядочены и не связаны в список. В этой ситуации для вставки новой строки в таблицу в адресном пространстве просто выделяется следующий экстент данных, если пространство на уже выделенных страницах заполнено. Доступ к конкретной строке осуществляется путем полного сканирования таблицы.

Кроме кластерных индексов, SQL-сервер позволяет создание некластерных индексов. Когда строится некластерный индекс для таблицы кучи, SQL-сер-

вер использует идентификаторы строк в страницах индекса как указатели для доступа к конкретной строке данных. Когда же строится некластерный индекс поверх кластерной таблицы, то SQL-сервер использует ключ кластеризации в странице индекса в качестве указателя на вход кластерного индекса.

Кроме страниц, содержащих собственно данные, в любой базе данных имеются страницы, содержащие служебную информацию. К служебным страницам, в первую очередь, относятся так называемые IAM-страницы (Index Allocation Map, карта размещения индексов), которые содержат сведения об экстентах файла базы данных, используемых таблицами кучи или индексами. IAM-страницы также отображают экстенты, в которых хранятся данные текстовых типов (`TEXT`, `NTEXT` и `IMAGE`). Каждый из этих объектов имеет, по крайней мере, одну IAM-страницу для каждого файла со сведениями обо всех экстентах, распределенных для этого объекта в данном файле. Может существовать и более одной IAM-страницы в файле, если диапазон экстентов в этом файле превышает диапазон одной IAM-записи.

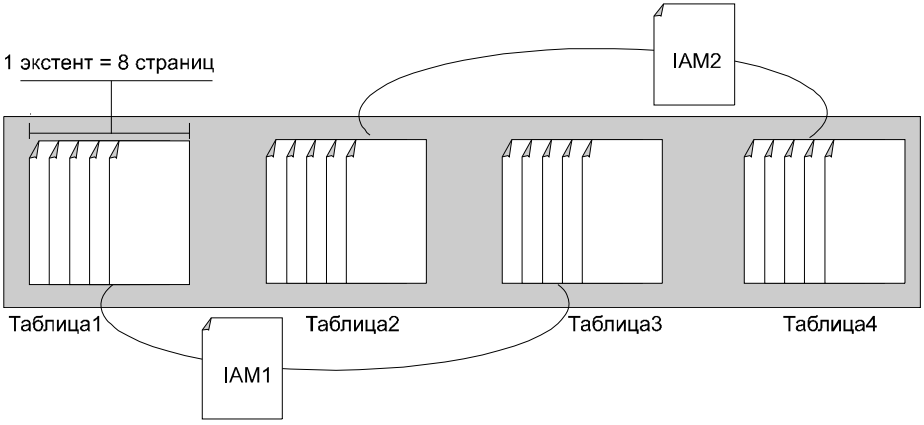


Рис. 3.2. Организация страниц данных на SQL-сервере

IAM-страницы выделяются по мере необходимости для каждого объекта и располагаются в файле случайным образом. В системной таблице `sysindexes` имеется указатель на первую IAM-страницу для каждого объекта кучи, а все IAM-страницы одного объекта связаны в цепочку, что выглядит, как показано на рис. 3.3.

Для навигации по куче SQL-сервер использует IAM-страницы. Напомним, что страницы данных и строки на этих страницах хранятся в произвольном порядке и не связаны вместе. Единственное место, где логическая взаимосвязь между страницами данных зарегистрирована, — это IAM-страницы. Доступ к данным в таблице кучи можно выполнить просматривая IAM-

страницы, чтобы найти экстенты, содержащие нужные страницы. Поскольку IAM-страницы представляют экстенты в том же порядке, в каком они расположены в файлах базы данных, то для доступа к конкретной строке куча сканируется последовательно. Это оказалось более эффективным, чем проход по связному списку случайно распределенных страниц кучи, использовавшийся в более ранних версиях SQL-сервера. Применение IAM-страниц для доступа к куче также означает, что строки из кучи обычно не возвращаются в том порядке, в котором они были вставлены.

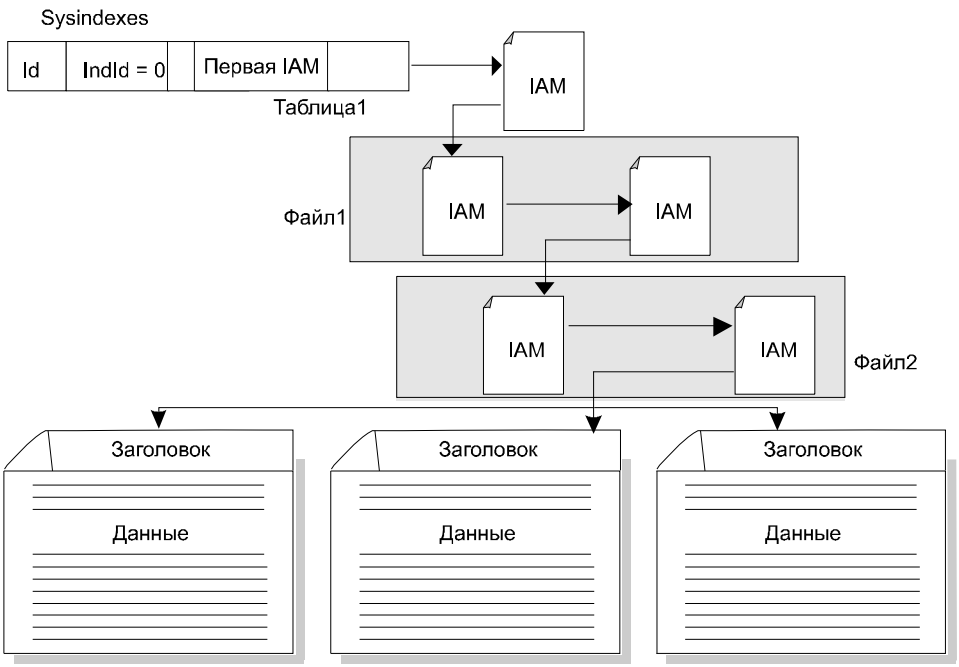


Рис. 3.3. Организация страниц данных на SQL-сервере

3.4.3. Вычисление размера таблицы кучи

Необходимо уметь вычислять размеры объектов, создаваемых SQL-сервером. Наиболее просто это выполняется для таблиц кучи, на примере которых и будет рассмотрен общий подход к вычислению объема объектов сервера.

Пусть известны значения:

- Num_Rows — число строк в таблице;
- Num_Cols — число столбцов;
- Fixed_Data_Size — сумма байтов во всех столбцах фиксированной длины;

- Num_Variable_Cols — число столбцов переменной длины;
- Max_Var_Size — максимальный размер всех столбцов переменной длины.

Тогда по этим данным может быть вычислен полный размер таблицы следующим образом.

Null_Bitmap = [2 + ((Num_Cols + 7) / 8)] — округляется до целого,
Variable_Data_Size = 2 + (Num_Variable_Cols × 2) + Max_Var_Size,
Row_Size = Fixed_Data_Size + Variable_Data_Size + Null_Bitmap + 4,

где Null_Bitmap — размер памяти, нужный для хранения информации о допустимости `NULL`-значений в столбцах, Variable_Data_Size — полный размер столбцов переменной длины, Row_Size — полный размер строки.

Приведенные формулы предполагают, что все столбцы переменной длины заполнены на 100 %. Если ожидается низкий процент их заполнения, то формула должна быть скорректирована. В заключительной формуле значение 4 представляет собой 4 байта, необходимые для хранения заголовка строки данных.

Далее можно вычислить количество строк, приходящихся на страницу данных (Rows_Per_Page), общее количество страниц, нужных для хранения таблицы (Num_Pages), и размер таблицы в байтах (Size):

Rows_Per_Page = (8096) / (Row_Size + 2),
Num_Pages = Num_Rows / Rows_Per_Page,
Size = 8192 × Num_Pages.

Все вычисленные величины должны быть округлены до ближайшего целого значения.

3.5. Управление выделением экстентов и поиск свободного места

Информация о свободном пространстве в базе данных SQL-сервера хранится достаточно просто. Имеются два типа специальных служебных страниц, содержащих сведения о свободном пространстве, к которым относятся:

- GAM-страницы (Global Allocation Map, общая карта размещения) — содержат информацию о том, какие экстенты выделены объектам. Одна GAM-страница соответствует 64 000 экстентам (или приблизительно 4 Гбайт данных). Каждая GAM-страница содержит по одному биту на каждый экстент. Если этот бит равен единице, то экстент свободен, а если бит равен нулю, то экстент занят;

- SGAM-страницы (Shared Global Allocation Map, разделяемая общая карта размещения) — содержат информацию о том, какие экстенты в данный момент распределены, но включают хотя бы одну пустую страницу. Каждая SGAM-страница содержит по одному биту на каждый экстент. Если этот бит равен единице, то экстент выделен объекту, но имеет хотя бы одну пустую страницу, а если бит равен нулю, то экстент либо полностью заполнен, либо не распределен.

Основываясь на данных из GAM- и SGAM-страниц, достаточно легко производить поиск экстентов для занесения новых данных.

Кроме GAM- и SGAM-страниц, SQL-сервер имеет еще один тип служебных страниц, называемых PFS-страницами (Page Free Space, страницы свободного пространства), которые содержат сведения о том, является ли конкретная страница пустой или заполненной и каков этот процент заполнения. То есть после того, как экстент выделен объекту, сервер использует информацию из PFS-страниц для того, чтобы осуществить вставку новой строки.

Информация на PFS-страницах также упакована очень плотно, так что на одну PFS-страницу записываются сведения приблизительно о 8 000 страниц данных. Это означает, что страниц подобного типа обычно немного и поэтому их легко постоянно хранить в оперативной памяти, а доступ к их данным и поиск происходит очень быстро.

Когда SQL-серверу необходимо вставить строку в таблицу и на текущей странице нет свободного места, он использует IAM- и PFS-страницы, чтобы найти страницу, в которую может быть помещена новая запись. Сначала просматриваются IAM-страницы для определения экстентов нужного объекта. Для каждого экстента просматриваются PFS-страницы, чтобы определить, имеется ли в нем место для размещения строки. SQL-сервер выделяет новый экстент объекту только тогда, когда не может быстро найти страницу в существующем экстенте с пространством, достаточным для размещения вставляемой строки. SQL-сервер распределяет экстенты в доступных группах файлов, используя пропорциональный алгоритм выделения, что означает, что каждый файл в группе файлов имеет примерно одинаковую степень заполнения.

3.6. Понятие о кэшировании данных

Основная цель базы данных состоит в том, чтобы хранить и находить данные, поэтому выполнение большого количества операций записи и чтения жесткого диска неизбежно. Дисковые операции, иначе называемые I/O (Input/Output, ввод/вывод) операциями, потребляют много ресурсов и требуют относительно продолжительного времени, особенно в сравнении со скоростью обработки данных процессором. Поэтому вопросы повышения эффективности операций ввода/вывода занимают одно из существенных мест в стратегии оптимизации производительности любой реляционной базы данных.

Современные аппаратные средства позволяют повысить скорость дисковых операций весьма существенно, однако этого недостаточно. Поэтому основная идея преодоления потенциально недостаточного быстродействия дисковых операций состоит в уменьшении их общего количества. Для этого SQL-сервер имеет такой механизм работы, как *буферное кэширование данных*.

Буферный кэш представляет собой специальную область оперативной памяти, в которую можно считывать данные с жесткого диска и сохранять их в течение нужного времени. Если информация занесена в буферный кэш и SQL-сервер об этом знает, то медленная дисковая операция считывания необходимой страницы данных будет заменена считыванием этой страницы из буферного кэша, что выполняется намного быстрее, т. к. кэш — это, по сути, оперативная память. Таким образом, данные считаются из файлов жесткого диска в буферный кэш и остаются в кэше на определенное время. В связи с использованием механизма кэширования различают два типа операций ввода/вывода.

Операция *физического чтения* состоит в чтении страницы данных с жесткого диска в *буферный кэш*. Операция *логического чтения* — это чтение страницы данных из буферного кэша для ее обработки. Аналогично определяются операции физической и логической записи.

Страница данных, измененная в кэше, но еще не записанная на диск, называется *грязной страницей*. Может оказаться, что перед выполнением физической записи на жесткий диск страница данных была многократно изменена в памяти в результате выполнения над ней последовательности операций логической записи.

Поскольку буферный кэш имеет конечный размер, в конце концов он будет целиком заполнен. В этом случае для занесения в буферный кэш новых данных какие-то из старых страниц должны быть стерты. Алгоритм кэширования работает таким образом, что стираются те данные, обращения к которым не было в течение достаточно продолжительного времени. При этом если эти страницы данных были подвергнуты операциям логической записи, то для них выполнится и физическая запись. Кроме того, физическая запись на диск происходит при выполнении процедуры контрольной точки для транзакции, в результате которой были изменены соответствующие страницы данных *(см. главу 6)*.

Каждый экземпляр SQL-сервера 2000 имеет свой собственный буферный кэш. Одна из основных задач оптимизации эффективности экземпляра SQL-сервера состоит в установлении оптимальных размеров его буферного кэша. Память, выделяемая для буферного кэша, должна быть достаточно большой, чтобы максимизировать отношение количества операций логического ввода/вывода к их физическому количеству. Однако слишком большой размер буферного кэша может привести к нехватке оперативной памяти для других нужд, что приведет к ее интенсивной подкачке или свопингу (swap, обмен), и

обмен информацией с диском снова возрастет. При стандартной настройке экземпляр SQL-сервера способен динамически управлять оптимальным размером буферного кэша. Такая конфигурация сервера является рекомендуемой.

В некоторых случаях, тем не менее, возможно управлять кэшированием данных вручную. Например, если все таблицы базы данных не могут быть помещены в буферный кэш, но известно, что имеется несколько таблиц небольшого объема, которые постоянно используются, их можно кэшировать принудительно, используя системную процедуру `sp_tableoption` с параметром `pintable`.

3.7. Средства настройки SQL-сервера

SQL-сервер способен автоматически настраиваться под конкретную среду, и эта возможность, установленная по умолчанию, является рекомендуемой. Следует только помнить, что оптимальная автонастройка выполняется не мгновенно, а требует некоторого времени на свое осуществление. Это время обычно занимает до нескольких часов в зависимости от интенсивности работы с сервером.

Хотя автоматическая настройка параметров сервера в большинстве случаев является рекомендуемой, однако иногда возникают ситуации, требующие задания определенных параметров вручную. Для задания параметров настройки сервера используется системная хранимая процедура `sp_configure`. С ее помощью можно задавать множество разнообразных настроек, некоторые из них приведены далее вместе с описанием типичных ситуаций, когда их рекомендуется использовать. Вызов указанной процедуры осуществляется следующим образом:

```
sp_configure ['название параметра настройки'], [заданное значение]
GO
RECONFIGURE WITH OVERRIDE
GO
```

Команда `RECONFIGURE` используется для того, чтобы заданная настройка вступила в силу немедленно.

Не все параметры настройки можно изменять по умолчанию. Существует ряд так называемых расширенных параметров (advanced options), которые можно изменять, только если предварительно была выполнена команда:

```
sp_configure 'show advanced options', 1
GO
RECONFIGURE
GO
```

Приведем некоторые рекомендации по использованию параметров настройки сервера с использованием процедуры `sp_configure`.

Замечание

К приведенным рекомендациям нужно относиться как к задачам экспериментального определения наиболее подходящих параметров конфигурации сервера для ситуаций, когда, во-первых, имеется в наличии сама проблема, а во-вторых, имеются средства мониторинга состояния сервера до и после изменения соответствующего параметра.

Если имеется очень активное OLTP-приложение с большим числом операторов `INSERT`, `UPDATE` и `DELETE`, то, возможно, что значение по умолчанию параметра настройки сервера `recovery interval` (интервал восстановления), равное нулю, не является наилучшим. При этом утилита Performance Monitor показывает, что имеются регулярные периоды во время выполнения процесса контрольной точки, когда диск загружен на 100 %. В этом случае может быть полезным увеличить значение параметра `recovery interval` до 5 или 10. Другая потенциальная причина использовать параметр `recovery interval` — это ситуация, когда база данных реализуется для OLAP бизнес-среды или хранилища данных и, следовательно, используется в основном для чтения, вообще не извлекая выгоды из короткого интервала восстановления.

Если сервер имеет достаточный объем оперативной памяти и на нем выполняется много конкурирующих интенсивных по использованию памяти запросов с сортировкой и хэшированием, то может оказаться полезным увеличение параметра конфигурации сервера `min memory per query` по сравнению с его значением по умолчанию, равным 1 024 Кбайт, до более высокого значения.

К настройкам сервера относятся такие параметры команды создания индексов, как `fill factor` и `index create memory`, которые соответственно определяют степень заполнения страниц вновь создаваемого индекса и количество памяти, используемое для выполнения сортировки при его создании. Эти параметры могут быть изменены исходя из соображений о том, какие индексы предстоит создавать. Например, увеличение принятого по умолчанию значения параметра `index create memory` позволит ускорить создание и перестройку индексов для больших таблиц. Нужно только помнить, что память, выделяемая для создания индексов, не может быть использована другими процессами.

К полезным параметрам настройки сервера относится опция `query governor cost limit`, которая устанавливает значение стоимости запроса таким образом, что запросы со стоимостью, превышающей установленную, выполняться не будут. На загруженном сервере эта опция окажется полезной, т. к. пользователь не сможет выполнять запросы, результатов которых приходится ожидать слишком долго.

ГЛАВА 4

Индексы как объект в реляционных базах данных

Индекс представляет собой физический объект базы данных, имеющий структуру сбалансированного B-дерева и применяемый оптимизатором для ускорения доступа к данным по сравнению с линейным сканированием таблицы. Дополнительным преимуществом индексов является то, что они могут использоваться для задания ограничения уникальности потенциального ключа базовой таблицы.

4.1. Структуры данных, ведущие к индексам

Индексы основаны на иерархических структурах данных, иначе называемых деревьями, а понятие дерева как структуры данных восходит к пониманию принципов связного распределения памяти. Кратко напомним суть этих структур данных.

Узел — элементарный объект для данных, к которому присоединены поля для хранения адресной информации (указателей).

Связный список — это множество физических записей, связанных указателями, поддерживаемыми самими записями. Связи часто изображаются просто стрелками, поскольку, как правило, безразлично, какую фактическую ячейку памяти занимает элемент, и поэтому физическая упорядоченность данных в связном списке отсутствует.

Схема со связями применима для структур более сложных, чем простые линейные списки. Допустимо существование переменного количества списков, размер которых непостоянен; любой узел одного списка может быть началом другого списка; в одно и то же время узлы могут быть связаны в несколько последовательностей, соответствующих различным спискам, и т. д.

Древовидная, или *иерархическая*, *структура данных* представляет собой такую разновидность связного списка, когда на каждый объект имеется ровно

одна связь, а сам он может ссылаться на 0, 1, 2 и более объектов. То есть структура дерева означает "разветвление" или такое отношение между "узлами", как и в обычных деревьях.

Определим формально *дерево* как конечное множество T, состоящее из одного или нескольких узлов, таких что:

- имеется один специально обозначенный узел, называемый *корнем* данного дерева;
- остальные узлы (исключая корень) содержатся в $m \geq 0$ попарно не пересекающихся множествах $T_1, ..., T_m$, каждое из которых, в свою очередь, является деревом.

Деревья $T_1, ..., T_m$ называются *поддеревьями* или *ветвями* данного корня.

Это определение является рекурсивным, т. е. дерево определено в терминах самих деревьев. Разумеется, никакого порочного круга здесь не возникает, поскольку деревья с $n > 1$ узлами определяются с использованием понятия дерева с количеством узлов меньшим, чем n. Следовательно, наше определение дает понятия деревьев с двумя узлами, тремя узлами, в конечном итоге с любым числом узлов. Рекурсивный характер деревьев налицо также и в природе, поскольку почки молодого дерева вырастают в ветви, имеющие собственные почки, которые дают новые ветви и т. д.

Из определения следует, что каждый узел дерева является корнем некоторого поддерева, которое содержится в этом дереве. Число поддеревьев данного узла называется степенью этого узла; узел с нулевой степенью — *концевым узлом*, иногда его называют *листом.* Неконцевые узлы часто называют узлами разветвления. В терминологии реализации индексов неконцевые узлы дерева будем называть *уровнем индекса*. На практике *узлами индексного дерева являются страницы памяти*.

Связные структуры данных, представляющие собой линейные списки, являются теоретически хорошо обоснованными и широко применяются на практике. Например, как мы знаем, по принципу двунаправленных связных списков хранятся страницы памяти, в которых располагаются записи таблиц реляционной базы данных.

Однако подобные связные структуры памяти не лишены недостатков, которые особенно заметны при росте их размеров. Решение задачи поиска записи по ключу для таких структур данных весьма неэффективно. Не вдаваясь в излишние подробности, можно считать, что более эффективной будет структура данных, в которой информация, необходимая для поиска записи, физически отделена от самой записи. То есть будет построена некоторая дополнительная структура данных, каждый элемент которой содержит всего два типа полей: интересующее нас значение поля записи из таблицы, например

БАНАНЫ, и все значения указателей на записи таблицы с таким значением в соответствующем поле. Подобные структуры данных носят название *инвертированного списка*. Другими словами, инвертированный список представляет собой структуру данных, каждый элемент в которой содержит указатели на все физические записи, содержащие заданное значение.

Развитием идеи инвертированного списка является так называемое *сбалансированное В-дерево*, являющееся той структурой данных, на основе которой строятся индексы реальных СУБД. С помощью В-дерева связываются воедино ключи поиска и указатели, локализующие записи данных. В-дерево состоит из иерархии индексных записей, последовательно расположенных на страницах памяти. Индексная запись состоит из двух полей: поля ключа и поля указателя. Индексные записи размещаются в индексных страницах памяти, образуя структуру В-дерева, которая формально может быть определена следующим образом.

Индексное В-дерево — это дерево с корнем, состоящее из страниц памяти и удовлетворяющее следующим условиям:

1. Все пути от корневой страницы к листовым страницам имеют одинаковую длину, т. е. для того чтобы пройти к любой из листовых страниц, нужно просмотреть одинаковое число страниц, не являющихся листовыми (сбалансированность дерева).
2. На любой странице индексные записи упорядочены по значениям ключа.
3. Каждая индексная запись с ключом *K* и указателем *P* на странице, не являющейся листовой, указывает на первую индексную запись страницы следующего уровня, причем все ключи в поддереве, на которое указывает *P*, строго больше или равны значению *K*. Другими словами, при движении от корневой страницы каждая некорневая страница индекса начинается со значения ключа, указатель которого привел на эту страницу со страницы предшествующего уровня.
4. Каждая индексная запись на листовой странице указывает на запись данных.

Это определение хорошо поясняется с помощью рис. 4.1.

Чтобы яснее представить структуру данных, реализующую сбалансированное В-дерево, приведем рисунок, который иллюстрирует процедуру вставки значений в дерево, изображенное на рис. 4.1. Этот процесс последовательно выполнен на рис. 4.2. При этом имеет место ситуация, когда данные, которыми являются ключи индексных записей, поступают на вход неупорядоченно. Вставка любого ключа должна быть выполнена таким образом, чтобы в любой момент времени для полученной структуры данных выполнялось определение В-дерева.

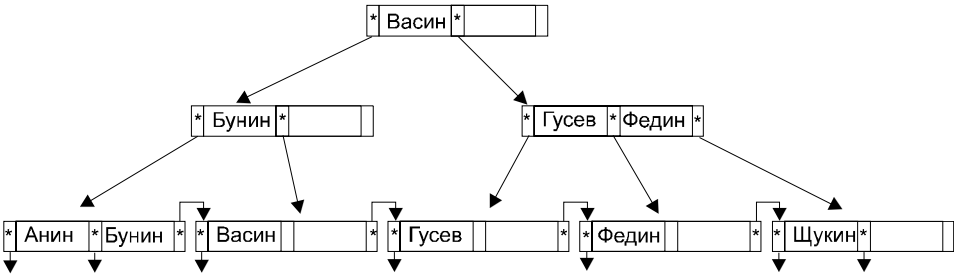


Рис. 4.1. Изображение индексного B-дерева

На рис. 4.2 иллюстрируется ситуация, когда размер страницы данных позволяет разместить всего две записи. В этой ситуации страницы данных (узлы дерева) заполняются наиболее быстро, и, следовательно, быстро наступает ситуация, когда для вставки нового значения необходимо выполнить операцию, которая носит название *расщепления страниц*.

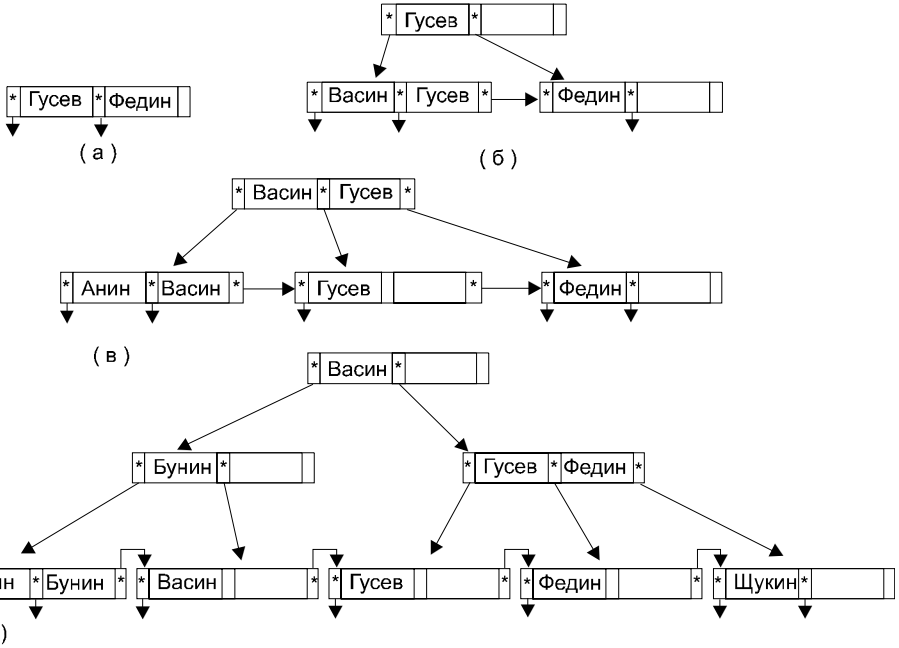


Рис. 4.2. Последовательность вставки данных в индексное B-дерево

На рис. 4.2, *а* показана структура дерева после вставки первых двух записей на страницу, которая в этом случае является и корневой, и листовой. После этого вставляется запись с ключом "Васин". Узел переполнен, и поэтому его нужно расщепить и переместить значение "Федин" в новый узел. Кроме того,

потребуется создать родительский узел, состоящий из самого правого значения ключа в левом узле, как показано на рис. 4.2, *б*. Затем вставляется запись с ключом "Анин". Она должна быть размещена слева от записи "Васин". Но в данном случае узел снова оказывается переполненным, поэтому его следует расщепить и переместить значение "Гусев" в новый узел. Кроме того, в родительский узел помещается значение "Васин", как показано на рис. 4.2, *в*. Затем вставляется запись с ключом "Бунин". Она должна быть размещена справа от записи "Анин". Но в данном случае узел снова оказывается переполненным, поэтому необходимо переместить значение "Васин" в новый узел. Кроме того, в родительский узел следует поместить значение "Бунин". Но родительский узел также оказывается переполненным и подлежит расщеплению, в результате чего для него потребуется создать новый родительский узел, как показано на рис. 4.2, *г*. Наконец, добавляется запись с ключом "Щукин", которая помещается в новый листовой узел, расположенный справа от записи "Федин", а значение ключа "Федин" помещается в соответствующий родительский узел, и мы приходим к дереву, изображенному на рис. 4.1.

Использование индексов в виде сбалансированных деревьев позволяет в большинстве случаев достичь высокого быстродействия при реализации поиска данных по ключу.

Как уже упоминалось в *главе 3*, на SQL-сервере существуют индексы двух типов — кластерные и некластерные. Между этими типами индексов имеются существенные различия как с точки зрения их физической реализации, так и при использовании индексов различных типов, о чем более подробно будет рассказано далее.

4.1.1. Механизм реализации кластерных индексов

При определении для таблицы кластерного индекса серверу фактически отдается приказание физически отсортировать данные в порядке индекса. Поэтому у любой базовой таблицы может быть только один кластерный индекс.

Одна из важнейших особенностей кластерного индекса состоит в том, что один элемент индекса на уровне индекса соответствует странице данных и эти значения отсортированы. На уровне листа кластерного индекса страницы памяти, предназначенные для хранения индексной информации, физически совпадают со страницами данных базовой таблицы, для которой этот кластерный индекс создан. Средний размер кластерного индекса на индексном уровне составляет около 5 % от размера таблицы, но может варьироваться в зависимости от размера индексируемого столбца. Когда создается кластерный индекс, SQL-сервер временно использует дисковое пространство текущей базы данных для сортировки. Это пространство равно примерно 120—

150 % от размера таблицы. Когда индекс создан, используемое в процессе его создания дисковое пространство автоматически возвращается базе данных.

Итак, в кластерном индексе данные на уровне листа физически отсортированы. Каждая запись уровня листа кластерного индекса имеет так называемый уникальный *ключ кластеризации*. Неповторяемость ключевых значений поддерживается явно, если задано ограничение `UNIQUE`, или неявно с помощью ключей кластеризации на уровне, недоступном пользователю. Подобная организация данных приведена на рис. 4.3.

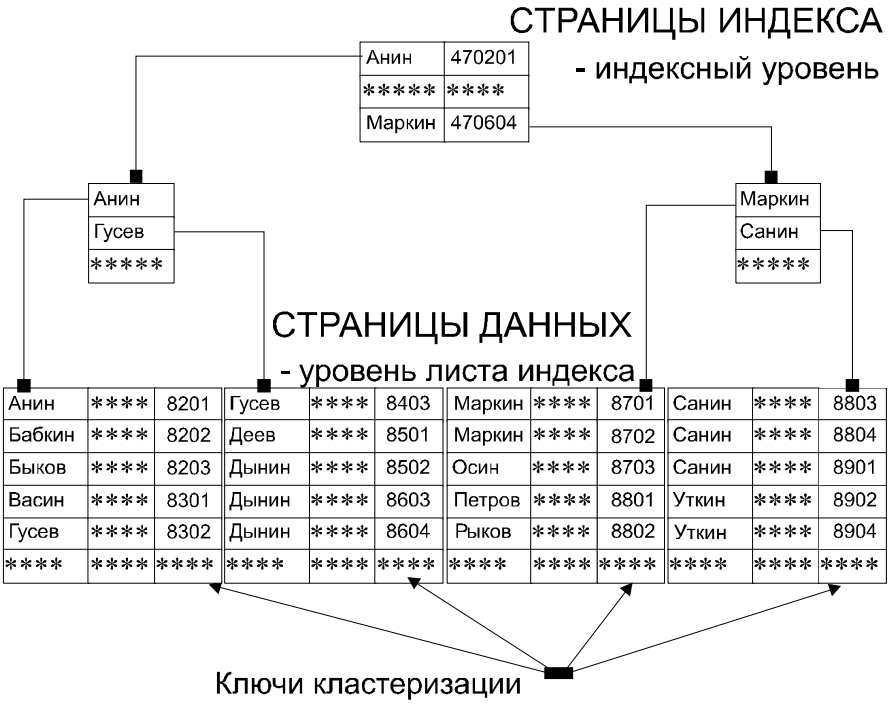


Рис. 4.3. Структура организации кластерного индекса

Для кластерного индекса характерно, что индекс этого типа и данные совместно используют одну и ту же запись в системной таблице `sysindexes`. Рассмотрим пример прохода по дереву кластерного индекса.

4.1.2. Механизм реализации некластерных индексов

Разумеется, одного индекса для таблицы недостаточно, т. к. при работе с базой данных приходится обращаться к данным разными способами. Некластерные индексы позволяют организовать доступ к данным с использованием

В-дерева в порядке, который отличен от порядка физической сортировки данных. Другими словами, некластерный индекс — это физический объект, задающий логический порядок сортировки данных в базовой таблице. Некластерных индексов у таблицы может быть несколько (до 249).

Важнейшая особенность некластерных индексов заключается в том, что один элемент индекса приходится на одну запись данных.

Для способа реализации некластерного индекса является важным то обстоятельство, строится ли он поверх кластерного индекса или нет. В последнем случае говорят, что некластерный индекс построен поверх кучи.

Рис. 4.4. Структура организации некластерных индексов

Когда строится некластерный индекс поверх кучи, сервер использует *идентификаторы строк* в страницах индекса (`row ID` или `BOOKMARK`), указывающие на строки в страницах данных. В этих идентификаторах строк как раз и хранится информация о расположении данных. Для работы со страницами кучи используются IAM-страницы *(см. главу 3)*.

Когда строится некластерный индекс поверх таблицы с кластерным индексом, сервер использует *ключ кластеризации* (clustering key) в странице индекса, который указывает на уровень листа кластерного индекса, т. е. собственно на данные.

Поясним сказанное с помощью рис. 4.4.

В зависимости от того, имеется ли в таблице ключ кластеризации или она является таблицей кучи, по-разному происходит доступ к данным. Для примера рассмотрим поиск по диапазону для обоих случаев.

4.1.3. Пример поиска по диапазону для таблицы кучи

Рассмотрим пример прохода по дереву некластерного индекса для таблицы кучи. В куче отсортированы только ключевые значения на уровне листа некластерного индекса, а сами данные не отсортированы никак. Поэтому при поиске диапазона данных сервер должен обратиться к каждой странице данных по идентификатору строки для поиска каждого конкретного значения из диапазона поиска. Другими словами, читается одна страница данных базовой таблицы для каждого ключевого значения из диапазона.

Предположим, что нужно выполнить запрос

```
SELECT * FROM TABLE1 WHERE ФИО BETWEEN 'Маркин' AND 'Рудин'
```

На первом этапе определяется, что существует индекс для столбца `ФИО` и что он полезен при поиске строк, содержащих фамилии между "Маркин" и "Рудин". Принимается решение начать индексный поиск. Дальнейшие шаги, которые выполняются для реализации этого запроса, приведены в табл. 4.1.

4.1.4. Пример поиска по диапазону при отображении некластерного индекса в ключ кластеризации

При доступе к диапазону значений с использованием некластерного индекса при наличии ключа кластеризации нужно пройти по кластерному индексу для каждого значения из диапазона.

Таблица 4.1. *Поиск по диапазону для таблицы кучи*

№	Действия и решения	Пояснения
1	Местоположение: *корневой уровень*, страница индекса 12, первое значение ключа Сравнение "Маркин" >= "Антонов" Решение: результат сравнения — истина Действие: перейти к следующему значению ключа	Выполняется сканирование корневой страницы до тех пор, пока не определится индексная страница, на которую нужно перейти
2	Местоположение: *корневой уровень*, страница индекса 12 Сравнение "Маркин" >= "Маркин" Решение: результат сравнения — истина Действие: "Маркин" — последнее значение ключа на корневой странице, перейти к странице не на уровне листа, на которую указывает значение ключа (страница 28)	
3	Местоположение: *индексный уровень*, страница индекса 28 Сравнение "Маркин" >= "Маркин" Решение: результат сравнения — истина Действие: "Маркин" — перейти к странице на уровне листа, на которую указывает значение ключа (страница 61)	Проходим по дереву индекса на индексном уровне
4	Местоположение: *уровень листа*, страница индекса 61, первое значение диапазона Действие: сканировать листовую страницу индекса и, обращаясь по идентификаторам строк к страницам данных, получать результаты запроса. Когда встретится фамилия "Рудин", поиск прекращается	

Пусть нужно выполнить запрос

```
SELECT * FROM TABLE1 WHERE COL_ID BETWEEN 5678 AND 6078
```

Определяется, что существует индекс для столбца `COL_ID` и что он полезен при поиске строк, содержащих номера между 5678 и 6078. Принимается решение начать индексный поиск.

При выполнении этого запроса SQL-сервер предпринимает следующие шаги, представленные в табл. 4.2.

Таблица 4.2. *Поиск по диапазону некластерного индекса для кластерной таблицы*

№	Действия и решения	Пояснения
1	Местоположение: некластерный индекс, *корневой уровень*, страница индекса 12, первое значение ключа Сравнение 5678 >= 1234, результат сравнения — истина Действие: перейти к следующему значению ключа	Начинаем сканирование корневой страницы до тех пор, пока не определится индексная страница, на которую нужно перейти
2	Местоположение: некластерный индекс, *корневой уровень*, страница индекса 12 Сравнение 5678 >= 5678, результат сравнения — истина Решение: результат сравнения — истина, 5678 — последнее значение ключа на странице Действие: перейти к странице не на уровне листа, на которую показывает ключевое значение 5678 (страница 28)	Движемся по дереву индекса от корня к первому значению диапазона поиска на уровне листа
3	Местоположение: некластерный индекс, *индексный уровень*, страница индекса 28, значение ключа 5678, последнее значение ключа на странице 6078 Сравнение 5678 >= 5678 AND 5678 <= 6078 Решение: результат сравнения — истина Действие: перейти к странице на уровне листа, на которую показывает ключевое значение (страница 61)	
4	Местоположение: некластерный индекс, *уровень листа*, страница индекса 61 Сравнение 5678 >= 5678 AND 5678 <= 6078 Решение: результат сравнения — истина Действие: пройти по кластерному индексу и найти строку	Найдено первое значение диапазона поиска на уровне листа
5	Местоположение: некластерный индекс, *уровень листа*, страница индекса 61 Сравнение 5778 >= 5678 AND 5778 <= 6078 Решение: результат сравнения — истина Действие: пройти по кластерному индексу и найти строку	Движемся последовательно по диапазону поиска, значения которого находятся в записях на уровне листа некластерного индекса

Таблица 4.2 *(окончание)*

№	Действия и решения	Пояснения
6	Местоположение: некластерный индекс, *уровень листа*, страница индекса 61 Сравнение 5878 >= 5678 AND 5878 <= 6078 Решение: результат сравнения — истина Действие: пройти по кластерному индексу и найти строку	Каждое значение диапазона поиска отображается в ключ кластеризации, проходя по которому извлекаем полную информацию из записи
7	Местоположение: некластерный индекс, *уровень листа*, страница индекса 61 Сравнение 5978 >= 5678 AND 5978 < =6078 Решение: результат сравнения — истина Действие: пройти по кластерному индексу и найти строку	
8	Местоположение: некластерный индекс, *уровень листа*, страница индекса 61 Сравнение 6078 >= 5678 AND 6078 <= 6078 Решение: результат сравнения — истина Действие: пройти по кластерному индексу и найти строку	
	..... и т. д. При движении по страницам листа некластерного индекса переходим по мере перебора значений диапазона поиска на следующие листовые страницы	
n	Местоположение: некластерный индекс, *уровень листа*, страница индекса 71 Сравнение 7678 >= 5678 AND 7678 <= 6078 Решение: результат сравнения — ложь Действие: прекратить сканирование некластерного индекса на уровне листа. Найден конец диапазона	

4.2. Использование индексов с точки зрения бизнес-среды базы данных

Стратегия использования индексов в базе данных определяется тем, к какому из двух типов (OLTP или OLAP) относится бизнес-среда этой базы данных *(см. главу 2)*. Напомним, что если для OLTP-системы основным показателем

быстродействия служит совокупная скорость выполнения элементарных транзакций, модифицирующих данные, то для OLAP-приложения наиболее важна производительность поиска и выборки данных. В реальной жизни любая база данных будет сочетать в себе черты как OLTP, так и OLAP-системы — ведь данные в нее должны как-то заноситься, и трудно представить себе базу данных, к которой вообще не формулируются никакие запросы. Индексы являются средством ускорения операций анализа данных, но они снижают скорость операций по их модификации.

Без индексов операция вставки выполняется фактически с одинаковой скоростью независимо от размера и количества атрибутов таблицы. Если же в таблице имеются индексы, то при вставке записи индексы должны быть перестроены для поддержания порядка их сортировки. На перестройку каждого индекса тратится время, поэтому скорость выполнения операции вставки обычно *уменьшается при увеличении количества индексов* у таблицы и *мало зависит от числа строк* в таблице.

Рассмотрим операции обновления и удаления записей из таблицы. Прежде чем обновить или удалить запись, ее необходимо найти. Если таблица не индексирована, то единственным способом поиска является последовательное сканирование таблицы. В этом случае скорость операций обновления и удаления существенно увеличивается с ростом количества записей в таблице и не зависит от количества атрибутов. Но если имеется индекс, полезный для поиска записи, то используется он, и сам поиск нужной записи производится очень быстро, практически независимо от количества строк и атрибутов в таблице. Однако если для таблицы объявлено несколько индексов, то само выполнение операции обновления или удаления уже найденной записи приводит к перестраиванию этих индексов, на что расходуется дополнительное время, и оно обычно в целом больше, чем время, сэкономленное при поиске нужной записи по индексу. Таким образом, скорость выполнения операций обновления и удаления также *уменьшается при увеличении количества индексов* у таблицы и *мало зависит от числа строк* в таблице. Можно предположить, что чем больше атрибутов имеет таблица, тем больше для нее будет объявлено индексов. Эта зависимость, конечно, не прямая, но *при одинаковых подходах* к физическому моделированию обычно так и происходит. Поэтому правомочно допущение того, что *чем больше атрибутов имеют отношения*, разработанные в ходе логического моделирования, *тем медленнее будут выполняться операции обновления данных* за счет затраты времени на перестройку большего количества индексов.

Таким образом, искусство оптимизации производительности работы базы данных зависит от способности разработчика найти наилучший компромисс между противоречивыми способами ускорения работы транзакций (чем меньше индексов, тем лучше) и запросов на выборку и поиск данных (чем

больше индексов, тем лучше), для повышения производительности которых индексы необходимы.

Из всего вышесказанного следует, что статические таблицы, которые редко подвергаются изменениям, можно сильнее нагружать индексами, чем динамические.

4.3. Средства работы с индексами на MS SQL-сервере

4.3.1. Создание индексов

Индексы можно создавать как с помощью визуальных средств, так и командами T-SQL. Следует помнить, что визуальные средства являются только оболочкой для команд T-SQL, поэтому именно командам будет уделено основное внимание. Причем мы не ставим задачу подробного разбора полного синтаксиса всех возможных команд работы с индексами. Интереснее дать некоторые рекомендации, связанные с оптимальными способами применения средств работы с индексами на SQL-сервере.

Если это возможно, лучше создавать индексы на столбцах, которые имеют целочисленные, а не строковые типы данных. Совсем неудачно использовать для построения индексов столбцы с типом данных `FLOAT` или `REAL`, и эти типы данных совершенно не подходят для первичных ключей. Строковые типы данных подходят для индексов, если соответствующие столбцы не велики по размеру, потому что с ростом ширины столбца сильно увеличиваются затраты на поддержание индекса. Нельзя строить индексы по типам данных `TEXT` и `IMAGE`.

Команда создания индекса имеет следующий вид

```
CREATE [ UNIQUE ] [ CLUSTERED | NONCLUSTERED ]
INDEX index_name ON { table | view }
( column [ ASC | DESC ] [, ...n ])
 [ WITH PAD_INDEX | FILLFACTOR = FILLFACTOR | IGNORE_DUP_KEY |
 DROP_EXISTING | STATISTICS_NORECOMPUTE |
 SORT_IN_TEMPDB ]
[ ON filegroup ]
```

Эта команда создает индекс с именем `index_name` для указанной таблицы `table` или представления `view` по перечисленным в параметре `column` столбцам. Для того чтобы команда создания индекса могла реализовать соединение, в контексте которого ее предстоит выполнять, должно быть обязательно установлено соединение со следующими параметрами:

- ANSI_NULLS = ON;
- ANSI_PADDINGS = ON;
- ANSI_WARNINGS = ON;
- ARITHABORT = ON;
- CONCAT_NULL_YIELDS_NULL = ON;
- QUOTED_IDENTIFIER = ON;
- NUMERIC_ROUNDABORT = OFF.

Кратко опишем параметры команды создания индекса (далее будут приведены рекомендации по их использованию):

- UNIQUE — говорит о том, что задается индекс, реализующий ограничение уникальности для своих столбцов, которое обозначает, что данные в этих столбцах не могут повторяться. По умолчанию индексы не являются уникальными;
- CLUSTERED — создается кластерный индекс;
- NONCLUSTERED — создается некластерный индекс (значение по умолчанию);
- PAD_INDEX — задает степень заполнения страниц на уровне индекса. По умолчанию, когда значение PAD_INDEX не задано, остается место для вставки на уровне страниц индекса двух индексных записей максимальной длины. Если значение PAD_INDEX задано, то это означает, что значение FILLFACTOR распространяется и на степень заполнения страниц на уровне индекса;
- FILLFACTOR — задает степень заполнения данными страниц индекса на уровне листа (значение по умолчанию FILLFACTOR = 0, что обозначает полное заполнение страниц на уровне листа индекса);
- IGNORE_DUP_KEY — говорит о том, что нужно игнорировать совпадающие значения в столбцах индекса;
- DROP_EXISTING — требует удалить существующие индексы перед созданием нового;
- STATISTICS_NORECOMPUTE — приказывает не перестраивать статистику таблиц после создания индекса;
- SORT_IN_TEMPDB — требует выполнять сортировку при создании индекса в базе данных TEMPDB;
- ON filegroup — указывает, в какой группе файлов нужно физически создать индекс (значение по умолчанию — PRIMARY).

Когда индекс создается или восстанавливается, можно задать значение параметра FILLFACTOR. Этот параметр определяет, какой процент индексной

страницы данных будет заполнен на уровне листа индекса. Значение `FILLFACTOR = 100` означает, что каждая индексная страница должна быть полностью заполнена при создании индекса. Если кластерный индекс имеет `FILLFACTOR = 100`, то это означает, что при вставке записи всегда будет происходить процесс разбиения страницы, потому что не имеется места для новых данных в существующих страницах. Многочисленные разбиения страниц могут замедлить эффективность SQL-сервера. С другой стороны, если значение `FILLFACTOR` будет слишком мало, индекс займет неоправданно много места в памяти, т. к. будет первоначально состоять из большого количества почти пустых страниц. В этой ситуации для доступа к данным нужно выполнить большое количество элементарных операций чтения страниц, что само по себе замедляет производительность, и, кроме того, буферный кэш заполняется пустой информацией.

Возникает вопрос, какое же значение `FILLFACTOR` является идеальным? Это значение зависит от величины отношения R / W, в числителе которого находится среднее значение количества операций чтения, а в знаменателе — среднее значение количества операций записи, выполняемых для компонентов индекса. Как правило, можно следовать приведенным далее рекомендациям:

- редко изменяемые таблицы (выполняется от 100 до 1 операции чтения на одну операцию записи): `FILLFACTOR = 100`;
- часто изменяемые таблицы (количество операций чтения превышает количество операций записи): `FILLFACTOR = 50-70`
- промежуточная ситуация: `FILLFACTOR = 80-90`.

Вероятно, придется экспериментировать, чтобы найти оптимум для фактора заполнения в каждом конкретном случае, потому что отношение R / W загадочно.

Помочь определить величину отношения R / W может утилита Performance Monitor с использованием двух счетчиков: `Physical Disk Object: % Disk Read Time и % Write Time`. Просмотр этих результатов позволяет получить хорошее представление о соотношении интенсивности записи и чтения в базе данных. Еще один счетчик, который может оказаться полезным, — это счетчик количества расщеплений страниц. Он называется Pages Splits/Sec и измеряет среднее число разбиений индексных страниц за секунду, которые происходят на SQL-сервере. Для повышения эффективности количество расщеплений страниц должно быть как можно меньше, добиться этого можно подбирая подходящее значение параметра `FILLFACTOR` экспериментальным путем.

Если явно не определять значение `FILLFACTOR`, приняв значение по умолчанию `FILLFACTOR = 0`, то это означает то же самое, что и 100 % (страницы ин-

декса уровня листа заполнены полностью, но имеется пустое место на индексных страницах верхних уровней). В большинстве случаев это значение по умолчанию не является хорошим выбором, особенно для кластерных индексов.

Если имеется таблица с монотонно возрастающим кластерным индексом, созданным по автоинкрементному столбцу, и если записи этой таблицы не должны изменяться и у таблицы нет столбцов типа VARCHAR, то идеальное значение FILLFACTOR = 100, т. к. вставка записей в этом случае происходит заранее упорядоченно и расщепления страниц произойти при вставке не может.

В команде CREATE INDEX опция FILLFACTOR имеет подопцию, называемую PAD_INDEX. Если значение PAD_INDEX явно не определено, то FILLFACTOR применяется только к страницам листа в индексе. Но если значение PAD_INDEX задано, то FILLFACTOR будет применяться ко всем страницам на всех уровнях индексного дерева. В табл. 4.3 показано, как взаимодействуют параметры FILLFACTOR и PAD_INDEX.

Таблица 4.3. *Взаимодействие параметров FILLFACTOR и PAD_INDEX*

FILLFACTOR	PAD_INDEX задан		PAD_INDEX не задан	
	Страницы на уровне листа индекса	Индексные страницы	Страницы на уровне листа индекса	Индексные страницы
1–99	Заполнение до процента, указанного в FILLFACTOR		Заполнение до процента, указанного в FILLFACTOR	Максимально допустимое заполнение
0 или 100	Максимально допустимое заполнение			

При использовании параметра PAD_INDEX нужно иметь в виду, что по умолчанию SQL-сервер всегда оставляет место для двух и более строк максимального размера на уровне листа независимо от коэффициента заполнения.

Если нужно перестроить кластерный индекс с применением команды CREATE INDEX для таблицы, которая также имеет некластерные индексы, максимальной эффективности можно достичь при использовании опции DROP_EXISTING наряду с командой CREATE INDEX. Опция DROP_EXISTING предотвращает повторную перестройку некластерных индексов таблицы.

Если нужно создать индексы на больших таблицах, ускорить их создание можно используя опцию SORT_IN_TEMPDB, доступную с командой CREATE INDEX. Эта опция дает возможность SQL-серверу использовать базу данных TEMPDB вместо текущей базы данных, чтобы сортировать в нее данные при создании

индексов. Применение этой опции оправдано, только если текущая база данных и база данных TEMPDB находятся на разных дисковых устройствах.

Параметр DROP_EXISTING позволяет изменять характеристики индекса, например некластерный индекс на кластерный, но не наоборот. С его помощью нельзя добавить ограничение уникальности, а также изменить порядок или названия столбцов, входящих в индекс.

Замечание

Результаты изменений в наборе индексов базы данных должны быть своевременно отражены в сопроводительной документации в нотации той модели, которая выбрана для описания базы данных. Если использовано какое-либо CASE-средство, то после создания новых индексов можно просто выполнить процедуру синхронизации модели с базой данных.

4.3.2. Получение информации об индексах

Информацию об индексах, которые созданы в базе данных, можно получать разными способами. Следует понимать, что изначально эта информация хранится в специальной системной таблице sysindexes. При создании любого индекса в эту таблицу заносится запись, содержащая всю информацию, касающуюся созданного индекса. Способы получения этой информации основаны на обращении к таблице sysindexes напрямую или с помощью хранимых процедур. Возможно также использование операторов средства DBCC.

Системная таблица *sysindexes*

Системная таблица sysindexes, которая имеется в каждой базе данных, содержит сведения об объектах кучи, т. е. о таблицах, не имеющих кластерных индексов, а также о самих индексах любых типов. Перечислим некоторые основные столбцы таблицы sysindexes:

- Id: идентификатор таблицы для Indid равен 0 или 255, в остальных случаях это идентификатор таблицы, которой индекс принадлежит;
- Indid: этот столбец указывает тип объекта, информация о котором хранится в строке: Indid = 0 — для таблицы кучи, Indid = 1 — для кластерного индекса, значение Indid большее, чем 1, соответствует некластерному индексу или статистике столбца, а значение Indid = 255 указывает на то, что таблица имеет данные типа TEXT или IMAGE;
- Dpages: это количество страниц данных, используемых для индекса при любых значениях Indid, кроме 255. Если Indid = 255, то Dpages = 0;
- OrigFILLFACTOR: это значение фактора заполнения индекса (FILLFACTOR), используемое при создании индекса;

- `Statversion`: это количество изменений статистики;
- `Used`: если `Indid` = `0` или `1`, то это значение выдает число полных страниц, используемых для всего индекса и данных таблицы. Если `Indid` = `255`, то выводится число страниц данных, использованных для хранения изображения или текста. Во всех других случаях — это число страниц в индексе;
- `FirstIAM`: указывает на первую IAM-страницу в цепочке IAM-страниц, управляющих пространством, выделяемым таблице кучи или индексу. Сервер использует IAM-страницы, чтобы найти страницы данных, потому что они не связаны между собой.

Примером работы с системными таблицами может служить запрос, который выдает информацию о пользовательских таблицах, хранящихся в куче, а также обо всех индексах базы данных:

```
SELECT I.Id, Indid, I.name AS INDEX_NAME,
O.name AS TABLE_NAME, O.type
FROM sysindexes I INNER JOIN sysobjects O ON I.Id = O.Id
WHERE O.type='u'
ORDER BY I.ID
```

В этом запросе системная таблица `sysindexes` соединяется с системной таблицей `sysobjects` по системному идентификатору объекта. В опции `WHERE` указывается тип объекта `(O.type ='u')`, что как раз и говорит о том, что выбирается пользовательский объект. Результаты работы этого запроса приведены в табл. 4.4. Эти результаты несколько сокращены и в них вручную добавлен столбец комментариев.

Таблица 4.4. *Объекты кучи базы данных*

Id	Indid	INDEX_NAME	TABLE_NAME	type	Комментарий
55671246	1	PK_Товары	ТОВАРЫ	U	Таблица ТОВАРЫ с кластерным индексом и статистикой
55671246	2	_WA_Sys_Товар_035179CE	ТОВАРЫ	U	
75147313	1	PK_РАСХОДЫ	РАСХОДЫ	U	Таблица РАСХОДЫ с кластерным индексом и статистикой
75147313	2	_WA_Sys_Дата_расх_047AA8	РАСХОДЫ	U	
123147484	0	НОМЕРА	НОМЕРА	U	Таблица кучи без кластерного индекса

Таблица 4.4 *(окончание)*

Id	Indid	INDEX_NAME	TABLE_NAME	type	Комментарий
135671531	1	PK_РАСХОДЫ_ТОВАРЫ	РАСХОДЫ_ТОВАРЫ	U	Таблица с кластерным индексом, статистикой и некластерным индексом
135671531	2	_WA_Sys_Товар_Ид_08162EE	РАСХОДЫ_ТОВАРЫ	U	
135671531	3	РАСХОДЫ_ТОВАРЫ4	РАСХОДЫ_ТОВАРЫ	U	
663673412	0	РАСХОДЫ_1	РАСХОДЫ_1	U	Таблица кучи со статистикой и некластерным индексом
663673412	2	IX_РАСХОДЫ_1_N_расх	РАСХОДЫ_1	U	
663673412	3	_WA_Sys_Товар_278EDA44	РАСХОДЫ_1	U	

Замечание

Признаком того, что объект представляет собой статистику *(см. разд. 5.2)*, является тот факт, что его имя начинается с префикса "_WA_Sys_".

Системные процедуры и средства DBCC

К системным хранимым процедурам, которые выдают информацию об индексах, относятся sp_statistics и sp_helpindex.

Процедура sp_statistics выдает наиболее полную информацию обо всех индексах заданной базовой таблицы или индексного представления, имена которых являются единственным обязательным аргументом этой процедуры.

Выходной набор этой процедуры приведен в табл. 4.5.

Таблица 4.5. *Результаты работы процедуры sp_statistics*

Имя столбца	Тип данных	Описание
TABLE_QUALIFIER	sysname	Имя базы данных. Допустимо NULL-значение, если строка относится к таблице, а не к индексу
TABLE_OWNER	sysname	Имя владельца таблицы. NULL-значение недопустимо
TABLE_NAME	sysname	Имя таблицы. NULL-значение недопустимо
NON_UNIQUE	smallint	Признак уникальности индекса. NULL-значение недопустимо 0 = Unique 1 = Not Unique

Таблица 4.5 *(окончание)*

Имя столбца	Тип данных	Описание
INDEX_QUALIFIER	sysname	Имя владельца индекса. Для SQL-сервера всегда совпадает с именем владельца таблицы, т. к. индексы может создавать только владелец
INDEX_NAME	sysname	Имя индекса. NULL-значение недопустимо
TYPE	smallint	Тип индекса. NULL-значение недопустимо. SQL-сервер может выдать 0, 1, 2 или 3. Это не аналогия столбцу Indid системной таблицы sysobjects 0 — статистика таблицы 1 — кластерный индекс 2 — хешированный индекс 3 — индекс другого типа
SEQ_IN_INDEX	smallint	Номер столбца в индексе. Существенно для составных индексов
COLUMN_NAME	sysname	Имя столбца для каждого из выданных для TABLE_NAME столбцов. NULL-значение недопустимо
COLLATION	char(1)	Порядок сортировки. Может иметь значения: A — по возрастанию D — по убыванию NULL — не использовано
CARDINALITY	int	Число строк в таблице или количество уникальных значений в индексе. Допустимо NULL-значение для индексов с малой селективностью
PAGES	int	Количество страниц данных для хранения таблицы или индекса
FILTER_CONDITION	varchar(128)	Всегда возвращается NULL-значение

Информацию об одном конкретном индексе можно получить с помощью системной хранимой процедуры sp_helpindex.

К средствам DBCC, содержащим ту или иную информацию об индексах, относятся команды DBCC SHOW_STATISTICS *(см. разд. 5.2)*, DBCC CHECKALLOC и DBCC CHECKTABLE.

4.3.3. Сопровождение индексов

При модификации таблиц возникает такое явление, как *фрагментация данных*, суть которого состоит в следующем. Когда для таблицы с индексом

вставляются или удаляются строки, сервер подстраивает индексы, поддерживая заданный порядок сортировки. Вставка новых строк и вызванное этим расщепление страниц приводят к тому, что данные, относящиеся к одному объекту (таблице или индексу), хранятся не сплошным фрагментом, а разбросаны по всему диску, поэтому головке диска нужно выполнять большое количество перемещений для чтения и записи дефрагментированных страниц данных. Удаление записей приводит к появлению незаполненных фрагментов на страницах данных. В результате этого таблица будет занимать больше страниц дисковой памяти, чем это реально нужно. Таким образом, фрагментация данных ведет к увеличению количества элементарных операций ввода/вывода, необходимых для доступа к данным. Учитывая, что операции ввода и вывода относятся к потенциально медленным операциям сервера в целом, с фрагментацией нужно бороться.

Эффективность воздействия дефрагментации на производительность SQL-сервера зависит от размера таблиц. Например, если таблица занимает меньше чем 100 страниц данных, перестройка и дефрагментация индексов не принесут ощутимой пользы. Это объясняется тем, что другие средства, например такие, как аппаратные дисковые кэши, буферный кэш самого сервера, а также так называемое опережающее чтение, скрывают отрицательный эффект фрагментации. Но для очень большой таблицы эффект от переиндексации может оказаться очень ощутимым.

В связи с возникновением такого явления, как фрагментация страниц данных, периодически необходимо реорганизовывать все индексы всех таблиц с целью ее устранения. Период выполнения дефрагментации зависит от типа среды. Так, фрагментация менее вредна в OLTP-среде, но для среды с преобладанием OLAP-функций средства, направленные на борьбу с фрагментацией данных, должны быть более эффективными и применяться чаще.

В SQL-сервере есть два способа борьбы с фрагментацией. Первый способ состоит в удалении и последующем воссоздании кластерного индекса таблицы. При этом применяется пара команд `DROP INDEX` и `CREATE INDEX`. Второй способ считается более предпочтительным и состоит в использовании такого средства перестройки индексов, как команда `DBCC DBREINDEX`, сокращенный синтаксис которой таков:

```
DBCC DBREINDEX('database.owner.table_name'[,index_name][,FILLFACTOR])
```

Реорганизацию базы данных можно также выполнить с помощью утилиты командной строки Sqlmaint.exe.

Следует помнить, что при выполнении перестройки кластерного индекса автоматически будут перестроены и все некластерные индексы таблицы. К сожалению, команда `DBCC DBREINDEX` автоматически не может перестроить все

индексы во всех таблицах базы данных, т. к. из синтаксиса команды следует, что она может работать только с одной таблицей. Следующий программный код (листинг 4.1) перестраивает все индексы базы данных, используя команду DBCC DBREINDEX.

Листинг 4.1. Программа перестройки всех индексов базы данных

```
-- Задается имя базы данных, индексы которой перестраиваются
USE DatabaseName
DECLARE @TableName varchar(255)
-- Объявляем курсор для выборки названий базовых таблиц
-- из информационной схемы базы данных
DECLARE TableCursor CURSOR FOR
SELECT table_name FROM information_schema.tables
WHERE table_type = 'base table'
    Открываем курсор и выбираем из него первую строку
    OPEN TableCursor
    FETCH NEXT FROM TableCursor INTO @TableName
    -- Заголовок цикла перебора записей курсора
    WHILE @@FETCH_STATUS = 0
    BEGIN
    PRINT "Reindexing " + @TableName
    Перестройка индексов текущей таблицы с фактором заполнения, равным 90
    DBCC DBREINDEX(@TableName, '', 90)
    FETCH NEXT FROM TableCursor INTO @TableName
    END
    -- Закрытие и освобождение курсора
    CLOSE TableCursor
    DEALLOCATE TableCursor
```

Обратим внимание на команду DBCC DBREINDEX в этом листинге. У нее три параметра, первый из которых — это имя таблицы, а второй — пустая строка, что говорит о необходимости перестроить все индексы, а не индекс с конкретным именем. Значение фактора заполнения индекса, равное 90 %, является наиболее распространенным, если нужно, его можно изменять.

Следует иметь в виду, что при перестройке некластерного индекса на таблицу накладывается разделяемая блокировка, при которой допустимо только выполнение оператора SELECT, а при перестройке кластерного индекса на таблицу накладывается исключительная блокировка, которая предотвращает любой доступ пользователей к таблице. Из-за этого команду DBCC DBREINDEX нужно выполнять в то время, когда пользователи не запрашивают доступ к реорганизуемым таблицам.

SQL-сервер 2000 имеет команду DBCC INDEXDEFRAG, которая аналогично команде DBCC DBREINDEX выполняет дефрагментацию как кластерных, так и некластерных индексов для таблиц и индексных представлений. Использование DBCC INDEXDEFRAG вместо DBCC DBREINDEX часто выгодно, потому что эта команда не накладывает блокировки на продолжительное время в отличие от DBCC DBREINDEX. Но отрицательной стороной использования DBCC INDEXDEFRAG является ее более длинное выполнение и то, что статистика автоматически не модифицируется.

Чтобы определить уровень фрагментации индексов, возникшей из-за разбиения страниц, и, следовательно, узнать, насколько насущно выполнение команды DBCC DBREINDEX, можно выполнить команду DBCC SHOWCONTIG со следующим синтаксисом (в сокращенном виде):

```
DBCC SHOWCONTIG ( { table_name | table_Id | view_name | view_Id }
 [, index_name | index_Id ])
```

Листинг 4.2 является примером вызова команды DBCC SHOWCONTIG в ситуации, когда известны только имена таблицы и индекса, но не их системные идентификаторы.

Листинг 4.2. Определение уровня фрагментации заданного индекса

```
DECLARE @ID int, @IndexID int, @IndexName varchar(128)
-- Задаем имя индекса и таблицы
SELECT @IndexName = 'index_name' -- имя индекса
SET @ID = OBJECT_ID('table_name') -- системный идентификатор таблицы
-- Получаем значение системного идентификатора индекса
SELECT @IndexID = Indid FROM sysindexes
WHERE Id = @ID AND name = @IndexName
-- Просмотр степени фрагментации
DBCC SHOWCONTIG (@Id, @IndexID)
```

На примере таблицы ПРИХОДЫ базы данных МАТ_УЧЕТ эта программа выдала следующий результат:

```
DBCC SHOWCONTIG scanning ' ПРИХОДЫ ' table...
Table: ' ПРИХОДЫ ' (706101556); index ID: 0, database ID: 6
TABLE level scan performed.
- Pages Scanned.............................: 1277
- Extents Scanned...........................: 162
- Extent Switches...........................: 161
- Avg. Pages per Extent.....................: 7.9
- Scan Density [Best Count:Actual Count].....: 98.77% [160:162]
```

```
- Extent Scan Fragmentation ...................: 3.09%
- Avg. Bytes Free per Page.....................: 1398.4
- Avg. Page Density (full).....................: 82.72%
DBCC execution completed. If DBCC printed error messages, contact your
system administrator.
```

Команда `DBCC SHOWCONTIG` выводит много различных параметров, но основным из них является параметр `Scan Density`. Его значение должно быть сколь возможно близко к 100 %. Значение `Scan Density`, меньшее 75 %, является сигналом того, что индексы таблицы должны быть перестроены. Вероятно, при этом полезно также увеличить значение `FILLFACTOR`.

Что еще можно узнать из значений параметров команды `DBCC SHOWCONTIG`? Приведем смысл некоторых из них:

- `Pages Scanned` — указывает на то, сколько страниц данных выделено индексу;
- `Extents Scanned` — указывает на то, сколько экстентов выделено индексу;
- `Extent Switches` — указывает на то, сколько раз при сканировании индекса происходило переключение между экстентами. Хорошо, если этих переключений примерно столько же, сколько и экстентов, т. к. от отношения количества экстентов к количеству переключений между ними напрямую зависит значение `Scan Density`;
- `Scan Density` — плотность сканирования, об этом параметре рассказано ранее;
- `Avg. Bytes Free per Page` — среднее количество незанятых байтов на страницу индекса;
- `Avg. Page Density` — средняя плотность заполнения страницы. При этом `Avg. Page Density = (8192 - Avg. Bytes Free per Page) / 8192`.

Значение `Avg. Bytes Free per Page` может измениться после вставки значительного количества строк в таблицу. Пусть, например, создан индекс с `FILLFACTOR` = 0. Параметр `Avg. Page Density` после этого будет равен приблизительно 100 %. Затем в таблицу было вставлено значительное количество записей так, что после вставки размер таблицы примерно удвоился в терминах восьмикилобайтовых страниц. При вставке записей страницы индекса подвергались активному расщеплению, т. к. свободного места в них практически нет. Если после этой вставки записей снова выполнить команду `DBCC SHOWCONTIG`, то окажется, что значение `Avg. Page Density` значительно уменьшится, также уменьшится и значение `Scan Density`.

Листинг 4.3 использует средства динамического SQL для построения последовательности команд `DBCC SHOWCONTIG` для всех индексов базы данных.

Листинг 4.3. Определение уровня фрагментации всех индексов базы данных

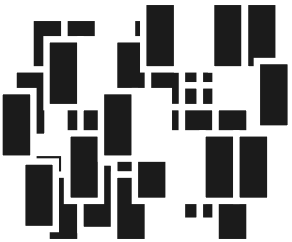
```
SELECT 'DBCC SHOWCONTIG (' +
CONVERT(varchar(20), i.Id) + ', ' +
CONVERT(varchar(20), i.Indid) + ') -- ' +
object_name(i.Id) + '.' + i.name
from sysobjects o inner join sysindexes I on (o.Id = i.Id)
 where o.type = 'U' and i.Indid < 2
and i.Id = object_Id(o.name)
ORDER BY object_name(i.Id), i.Indid
```

Результаты работы этой программы для базы данных `МАТ_УЧЕТ` частично приведены далее:

```
DBCC SHOWCONTIG (338100245, 1) -- ИСТОРИЯ_ЦЕН.PK_ЦЕНЫ_ПЕРЕМ
DBCC SHOWCONTIG (274100017, 1) -- ПРИХОДЫ.PK_ПРИХОДЫ
DBCC SHOWCONTIG (370100359, 1) -- РАСХОДЫ.PK_РАСХОДЫ
DBCC SHOWCONTIG (354100302, 1) -- РАСХОДЫ_ТОВАРЫ.PK_РАСХОДЫ_ТОВАРЫ
DBCC SHOWCONTIG (546100986, 1) -- ТОВАРЫ.PK_ТОВАРЫ
```

Выполнив код последнего листинга с помощью SQL Query Analyzer, результаты можно занести в буфер обмена и сохранить в текстовом виде для того, чтобы в дальнейшем выполнять команду `DBCC SHOWCONTIG` по мере необходимости.

ГЛАВА 5



Программирование на SQL-сервере

Данная глава посвящена рассмотрению вопросов практического программирования для SQL-сервера на языке T-SQL. При этом непосредственное изложение синтаксиса операторов языка не является основной задачей. Преследуется цель дать навыки решения типичных задач на примере простейшего проектного решения для предметной области задачи материального учета. Кроме того, подробно разбираются методы оценки и сравнения эффективности того или иного способа решения для большинства задач, даются общие рекомендации по стилю программирования и демонстрируется применение специальных средств SQL-сервера, позволяющих упростить или автоматизировать поиск оптимального способа решения задачи.

База данных с именем `МАТ_УЧЕТ`, которая выбрана для демонстрации примеров из данной главы, является физической реализацией простейшего проектного решения для OLTP бизнес-среды. В предыдущих главах разбиралось большое количество примеров получения отдельных частей проектных решений для нескольких предметных областей и, в частности, для задачи материального учета. Далее описана именно та бизнес-среда, для которой создана база данных `МАТ_УЧЕТ`.

Рассмотрим гипотетическое предприятие, которое занимается оптовыми закупками или производством каких-либо товаров с последующим сбытом продукции в розницу. Подобная деятельность порождает одно из самых распространенных проектных решений для реляционных баз данных, связанное с учетом материальных ценностей. Примером может служить широко известная учебная база `NORTHWIND` от Microsoft.

В нашем случае в задаче материального учета выделяются две основные бизнес операции, соответствующие транзакциям базы данных: закупка (производство) партии товаров и их поставка (отпуск) контрагентам. Каждому факту выполнения любой из этих операций соответствует документ, идентифи-

цируемый уникальным кодом, который назовем *накладной*. На закупку (производство) товара оформляется приходная накладная, имеющая только одну товарную позицию. Оформление приходной накладной соответствует тому факту, что на склад нашего предприятия поступает партия однотипных (одинаково идентифицируемых) товаров в известном количестве. Отпуску товаров (изделий) со склада предприятия соответствует расходная накладная. В отличие от приходной накладной, расходная накладная может включать в себя несколько товарных позиций, для каждой из которых должен быть указан код товара и его количество в данной расходной накладной. В одну дату может быть оформлено несколько как приходных, так и расходных накладных с любой номенклатурой товаров.

Товар идентифицируется уникальным кодом и характеризуется такими неизбыточными атрибутами, как наименование и учетная цена. Учетная цена в нашей задаче имеет экономический смысл себестоимости товара. Отпуск товаров, обеспечивающий прибыльность работы предприятия, производится по простейшей схеме — любой товар продается с фиксированной наценкой, равной в нашем случае 10 % от учетной цены.

Для каждого кода товара в базе данных должен храниться избыточный атрибут, называемый *остатком товара*, который соответствует количеству данного товара на складе в текущий момент времени. Значение этого остатка должно быть согласовано со всеми оформленными приходными и расходными накладными *(см. бизнес-правила, сформулированные далее)*.

Назначение базы данных состоит, в первую очередь, в хранении данных для оперативного решения задач, связанных с оформлением документов о движении товаров (изделий) и об их запасах. Кроме того, должна быть обеспечена возможность простейшего анализа соотношения закупок и сбыта товаров по временным параметрам (периодам работы).

На рис. 5.1 показано основное проектное решение для описанной задачи, которое соответствует диаграмме с именем `МАТ_УЧЕТ_ОСН`.

Стандартные ограничения целостности данных в виде первичных ключей и внешних ключей видны на рис. 5.1. Эти ограничения физически реализованы с помощью индексов, удовлетворяющих общепринятым рекомендациям. Для первичных ключей автоматически созданы уникальные кластерные индексы. Для внешних ключей созданы некластерные индексы.

Замечание

Никаких других индексов в базе данных `МАТ_УЧЕТ` нет. Однако дальше будет разбираться большое количество примеров, связанных с созданием дополнительных индексов, необходимых для оптимизации работы запросов в нашей базе данных. Во всех этих примерах будет приведен оператор создания соответствующего индекса.

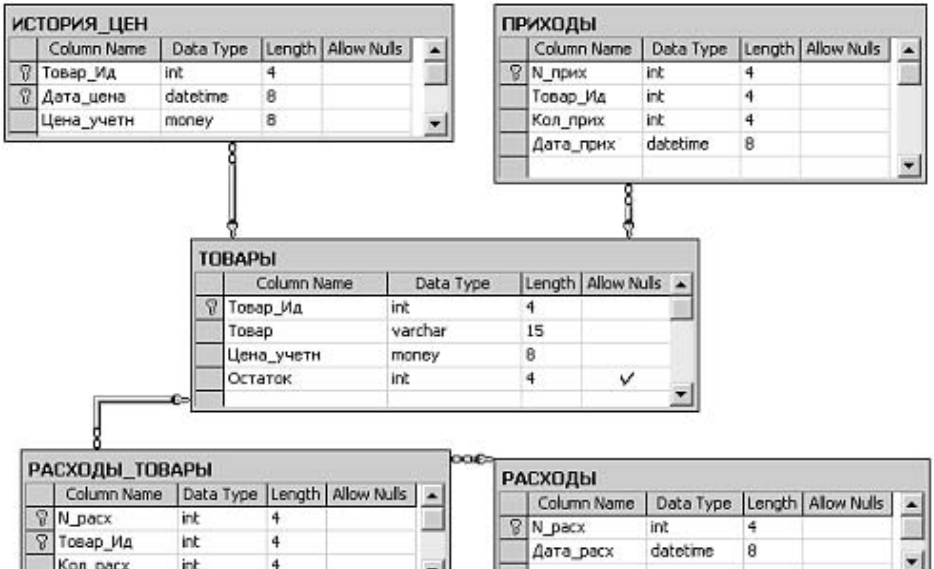


Рис. 5.1. Диаграмма для основного проектного решения задачи материального учета

База данных должна поддерживать ограничения целостности данных, соответствующие следующим бизнес-правилам:

1. Оформление расходной накладной может быть выполнено, только если на складе в достаточном количестве имеется каждый товар, входящий в эту расходную накладную.
2. Остаток каждого товара в любой момент времени должен соответствовать разнице между общей суммой закупок и общей суммой продаж для соответствующего товара.
3. Идентификаторы кодов товаров и накладных генерируются автоматически.

Эти ограничения целостности поддерживаются с помощью триггеров *(см. разд. 5.11)*.

Данные, содержащиеся в базе данных `МАТ_УЧЕТ`, получены путем автогенерации, полный сценарий которой не приводится. Однако в *разд. 5.8* приведены и подробно разобраны листинги программ, которые могут служить прототипом написания собственного сценария автогенерации данных с необходимыми свойствами.

5.1. Понятие об оптимизаторе запросов

5.1.1. Общая схема выполнения SQL-запроса

На текущий момент времени фактическим стандартом доступа к реляционным базам стал язык SQL. Все СУБД, претендующие на название "реляционные", реализуют тот или иной диалект SQL. Операторы языка SQL по своей сути представляют собой способ записи операторов реляционной алгебры. Важным моментом здесь является вывод о реляционной полноте стандартизированного SQL, который сводится к утверждению о том, что любой оператор реляционной алгебры может эквивалентным образом быть записан в синтаксисе SQL.

Чистый SQL не является алгоритмическим языком в обычном понимании этого определения, т. к. не содержит в себе управляющих структур, таких как условные операторы или циклические конструкции. Кроме того, в чистом SQL нет таких средств структурирования программы, как пользовательские процедуры и функции. Эти средства могут быть добавлены к SQL, и тогда возникают языки, известные как процедурные расширения SQL, применяемые для программирования на современных серверах баз данных. Язык программирования SQL-сервера под названием T-SQL относится как раз к ним. Однако следует понимать, что ядром любого языка реляционных баз данных являются именно реляционные операции, от эффективности выполнения которых, в первую очередь, зависит эффективность работы сервера в целом. Основным оператором SQL, который реализует обработку таблиц, т. е. собственно выполняет операции реляционной алгебры, является оператор `SELECT`, называемый также оператором выборки. Иногда оператор `SELECT` называют просто запросом. Часто этот оператор используется не сам по себе, а является основой других операторов как языка SQL, так и языка T-SQL. Неудивительно поэтому, что основное внимание уделяется прежде всего оптимизации работы оператора выборки, для чего разработаны специальные средства.

Процесс обработки запроса на языке высокого уровня (обычно SQL) состоит из следующих этапов:

1. Декомпозиция запроса или его синтаксический разбор.
2. Дополнительная проверка на предмет действительности имен системных каталогов, базовых таблиц и названий полей, используемых в операторе, а также соответствующих привилегий пользователя.
3. Генерация и оптимизация плана доступа к ресурсам *(плана выполнения)*, т. е. двоичного представления выполнимого кода по отношению к данным.
4. Генерация процессорного кода для признанного оптимальным плана доступа.
5. Фактическое выполнение запроса.

Декомпозиция запроса состоит в преобразовании запроса, сформулированного на языке высокого уровня, в нормализованное выражение реляционной алгебры с последующей проверкой его синтаксической и семантической корректности.

Основной оператор SQL — это оператор `SELECT`, представляющий собой формализованную запись некоторого выражения реляционной алгебры, которое нужно применить к заданным в нем таблицам. Следует помнить, что инструкции запроса на выборку не выполняются в том порядке, в котором они перечислены в операторе `SELECT`. В силу коммутативности операторов реляционной алгебры оптимизатором генерируется такой план запроса, который выполняется наиболее эффективно. При этом обычной практикой является выполнение операции выборки, задаваемой опцией `WHERE`, на самых ранних стадиях общей обработки запроса перед операциями соединения, указанными в опции `FROM`.

Если опция `WHERE` не используется, то SQL-сервер возвращает все строки, что негативно влияет на производительность по нескольким причинам. Прежде всего, обрабатывая ненужные данные, приходится исполнять очень ресурсоемкую операцию чтения с диска, тратя впустую ресурсы сервера. Кроме того, если запрос написан так, что лишние ненужные строки появляются в финале запроса и передаются клиентскому приложению, то тем самым увеличивается сетевой трафик, что является одной из основных предпосылок, ведущих к уменьшенной общей производительности системы. Другой отрицательный аспект полного сканирования таблицы состоит в том, что в буферный кэш будут заноситься страницы с бесполезными данными, снижая эффективность процесса кэширования данных. Кроме того, полное сканирование таблицы повышает вероятность конфликтов блокировок, снижая параллелизм.

Поэтому следует везде, где только возможно, использовать *горизонтальную фильтрацию* данных, суть которой состоит в том, что запросы нужно писать таким образом, чтобы избегать считывания и обработки лишних строк на самых ранних шагах выполнения запроса. Практически это достигается путем включения опции `WHERE` в операторы `SELECT` везде, где это возможно, чтобы уменьшить число возвращенных строк еще до выполнения операций соединения таблиц.

Для выполнения горизонтальной фильтрации по количеству возвращаемых записей можно воспользоваться одной из двух альтернатив: либо командой `SET ROWCOUNT`, либо опцией `TOP` оператора `SELECT`. В большинстве случаев эти способы одинаково эффективны, но имеются некоторые ситуации (например возврат строк из неотсортированной кучи), когда опция `TOP` более производительна, поэтому в целом ее использование более предпочтительно.

Кроме горизонтальной фильтрации полезно везде, где только можно, использовать *вертикальную фильтрацию*, которая означает, что не следует в запро-

сах возвращать данные из столбцов, в которых нет нужды. Прежде чем написать `SELECT *` для возврата всех столбцов таблицы, нужно хорошо подумать, а действительно ли все столбцы необходимы приложению? Передача данных из ненужных столбцов, кроме очевидного недостатка, связанного с ростом сетевого трафика, еще и сводит на нет такой мощный способ повышения эффективности выполнения запроса, как использование *перекрывающихся индексов.*

Декомпозированный запрос должен быть выполнен наиболее оптимальным способом. Критерий оптимальности в данной постановке задачи является многоцелевым и крайне сложным, т. к. в нем должны учитываться многогранные аспекты понятия интегрированной производительности сервера. Но как общий вывод можно принять утверждение о том, что отдельный запрос считается оптимизированным, если он выполняется наиболее быстрым способом. Вопросами оптимизации запросов занимается специальный блок ядра СУБД, называемый *оптимизатором запросов*.

При рассмотрении проблемы оптимизации запросов различают *эвристическую* оптимизацию и *стоимостную* оптимизацию. Эвристическая оптимизация основана на том факте, что сложные выражения реляционной алгебры можно выполнять разными способами. Эти способы дают эквивалентный результат, но требуют для своего выполнения различных ресурсов. Владея информацией о правилах преобразований операторов реляционной алгебры, оптимизатор может построить различные планы выполнения для одного и того же запроса. Под *планом выполнения запроса* понимается древовидная структура, отображающая последовательность выполнения элементарных операций реляционной алгебры, реализующих запрос. Кроме того, сами элементарные операции, из которых составлен запрос, могут выполняться разными способами в зависимости от наличия или отсутствия индексов на поля таблиц, участвующих в этих операциях.

Стоимостная оптимизация запроса заключается в подсчете некоторого числового параметра, называемого *стоимостью запроса*, для каждого из возможных планов его выполнения. Смысл введения понятия стоимости заключается в попытке формализации до уровня числового выражения критерия производительности запроса, подлежащего оптимизации. Часто в основу вычисления стоимости запроса закладывают количество операций чтения данных с диска, т. к. эта операция выполняется относительно медленно. Выражение для определения стоимости запроса опирается на статистические показатели таблиц и индексов, участвующих в запросе. В первую очередь, к таким показателям относится число страниц диска, занимаемое таблицей или индексом. Кроме того, значение стоимости запроса в каждом конкретном случае зависит от аппаратных параметров и конфигурации того компьютера (сервера), на котором выполняется запрос. Статистическая информация, на основании которой

выполняется оценка стоимости запроса, по умолчанию создается сервером автоматически и автоматически же поддерживается в адекватном состоянии незаметно для пользователя. Пользователь, впрочем, имеет некоторые возможности управления этой статистикой, подробнее о которых будет рассказано далее *(см. разд. 5.2)*.

Стоимостная оптимизация запроса оперирует *оценочной стоимостью* (estimated cost). Это и понятно — если производить поиск оптимального плана выполнения на основании реальной стоимости, то нужно несколько раз выполнять запрос по разным планам для реальных данных. Ясно, что для сколько-нибудь значительных объемов данных такая оптимизация оказывается бессмысленной — первый же план выполнения, пусть и не оптимальный, уже дает тот ответ (результаты запроса), ради получения которого оптимизация проводилась. Идея же оптимизации состоит в том, чтобы на больших объемах данных реально выполнять запрос сразу самым эффективным образом. Поэтому из больших таблиц производятся некоторые выборки данных существенно меньшего объема, которые и представляют собой статистическую информацию, лежащую в основе решений оптимизатора запросов. Оптимизатор имитирует выполнение запроса по разным планам для этих выборок, подсчитывая, таким образом, оценочные стоимости и сравнивая их между собой.

В реальной практике оптимизатор SQL-сервера применяет комбинированную методику, сочетающую как эвристическую, так и стоимостную оптимизацию. В конце концов, для множества сгенерированных с помощью эвристических методик планов запроса подсчитывается оценочная стоимость их выполнения с учетом наличных индексов, после чего выбирается в качестве оптимального тот план выполнения, который имеет минимальную стоимость. В тех случаях, когда не используются подсказки оптимизатору, позволяющие разработчику изменять план выполнения запроса в соответствии со своими представлениями, запрос будет выполнен по тому плану, у которого оценочная стоимость минимальна.

Далее на основании оптимального плана запроса генерируется процессорный код и происходит собственно выполнение запроса с возвратом его результатов.

В данной главе не ставится задача рассмотрения конкретных методов декомпозиции запросов как операторов реляционной алгебры или разбор методик подсчета стоимости запросов. Конечно, понимание математических основ является немаловажным, поэтому приведем ссылку на источник, который в достаточно полном объеме раскрывает математическую суть дела [13]. На практическом уровне обычно достаточно выводов из теории, которые оформлены в конкретные рекомендации, подкрепленные примерами и снабженные объяснениями, почему же данная рекомендация возникла.

5.1.2. Графическое представление плана запроса SQL-сервером

При изложении материала данной главы достаточно актуальна проблема нотации результатов. Речь идет вот о чем. Необходимо как-то изображать планы запросов, а для этого нужно рисовать деревья, в узлах которых находятся символы, обозначающие *элементарные операции* реляционной алгебры. Для решения этой задачи будем ориентироваться на те средства, которые предлагает SQL-сервер, вместо того чтобы пользоваться чисто математическими способами обозначения. Поэтому *дерево запроса*, называемое обычно *планом выполнения запроса*, будет изображаться в той нотации, которую предлагает графическая утилита SQL Query Analyzer (анализатор SQL-запросов). На рис. 5.2 представлено такое дерево запроса вместе с его текстом. Пока не имеет значения, что этот запрос делает и для каких данных. О том, как выводить план выполнения запроса на экран, будет рассказано далее.

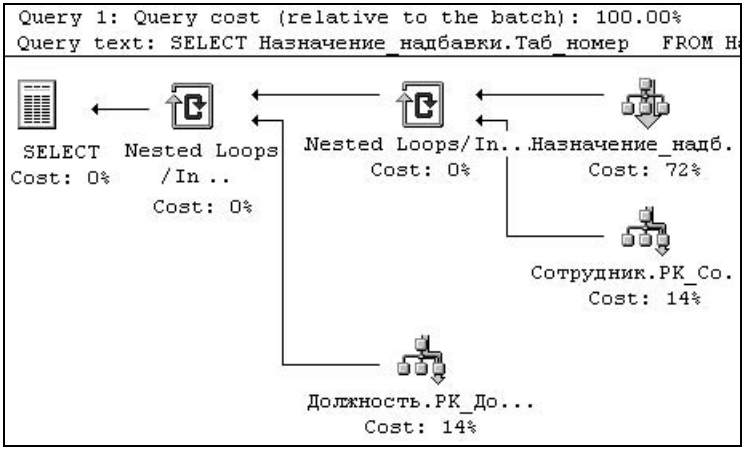


Рис. 5.2. Пример дерева запроса в нотации утилиты SQL Query Analyzer

Графический план выполнения запроса в нотации утилиты SQL Query Analyzer читается справа налево и сверху вниз. Для каждого узла дерева за исключением корневого (самого левого) имеется родительский узел (предок). При этом все узлы дерева, имеющие одинакового предка, изображаются в одном столбце (друг под другом). То есть план запроса выводится так, как выглядело бы дерево, растущее от корня, поваленное вправо на бок с задранным вверх корнем. Узлы дерева соединены стрелками, идущими к родительским узлам, и чем стрелка более широкая, тем больше записей возвращается в результате выполнения элементарной операции, от которой эта стрелка отходит.

Каждая элементарная операция реляционной алгебры, находящаяся в узле дерева запроса, изображается соответствующей пиктограммой. В табл. 5.1 приведены изображения наиболее часто встречающихся элементарных операций. Их полный перечень можно просмотреть в BOL (поиск "Graphically Displaying the Execution Plan").

Таблица 5.1. *Перечень наиболее часто встречающихся операций плана запроса*

Пикто-грамма	Физический оператор	Пикто-грамма	Физический оператор
	Поиск по закладке (`row ID`) (Bookmark Lookup)		Соединение перемешиванием (хешированием) (Hash Match)
	Сканирование кластерного индекса (Clustered Index Scan)		Соединение хешированием для корневых значений (Hash Match Root)
	Поиск по кластерному индексу (Clustered Index Seek)		Соединение хешированием для набора (Hash Match Team)
	Фильтрация (Filter)		Спулинг индекса (Index Spool)
	Сканирование некластерного индекса (Index Scan)		Соединение сортировкой-слиянием (Merge Join)
	Поиск по индексу (Index Seek)		Соединение вложенными циклами (Nested Loops)
	Выборка (SELECT)		Сортировка (Sort)
	Агрегация потоком (Stream Aggregate)		Сканирование таблицы (Table Scan)
	Спулинг таблицы (Table Spool)		Отбор верхних строк (Top)

При наведении указателя мыши на любую из пиктограмм на экран выводится окно сведений, содержащее подробную информацию о том, как выполняется операция для этого узла дерева запроса. При наведении мыши на стрелку выводятся сведения о размере и количестве строк, возвращаемых в результате выполнения операции в том узле, от которого идет стрелка. Для совпадающих по внешнему виду пиктограмм табл. 5.1, например для индексных операций, нужно уточнять тип индекса, используя окно сведений элементарной операции.

В SQL-сервере существуют два плана запроса — *план выполненного запроса* (execution plan) и *оценочный план выполнения* (estimated execution plan). Первый из них может быть получен, если реально выполнить запрос с помощью утилиты SQL Query Analyzer при включенной опции меню **Query/Show Execution Plan**. Тогда в окне результатов появится дополнительная вкладка, с помощью которой и можно просматривать план выполненного запроса. Оценочный план выполнения выводится с помощью опции меню **Query/Display Estimation Execution Plan**. Для вывода на экран этого плана не нужно выполнять сам запрос. Параметры оценочного плана запроса рассчитываются на основании статистики базы данных и не всегда совпадают с соответствующими параметрами плана выполненного запроса.

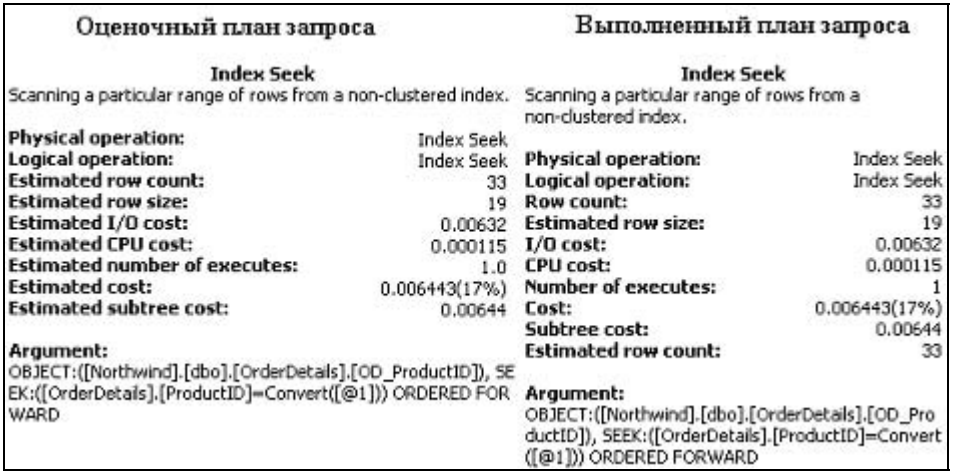
В табл. 5.2 приведена информация о параметрах операций в узлах дерева запроса как для плана выполненного запроса, так и для оценочного плана запроса (во втором случае перед названием соответствующего параметра добавляется слово "Estimated").

Таблица 5.2. *Параметры операций в планах запросов*

Параметр	Смысл параметра	Примечание
`Physical operation`	Физический оператор	Алгоритм, с помощью которого выполняется операция в узле дерева
`Logical operation`	Логический оператор	Название операции реляционной алгебры для узла дерева
`Row count`	Число записей, полученное в результате выполнения операции в узле	Выводится только для плана выполненного запроса, а не для оценочного плана выполнения. Совпадает со значением для стрелки, выходящей из узла
`Estimated row size`	Оценочное значение длины записи	Совпадает со значением для стрелки, выходящей из узла
`I/O cost` `Estimated I/O cost`	Стоимость выполнения ввода/вывода	Операция ввода/вывода выполняется для каждой страницы дисковой памяти таблицы или индекса

Таблица 5.2 *(окончание)*

Параметр	Смысл параметра	Примечание
`CPU cost` `Estimated CPU cost`	Стоимость использования процессора	
`Number of executes` `Estimated Number of executes`	Количество выполнений операции при работе запроса	
`Cost` `Estimated Cost`	Общая стоимость операции	`Cost` ≈ (`I/O cost`) + (`CPU cost`)
`Subtree cost` `Estimated Subtree cost`	Стоимость выполнения операций поддерева	Суммарная стоимость выполнения операций во всех предшествующих узлах
`Estimated row count`	Оценочное число записей, полученное при выполнении операции	Если статистика базы данных адекватна, то это значение должно приблизительно совпадать с `Row count`
`Argument`	Предикаты и параметры, использованные в запросе	



Оценочный план запроса

Index Seek
Scanning a particular range of rows from a non-clustered index.

Physical operation:	Index Seek
Logical operation:	Index Seek
Estimated row count:	33
Estimated row size:	19
Estimated I/O cost:	0.00632
Estimated CPU cost:	0.000115
Estimated number of executes:	1.0
Estimated cost:	0.006443(17%)
Estimated subtree cost:	0.00644

Argument:
OBJECT:([Northwind].[dbo].[OrderDetails].[OD_ProductID]), SEEK:([OrderDetails].[ProductID]=Convert([@1])) ORDERED FORWARD

Выполненный план запроса

Index Seek
Scanning a particular range of rows from a non-clustered index.

Physical operation:	Index Seek
Logical operation:	Index Seek
Row count:	33
Estimated row size:	19
I/O cost:	0.00632
CPU cost:	0.000115
Number of executes:	1
Cost:	0.006443(17%)
Subtree cost:	0.00644
Estimated row count:	33

Argument:
OBJECT:([Northwind].[dbo].[OrderDetails].[OD_ProductID]), SEEK:([OrderDetails].[ProductID]=Convert([@1])) ORDERED FORWARD

Рис. 5.3. Сравнение параметров оценочного и выполненного планов запроса

Дадим некоторые пояснения к табл. 5.2. В первом ее столбце указано название параметров для двух типов планов выполнения запроса. На рис. 5.3 пока-

заны различия в отображении параметров одной и той же элементарной операции для выполненного и оценочного планов запроса на примере элементарной операции, соответствующей одному из узлов плана запроса к учебной базе данных NORTHWIND.

Из рис. 5.3 видно, что после параметра стоимость (**Cost**) в скобках выводится его относительное значение в процентном выражении, которое показывает, какой вклад в суммарную стоимость запроса вносит конкретная операция. Суммарную стоимость запроса можно просмотреть, если вывести параметры корневой пиктограммы графического плана выполнения. Тогда в строке **Estimated subtree cost** будет указано суммарное значение стоимости запроса в целом без стоимости самой корневой операции.

5.1.3. Оценка и измерение времени выполнения запроса

Зависимость между стоимостью выполнения запроса и временем его выполнения достаточно сложная и далеко не во всех случаях прямо пропорциональная. В общем считается, что запрос с меньшей стоимостью выполняется быстрее. Поэтому просмотр стоимостей отдельных операций и всего запроса в целом безусловно полезен, но нужно быть готовым к ситуации, что запрос с меньшей стоимостью будет выполняться дольше. Вразумительного ответа на этот парадокс Microsoft не дает, поэтому в реальной жизни необходимо уметь замерять и прогнозировать время выполнения запросов. Для замера реального времени выполнения запрос должен быть обязательно выполнен. При этом нужно иметь в виду следующее.

- Размеры таблиц должны быть по возможности приближены к реально ожидаемым, а данные в столбцах должны иметь адекватную *селективность*. Если предполагается при промышленной эксплуатации использовать сервер в режиме автоматического обновления статистики, то перед проведением теста нужно позаботиться о том, чтобы статистика тестовых данных была адекватна (обновлена). Если в реальной жизни статистика использоваться не будет, то ее нужно отключить и при тестировании.
- Выполнять запрос следует на самом сервере, чтобы на полученные результаты не влияло время, затрачиваемое на передачу данных по сети, которое обычно существенно больше, чем время реальной обработки и, следовательно, может исказить результат до полной непригодности.
- При проведении теста необходимо создать ситуацию, когда невозможно возникновение конфликта блокировок для данных, обрабатываемых в запросе. Идеальна с этой точки зрения ситуация, когда сервер работает в монопользовательском режиме.

- Все другие приложения при проведении теста по быстродействию должны быть закрыты.
- Следует понимать, что абсолютные временные характеристики, полученные при тестировании, очень сильно зависят от аппаратной конфигурации компьютера, выбранного в качестве сервера. Поэтому при переносе серверного программного обеспечения на другой компьютер придется снова выполнить необходимые тесты для критически важных запросов.
- Сравнение относительных временных характеристик, например для разных способов решения одной и той же задачи, в меньшей степени привязано к конфигурации конкретного компьютера. Поэтому можно с достаточной степенью уверенности предполагать, что найденное разработчиком оптимальное решение задачи останется наиболее производительным в любой ситуации.

Обычно время выполнения запроса оценивается в миллисекундах. На практике для его замера можно воспользоваться одной из перечисленных возможностей.

Во-первых, перед выполнением запроса и сразу после него можно вставить команду `SELECT GetDate()`, после чего вычесть из второго значения времени первое, например, так, как это сделано в листинге 5.1.

Листинг 5.1. Использование функции `GetDate()` для замера времени выполнения

```
-- Объявляем две переменные типа дата/время
DECLARE @t1 DATETIME, @t2 DATETIME
-- Заносим текущее время в первую переменную
SELECT @t1 = GetDate()
-- Запрос, время выполнения которого нужно замерить
SELECT * FROM OrderDetails
-- Заносим текущее время во вторую переменную
SELECT @t2 = GetDate()
-- Выводим разность текущего времени до и после выполнения запроса, в мс
SELECT DATEDIFF(ms, @t1, @t2) AS ВРЕМЯ_ВЫПОЛНЕНИЯ
```

Во-вторых, для определения продолжительности выполнения запроса в опциях настройки соединения можно включить параметр `SET STATISTICS TIME ON`, по умолчанию имеющий значение `OFF`. Это выполняется как с помощью приведенной команды, так и с помощью меню утилиты SQL Query Analyzer. В последнем случае нужно выбрать опцию **Tools**, затем опцию **Options...**, после чего на вкладке **Connection Properties** появившегося диалогового окна

установить соответствующий флажок. Тогда в окне результатов после собственно результатов запроса будут выведены два значения `SQL Server Execution Times`, показывающие время, затраченное сервером на выполнение запроса, а именно `CPU time` (время работы процессора) и `elapsed time` (общее затраченное время) в миллисекундах. Выглядит это примерно так, как показано далее.

```
SQL Server Execution Times:
  CPU time = 30 ms, elapsed time = 2061 ms.
```

Если запрос выполняется более секунды, то в нижней полосе окна утилиты SQL Query Analyzer после названия текущей базы данных можно увидеть динамически изменяющееся время, которое как раз и показывает, сколько выполнялся последний пакет команд.

Два вышеуказанных способа замера времени выполнения запроса выдают результаты, которые сильно зависят от окружающей среды проведения теста. Это связано с тем, что как время, выдаваемое функцией `GetDate()`, так и время, отсчитываемое на панели окна утилиты SQL Query Analyzer, — это просто время, которое само по себе никак не связано с работой тестируемого запроса. Поэтому оно может быть самым разным в зависимости от того, что еще происходило на компьютере при проведении теста. Значение параметра `elapsed time` обычно почти совпадает с разностью `DATEDIFF(ms, @t1, @t2)` из листинга 5.1, если при проведении тестирования придерживаться вышеприведенных рекомендаций.

И наконец, самые обширные возможности по измерению времени разнообразных событий предоставляет утилита SQL Profiler *(см. приложене 1)*. К столбцам данных следа этой утилиты, которые связаны с измерением времени, относятся: `StartTime` (время начала события), `EndTime` (время окончания события), `Duration` (продолжительность события, `Duration = DATEDIFF(ms, StartTime, EndTime)`), а также столбец с именем `CPU` (время обработки события центральным процессором). Пример простейшего следа трассировки, содержащего информацию о времени выполнения операторов T-SQL (`EventClass` = 40 соответствует событию `SQL:StmtStarting`, а `EventClass` = 41 соответствует событию `SQL:StmtEnding`), приведен в табл. 5.3.

Замечание

Таблица следа трассировки отформатирована вручную, в нее добавлены строки, содержащие комментарии, текст которых выделен курсивом. Текст операторов фонового процесса приведен в неполном объеме.

В утилиту SQL Query Analyzer включен простейший вариант шаблона следа трассировки, который содержит столбцы данных, наиболее нужных при от-

ладке запросов. Если для выполнения запроса нужно знать только параметры `Duration` и `CPU`, а именно они интересуют разработчика серверной части приложения, то удобнее всего воспользоваться следом трассировки прямо из SQL Query Analyzer. Для этого нужно включить опцию меню **Query/Show Server Trace**, после чего выполнить запрос и просмотреть информацию на соответствующей вкладке окна результатов.

Таблица 5.3. *Результаты использования утилиты SQL Profiler для измерения времени*

EventClass	TextData	SPID	Duration	StartTime	EndTime	CPU
*Послан на выполнение запрос "SELECT * from ПРИХОДЫ"*						
40	`SELECT * from ПРИХОДЫ`	54		2004-08-20 11:03:30.337		
41	`SELECT * from ПРИХОДЫ`	54	14191	2004-08-20 11:03:30.337	2004-08-20 11:03:44.527	50
Начал выполняться следующий оператор некоторого фонового процесса						
40	`SELECT` 504,...	55		2004-08-20 11:03:57.177		
41	`SELECT` 504,...	55	0	2004-08-20 11:03:57.177	2004-08-20 11:03:57.177	0

5.2. Управление статистикой

Как уже говорилось, сервер может сохранять определенную информацию о характеристиках таблиц и индексах базы данных, которая является необходимой оптимизатору для построения наиболее эффективного плана выполнения запросов. Основными статистическими показателями для таблицы являются:

- количество записей в таблице (кардинальность базового отношения — C);
- количество уникальных значений для каждого из использованных атрибутов (кардинальность выборки по каждому атрибуту — k_i, $i = 1, n$, где n — количество атрибутов отношения);
- количество страниц дисковой памяти, используемых для хранения таблицы;
- количество операторов `INSERT`, `UPDATE` и `DELETE` с момента последнего обновления статистики.

Кроме табличной статистики, используется также статистика индексов, включающая в себя:

- число уровней для каждого из многоуровневых индексов;
- количество страниц дисковой памяти, используемых для хранения индекса;
- гистограмму распределения данных в первом столбце индекса;
- среднюю длину ключевого значения в индексе.

Полный перечень статистических параметров для таблиц и индексов можно просмотреть с помощью команды `DBCC SHOW_STATISTICS`, например команда

```
DBCC SHOW_STATISTICS (ПРИХОДЫ, IX_ПРИХОДЫ_Товар_Ид
```

выводит отчет о статистических параметрах индекса `IX_ПРИХОДЫ_Товар_Ид` таблицы `ПРИХОДЫ`. Подробнее с параметрами этой команды можно ознакомиться в BOL (поиск "DBCC SHOW_STATISTICS").

По умолчанию сервер создает и поддерживает необходимую для его работы статистику автоматически, что соответствует параметрам базы данных `Auto Create Statistics` и `Auto Update Statistics` (иначе `AUTOSTATS`), установленным по умолчанию в значение истина. Еще эта особенность может быть изменена отключением опции **Auto update statistics** на вкладке **Option** (опции) окна **Properties** (свойства) базы данных в SQL Server Enterprise Manager (сокращенное название — Enterprise Manager).

Если база данных конфигурируется с параметром `Auto Create Statistics`, установленным в значение истина, то оптимизатор может принять решение о добавлении к статистике информации о столбце без индексов, для которого обычно по умолчанию статистика не строится. Как же узнать, что подобная статистика автоматически создана? Для ответа на этот вопрос нужно выполнить следующий запрос для системной таблицы `sysindexes`:

```
SELECT name FROM sysindexes WHERE (name LIKE '%_WA_Sys%')
```

Этот запрос возвращает все столбцы таблиц базы данных, которые имеют статистику, созданную автоматически оптимизатором запросов. В этом случае значение, которое находится в столбце `name` таблицы `sysindexes`, — это имя соответствующей статистики. Наличие автоматически построенной статистики для столбца без индекса является признаком того, что стоит рассмотреть полезность создания индекса для этого столбца.

Чтобы результаты оценки стоимости запроса были максимально полезными и адекватными, необходимо, чтобы статистика была актуальной. Напомним, как, используя графические планы запросов, убедиться в адекватности статистики. Статистика таблиц, участвующих в запросе, адекватна, если значения параметров оценочного и выполненного планов запроса совпадают. Специально для проведения такого сравнения в плане выполненного запроса при-

водится значение параметра `Estimated row count`, которое должно совпадать со значением `Row count`.

Табличную и индексную статистику SQL-сервер по умолчанию модифицирует автоматически. Однако эта опция — не панацея. Когда сервер интенсивно загружен, автомодификация статистики, которая является ресурсоемким процессом, может происходить в неподходящее время, замедляя общую производительность. Получается замкнутый круг — для оптимизации быстродействия выполнения запросов нужна адекватная статистика, а обновление статистики требует ресурсов, замедляя общую производительность системы. В этом случае, как и при решении большинства оптимизационных задач, вероятно, придется экспериментировать, чтобы увидеть, является ли включение автомодификации статистики эффективным для конкретной среды. Как правило, эта опция остается включенной для системы, ресурсы которой задействованы не на предельном уровне, — это, видимо, лучшее решение.

Если сделан вывод о том, что автоматическая модификация статистики является фактором, существенно замедляющим работу системы, то ее можно отключить, например, с помощью команды:

```
EXEC sp_dboption <database_name>, 'auto update statistics', 'False'
```

При отключенном автоматическом поддержании статистики имеется рекомендация административного плана, согласно которой целесообразно включить задание на обновление статистики в план сопровождения базы данных и выполнять его каждую ночь (или в другое время минимальной нагрузки на сервер), если позволяет производительность сервера. Для ручного обновления статистики можно использовать системную процедуру `sp_updatestats` или команду `UPDATE STATISTICS`.

Если имеется база данных, предназначенная только для чтения, которая используется, например, как хранилище данных, то можно любым способом установить опцию этой базы данных **read only** на значение истина, что отключает блокировки и существенно ускоряет выполнение запросов. Нужно только убедиться в том, что перед переводом базы данных в состояние "только для чтения" ее статистика была модифицирована вручную и полностью соответствует состоянию базы данных. Это особенно важно, поскольку автомодификация статистики обязательно отключается при переводе базы данных в режим "только для чтения". Перевод базы данных в этот режим также можно выполнить с помощью хранимой системной процедуры `sp_dboption`:

```
EXEC sp_dboption <database_name>, 'read only', 'TRUE'
```

В реальной жизни автоматическое обновление статистики чаще включено, поэтому нужно хотя бы в общих чертах понимать, как оно происходит. Как же SQL-сервер узнает, когда модифицировать статистику? Набор правил для определения момента модификации статистики индекса таков: если число

строк в таблице больше, чем 6, но меньше или равняется 500, то статистика автоматически модифицируется после выполнения 500 модификаций таблицы. Если число строк в таблице $k > 500$, то модификация статистики автоматически выполняется после изменения $500 + 0{,}2k$ строк в таблице.

Чтобы увидеть, какое количество модификаций было сделано в таблице с момента последнего обновления статистики, и оценить, когда произойдет ее автоматическая модификация, можно обратиться к таблице `sysindexes` базы данных и просмотреть значение столбца `rowmodctr` для интересующего индекса. Дату и время последнего обновления статистики для заданного индекса из заданной таблицы можно посмотреть с помощью функции

```
STATS_DATE(<table_id>, <index_id>).
```

Аргументами этой функции являются системные идентификаторы таблицы и индекса, поэтому обращение к ней удобнее выполнять с использованием системных таблиц (пример взят из BOL):

```
USE master
GO
SELECT 'Index Name' = i.name,
'Statistics Date' = STATS_DATE(i.id, i.indid)
FROM sysobjects o, sysindexes i
WHERE o.name = 'employee' AND o.id = i.id
GO
```

Другой хороший способ выяснения того, как часто происходит модификация статистики и как она влияет на общую производительность, — это использование утилиты SQL Profiler *(см. приложение 1)*. Для этого нужно открыть объект события соответствующей вкладки, а затем выбрать событие `Auto-Update Stats`. Можно также выбирать любые другие события и столбцы данных. Рекомендуется сортировать результаты по `Event Class` так, чтобы было проще идентифицировать и рассмотреть события `Auto-Update Stats`. Далее необходимо запустить созданный след на выполнение в течение дня и посмотреть, сколько происходит событий автомодификации статистики. Если окажется, что события автомодификации статистики происходят в критические по нагрузке периоды или слишком часто, или же занимают слишком продолжительное время (это время указано в столбце `Duration` в окне следа трассировки SQL Profiler), то можно принимать обоснованное решение об отключении автомодификации статистики.

Еще один способ определять, как часто выполняется `AUTOSTATS`, состоит в запуске SQL-сервера с флагом трассировки 8721. В этом случае происходит запись данных в журнал событий SQL-сервера (файлы с именем errorlog в каталоге Program Files\Microsoft SQL Server\Mssql\Log) каждый раз, когда

выполняется AUTOSTATS. Стоит убедиться в том, что данный флажок снят после того, как работа с ним завершена.

Кроме представления о частоте обновления статистики нужно хотя бы в общих чертах понимать, как она строится. По умолчанию, когда таблица имеет размер меньше 8 Мбайт, тогда статистическая выборка совпадает с полной таблицей, и невозможно создать статистику, лучшую, чем уже есть. Поэтому, если модификация статистики не решает проблему повышения быстродействия для маленькой таблицы, остается только использовать подсказки оптимизатору.

Но если таблица занимает больше 8 Мбайт, то SQL-сервер автоматически создает статистику для таблицы, используя выборку доступных строк. Эта выборка может оказаться недостаточно хорошей для точных оценок. Существует способ получить лучшую статистику используя команду UPDATE STATISTIC, упрощенный синтаксис которой приведен далее:

```
UPDATE STATISTICS table | view [index | ( statistics_name [,...n ]) ]
[ WITH [[ FULLSCAN ] | SAMPLE number { PERCENT | ROWS } ] | RESAMPLE ]
 [ [, ] [ ALL | COLUMNS | INDEX ]
[ [, ] NORECOMPUTE ] ]
```

Эта команда имеет два параметра, которые позволяют в некоторой мере управлять выборкой строк в таблице для получения статистики. Так, параметр FULLSCAN вынудит SQL-сервер использовать каждую строку в таблице для построения статистики столбца. Очевидно, что время на обновление такой статистики может быть неприемлемым, особенно если таблица огромна. Как альтернативу опции FULLSCAN можно использовать опцию SAMPLE, что позволяет явно задать, сколько строк использовать для построения статистики. Как обычно, компромисс между точностью статистики и ресурсами на ее поддержание в каждом конкретном случае приходится достигать путем экспериментов.

Как уже говорилось, практический способ убедиться в том, что статистика устарела или имеет недостаточную точность, состоит в сравнении графических планов реально выполненного запроса с оценочным планом его выполнения. При этом значение параметра Estimated row count должно приблизительно совпадать со значением Row count. Если эти значения сильно различаются, то это может означать три вещи: статистика отсутствует, она устарела или же недостаточно точна. Сначала нужно проверить, отсутствует ли статистика. Если это так, то она легко может быть добавлена с помощью команды CREATE STATISTICS. Если статистика существует, то ее нужно модифицировать, и, скорее всего, проблема будет решена. Если же первые два предложения неверны, то статистика вероятно недостаточно точна, и нужно увеличить выборку для ее построения, используя опцию SAMPLE.

5.3. Общие рекомендации по использованию индексов SQL-сервером

Как уже говорилось в *главе 4*, основным средством повышения скорости выполнения конкретных запросов на выборку данных, реализуемых с помощью оператора `SELECT`, является создание необходимых индексов.

Индексы не могут быть созданы в вакууме. Другими словами, прежде чем создавать оптимальные индексы для таблиц, необходимо понять, какие запросы и в какие моменты времени будут выполняться в базе данных, что является непростой задачей, особенно если предприняты попытки добавить индексы к новой базе данных.

Оптимизируя индексы для базы данных, нужно в первую очередь идентифицировать проблемные запросы, т. е. те запросы, которые выполняются наиболее часто, а также запросы, которые, являясь наиболее ресурсоемкими, выполняются в моменты особенно высокой общей нагрузки на сервер. Но как идентифицировать эти запросы? Если база данных уже находится в режиме эксплуатации, то один из способов состоит в использовании утилиты SQL Profiler для типичной рабочей нагрузки с сортировкой запросов по их продолжительности или по частоте выполнения.

Если индексы создаются для новой базы данных, то проблемные запросы придется выявлять на основании понимания общей теории оптимизации, и результат во многом зависит от знаний и квалификации разработчика.

Следует отчетливо понимать, что все теоретические советы относительно стратегии индексации — это только общие руководящие принципы. Чтобы убедиться, что база данных оптимизирована должным образом, любую из выбранных стратегий индексации нужно проверить для конкретной ситуации.

Как только проблемные запросы идентифицированы, возникает задача найти для них наилучшие индексы, искусству решения которой и посвящено дальнейшее изложение.

5.3.1. Использование кластерных индексов

Напомним, что кластерный индекс соответствует физическому порядку сортировки данных в таблице. Отсюда следует, что таблица может иметь только один кластерный индекс. Обычно этот единственный кластерный индекс используют для реализации ограничения первичного ключа, следовательно, в этом случае кластерный индекс обладает ограничением уникальности. Но последнее условие не является обязательным; если это необходимо, то вполне можно создать кластерный индекс без указания опции `UNIQUE`. Однако не-

уникальный кластерный индекс автоматически включает в себя дополнительную информацию, размером 4 байта, на каждую индексную запись, чтобы гарантировать их уникальность на скрытом от пользователя уровне, что фактически ведет к увеличению размера кластерного индекса.

Следует принять за правило, что каждая таблица должна иметь кластерный индекс. Объяснение этому правилу связано с особенностью SQL-сервера считывать данные. Всякий раз, когда считываются данные, содержащиеся в кластерном индексе, SQL-сервер имеет возможность за одну элементарную операцию чтения считать целый экстент (8 последовательных страниц или 64 Кбайта), а не одну страницу, как это происходит при чтении страниц данных других типов. Некластерные индексы и данные из таблиц считываются постранично (8 Кбайт за одну элементарную операцию чтения), поэтому преимущество от наличия кластерных индексов очевидно. Кроме того, наличие кластерного индекса у таблицы гарантирует, что страницы данных этой таблицы будут дефрагментированы при выполнении операции перестраивания индексов.

Кластерный индекс нужно создавать прежде, чем создаются любые некластерные индексы. Если этот порядок нарушен, то при создании кластерного индекса все некластерные индексы будут перестроены автоматически, а это весьма длительный процесс.

Если таблица имеет и кластерный индекс, и некластерные индексы, то предпочтительно, чтобы кластерный индекс был основан на единственном столбце минимальной длины. Часто этому условию удовлетворяет атомарный автоинкрементный первичный ключ таблицы. В этом случае имеется еще одно преимущество, а именно устраняется расщепление страниц данных, т. к. если в таблицу с автоинкрементным кластерным индексом вставляются новые строки, то эти вставки физически произойдут последовательно одна за другой. При этих условиях разбиение страниц происходить не будет.

Для построения кластерного индекса плохо подходят те столбцы, которые часто модифицируются, потому что при любом изменении кластерного индекса для поддержания заданного порядка сортировки физически перестраиваются не только сами данные, но и все некластерные индексы.

При выполнении запросов кластерные индексы наиболее полезны в следующих ситуациях:

- для запросов на выборку по большим диапазонам значений. Особенно хорошо в этом случае, когда удалось выбрать такой кластерный индекс, по которому операция поиска (опция `WHERE`) выполняется достаточно часто;
- для запросов на выборку, в которых результаты должны быть отсортированы на тех или иных стадиях обработки запроса. Подобные запросы

обычно включают либо предикаты BETWEEN, либо операции сравнения <, > или же опцию ORDER BY;

- для запросов, содержащих соединение таблиц (опцию JOIN) по столбцам кластерного индекса;
- для запросов с группировкой и функциями агрегирования, такими как MAX, MIN или COUNT, т. к. все эти операции выполняются через промежуточную операцию сортировки данных.

5.3.2. Использование некластерных индексов

Некластерные индексы, являясь основным средством повышения эффективности операции поиска (опция WHERE), максимально полезны в следующих ситуациях:

- когда запрос возвращает небольшое количество строк и когда индекс имеет хорошую селективность (более 95 %). Если селективность столбца менее 95 %, то наиболее вероятно, что оптимизатор не будет использовать для выполнения операции поиска некластерный индекс, основанный на таком столбце. Изменить значение порога селективности нельзя, но можно, используя подсказку, явно приказать оптимизатору использовать индекс *(см. разд. 5.6)*;
- когда опции WHERE и ORDER BY определены для одного и того же столбца. В этом случае некластерный индекс дает двойное преимущество: ускоряет поиск нужных записей, а также их сортировку, потому что возвращаемые данные уже отсортированы, хотя для больших объемов возвращаемых данных полезнее кластерные индексы;
- для выполнения соединения таблиц (хотя кластерные индексы лучше, но они могут быть использованы для других столбцов);
- когда столбец или столбцы, которые будут проиндексированы, очень широки. Хотя создание индексов на широкие столбцы является в принципе плохой идеей, но если нет выбора, то для этой ситуации предпочтительнее некластерный индекс.

Напомним, что селективностью столбца S называется отношение кардинальности выборки по этому столбцу (k) к кардинальности таблицы в целом (C), обычно выражаемое в процентах, т. е.

$$S = \frac{k}{C} 100\,\% \qquad (1)$$

Здесь под величиной k имеется в виду количество различных значений в столбце, а под величиной C — количество строк в таблице. Параметры k и C

всегда включаются в статистику таблицы, поэтому оптимизатор может оперировать ими при построении плана запроса.

Как же подсчитать на практике значение селективности того или иного столбца? Листинг 5.2 решает эту задачу на примере столбца `Товар_Ид` таблицы `ПРИХОДЫ`. Подсчет выполняется по формуле (1). Заменив в программе из листинга 5.2 имя таблицы и имя столбца, можно ее использовать для других таблиц.

Листинг 5.2. Подсчет селективности для столбца

```
DECLARE @total_unique float,@total_rows float, @selectivity_ratio int
SELECT @total_unique = 0,@total_rows = 0,
@selectivity_ratio = 0
--Ищем общее количество уникальных значений в столбце Товар_Ид
--таблицы ПРИХОДЫ
 SELECT @total_unique = (SELECT COUNT(DISTINCT Товар_Ид) FROM ПРИХОДЫ)
-- Ищем общее количество записей в таблице ПРИХОДЫ
SELECT @total_rows = (SELECT COUNT(*) FROM ПРИХОДЫ)
--Вычисляем селективность столбца
SELECT @selectivity_ratio =ROUND(@total_unique/@total_rows,2,2)*100
SELECT @selectivity_ratio as 'Селективность'
```

Поскольку селективность является важным статистическим параметром, представление об ее значении можно получить используя такое средство построения отчетов о статистической информации, как `DBCC SHOW_STATISTICS` *(см. BOL)*. Параметр `All Density` этой команды изменяется от нуля до единицы. Чем меньше его значение для проверяемого индекса, тем выше селективность этого индекса.

В некоторых случаях, даже когда столбец имеет подходящий некластерный индекс, оптимизатор не использует его для поиска, выполняя вместо этого сканирование таблицы или кластерного индекса, что, конечно, может снижать эффективность. Эта специфическая проблема возникает, когда имеет место так называемая корреляция данных между порядком строк в таблице и порядком входов в некластерном индексе. Например, кластерный индекс может быть создан для столбца, содержащего дату заказа, а некластерный — для столбца с номером заказа. Если дело обстоит так, то имеется корреляция (или прямая связь) между увеличивающимися датами и увеличивающимися номерами заказов. Оптимизатор же работает в предположении, что не имеется никакой корреляции, что ведет, в свою очередь, к ошибочным решениям. Если эта проблема возникла, то имеются три потенциальных способа ее решения:

- по возможности нужно переупорядочить составной некластерный индекс так, чтобы самый селективный оказался на первом месте;
- создать покрывающий индекс;
- добавить подсказку оптимизатору запроса.

5.3.3. Составные и покрывающие индексы

Составными называются индексы, включающие в себя два и более столбцов. Общим правилом считается, что при создании составного индекса на первое место нужно ставить столбец с максимальной селективностью. Вообще говоря, составных индексов нужно избегать, потому что они, будучи громоздкими, требуют большой затраты ресурсов сервера на выполнение операций ввода/вывода, снижая тем самым общую производительность. Для оптимизации операции поиска составные индексы сами по себе не пригодны, т. к. их можно использовать, только если опция `WHERE` запроса требует выполнения поиска по столбцу, который является крайним левым в индексе. Так, если создан составной индекс по паре столбцов (`City`, `State`), тогда запрос с условием поиска `WHERE City = 'Москва'` будет использовать индекс, а запрос с условием поиска `WHERE STATE = 'Россия'` не будет.

Когда же полезны составные индексы? Уникальные составные индексы являются единственным средством реализации такого ограничения целостности отношения, как ограничение первичного ключа в том случае, когда первичный ключ состоит из нескольких полей. Но для этого обычно используются кластерные индексы. Некластерные составные индексы полезны в основном в тех случаях, когда они являются так называемыми *покрывающими* индексами (covering indexes).

Эффект использования покрывающих индексов состоит в том, что можно написать запрос таким образом и построить такие индексы, что все результаты запроса будут получены только чтением индексных страниц дисковой памяти на листовом уровне без обращения к страницам памяти, содержащим данные. Понятно, что в этой ситуации производительность запроса существенно возрастает, т. к. налицо экономия времени, связанная с тем, что не нужно выполнять медленные операции чтения данных с диска. Если же покрывающий индекс содержит слишком много столбцов, то это может сильно увеличивать накладные расходы, связанные с его поддержанием, от чего в целом ухудшается эффективность работы сервера. Компромисс здесь достигается, если руководствоваться следующими принципами:

- если запрос выполняется редко, то затраты ресурсов сервера на поддержание покрывающего индекса могут перевешивать выгоды от его использования;

- покрывающий индекс не должен заметно увеличивать размер ключа таблицы;
- покрывающий индекс обычно полезен, если без него в плане выполнения запроса используется операция поиска по закладке (**Bookmark Lookups**). Наличие этой операции как раз и говорит о считывании страницы данных таблицы по уже найденной ссылке на нужную запись на уровне листа индекса;
- при создании покрывающего индекса нужно по возможности использовать те индексы, что уже имеются. Например, требуется создать покрывающий индекс для столбцов `c1` и `c3`. Если уже имеется индекс на столбец `c1`, то вместо создания нового можно изменить индекс на столбец `c1`, включив в него столбец `c3`.

Индексы, создаваемые как покрывающие, редко бывают кластерными. Это объясняется тем, что единственный кластерный индекс лучше использовать для задания ограничения первичного ключа таблицы. Чаще всего покрывающие индексы являются составными, т. к. маловероятно, чтобы запрос был написан для возврата данных из единственного столбца.

Для того чтобы реализовать эффект покрывающего индекса, нужно включить в него все столбцы, которые встречаются в опциях `SELECT`, `JOIN` и `WHERE` запроса. Полезными покрывающие индексы могут оказаться и для выполнения операций группировки.

Как убедиться, что покрывающий индекс фактически используется оптимизатором запроса? Это видно, если в плане запроса присутствует пиктограмма с фразой "Scanning a non-clustered index entirely or only a range", а пиктограммы с операцией **Bookmark Lookups** отсутствует.

Альтернативой ручному созданию покрывающих индексов является возможность позволить SQL-серверу создавать их автоматически. Оптимизатор запроса может осуществлять процесс, который называется *индексным пересечением* и заключается в просмотре всех существующих индексов для построения хеш-таблицы, на основании которой можно выполнить запрос в целом таким образом, что его результаты будут получены только из листовых страниц индексов без чтения страниц данных. Таким образом, эта хеш-таблица и становится индексом покрытия для запроса. Индексированное пересечение выполняется автоматически, нужно только помочь оптимизатору, создав атомарные индексы для всех столбцов таблицы, часто участвующих в запросах.

Индексы могут не только помогать выполнению запросов, но и мешать общему быстродействию сервера в случаях, когда:

- индекс никогда не используется оптимизатором запроса;
- селективность некластерного индекса меньше чем 90—95 %;

- индексированный столбец очень широк;
- запросы к таблице с индексом выполняются редко.

5.3.4. Использование мастера настройки индексов

Настройка индексов базы данных — это сложная задача, которая требует немалого опыта разработчика или администратора. Существует утилита под названием Мастер Настройки Индексов (Index Tuning Wizard), которая позволяет в определенной мере выбирать и создавать оптимальный набор индексов и поддерживать адекватную статистику при отсутствии углубленных знаний о сути процесса оптимизации. Мастер Настройки Индексов можно запустить как обычный мастер из рубрики **Management** утилиты Enterprise Manager (опция меню **Tools/Wizards.../Management**). Запуск Мастера Настройки Индексов можно выполнить и из утилиты SQL Query Analyzer. Для этого в ее окне должен быть текст запроса или программы, тогда в опции меню **Query** активизируется опция **Index Tuning Wizard.** В зависимости от способа запуска Мастера Настройки Индексов, его окна немного отличаются. Так, если Мастер Настройки Индексов запущен из SQL Query Analyzer, то в его третьем окне появляется дополнительный переключатель, указывающий, что анализу будет подвергаться тот текст запроса, который сейчас находится в окне SQL Query Analyzer.

Чтобы создавать рекомендации об оптимальном наборе индексов, Мастер Настройки Индексов требует входной информации или *входного сценария*, который может быть представлен двумя способами. Во-первых, в виде текста запроса на языке T-SQL, и, во-вторых, в виде сохраненного в файле или таблице следа утилиты SQL Profiler, полученного для специального набора классов событий и объектов *(см. приложение 1)*.

Мастер Настройки Индексов может выполнять следующие действия:

- рекомендовать наилучший набор индексов для выбранной базы данных на основе анализа и стоимостной оптимизации запросов, включенных в его входной сценарий;
- реализовывать найденные рекомендации, создавая необходимые объекты базы данных немедленно или генерируя скрипты их создания на языке T-SQL;
- анализировать эффекты от предложенных изменений, генерируя большое количество всевозможных отчетов, сравнивающих текущее состояние базы данных с рекомендованным;
- рекомендовать способ настройки базы данных для небольшого набора прикладных запросов;

- рекомендовать настройки дополнительных параметров, например ограничений дискового пространства;
- находить неиспользуемые индексы, подлежащие удалению.

Кратко рассмотрим последовательность диалоговых окон, открываемых Мастером Настройки Индексов.

В первом диалоговом окне выводится приветствие и краткий перечень возможностей Мастера Настройки Индексов.

Во втором окне можно выбрать экземпляр SQL-сервера и название базы данных, а также задать общие настройки оптимизации индексов. Так, если флажок, соответствующий опции **Keep all existing indexes**, включен, то рекомендации будут выдаваться только по поводу добавления индексов, но не их удаления. Опция **Add indexed views** включает процесс оптимизации и для индексных представлений. Переключатель **Fast/Medium/Thorough** позволяет выбрать уровень тщательности оптимизации.

В третьем окне мастера нужно с помощью переключателей указать, какая входная информация (workload) поступит на его вход. Этой информацией может служить файл с текстом программы на T-SQL (расширение sql), файл, содержащий след утилиты SQL Profiler (расширение log или trc), базовая таблица SQL-сервера, а также текущая программа утилиты SQL Query Analyzer. При необходимости можно настроить еще и дополнительные опции.

В четвертом окне указывается, какие таблицы предполагается выбрать для участия в процессе оптимизации. Для ускорения работы здесь нужно указывать таблицы только из наиболее критических запросов.

После настройки параметров в указанных окнах мастер начинает работу, которая может занять весьма продолжительное время. Полученные рекомендации выводятся в следующем (пятом) окне мастера в виде таблицы с именами рассмотренных объектов и предлагаемыми рекомендациями. Вид этой таблицы представлен в табл. 5.4.

Таблица 5.4. *Пример рекомендаций Мастера Настройки Индексов*

Clustered	Index Name	Table/View Name	Column Name
☑	[IX_T1]	[dbo].T1	`[Col1]` ASC
☐	[IX_T1_1]	[dbo].T1_1	`[Col2]`
☑	[PK_T2]	[dbo].T2	`[Order]`
☐	[IX_T3_IND1]	[dbo].T3	`[Накладная]` ASC

Столбцы табл. 5.4 имеют следующий смысл: Clustered (кластерный), Index Name (имя индекса), Table/View Name (имя таблицы или представления) и Column Name (имя столбца).

Рекомендации в этой таблице показаны с помощью пиктограмм. В первом столбце (Clustered) указано, является ли индекс кластерным ☑ или нет ☐. Во втором столбце (Index Name) указано имя индекса и приведена рекомендация по поводу его полезности. Пиктограмма указывает, что индекс нужно создать, пиктограмма говорит о том, что индекс не используется для обработанной входной информации, а пиктограмма свидетельствует о наличии индекса, нужного для работы какого-либо из проанализированных запросов. Третий (Table/View Name) и четвертый (Column Name) столбцы соответственно выводят имя таблицы (или индексированного представления) и названия столбца (столбцов), по которым строится индекс.

В окне рекомендаций имеется кнопка с надписью **Analysis...**. Если ее нажать, то можно просмотреть более подробные отчеты Мастера Настройки Индексов. Названия этих отчетов выбираются из списка. Особенно полезен отчет под названием **Index Usage Report (Recommended Configuration)**, который содержит в себе перечень рекомендованных мастером индексов. Результаты отчетов выводятся в табличном виде. Их можно анализировать немедленно или сохранить в текстовых файлах.

Последние два окна (шестое и седьмое) Мастера Настройки Индексов содержат опции, позволяющие выполнить сформулированные рекомендации, установив флажок **Apply Changes**. Необходимые индексы можно создать немедленно, можно указать время, когда их создавать, а можно сохранить в файле сценарий выполнения рекомендаций и работать с ним по своему усмотрению. Если полученные рекомендации вам не нравятся и применять их вы не собираетесь, то выходить из мастера нужно либо нажатием клавиши <Esc> клавиатуры, либо с помощью кнопки **Cancel**, которая присутствует в любом окне мастера. Выход из шестого окна Мастера Настройки Индексов путем нажатия кнопки **Next** с последующим нажатием кнопки **Finish** в седьмом окне приведет к выполнению рекомендаций при установленном флажке **Apply Changes**. Выполнять эти действия нужно крайне осмотрительно — ведь откат сделать будет невозможно. Особенно неприятна ситуация поспешного выхода из Мастера Настройки Индексов в том случае, когда вы применяли его для оптимизации одного запроса, выключив опцию **Keep all existing indexes** и выбрав все таблицы. В этом случае при нажатии кнопки **Finish** из базы данных будут удалены почти все индексы. Поэтому лучше все же перед непосредственной реализацией рекомендаций выполнять их сохранение в виде скрипта — тогда по нему можно разобраться в том, какие изменения были сделаны и, в крайнем случае, запрограммировать откат вручную.

В листинге 5.3 приведена часть программы, содержащей рекомендации Мастера Настройки Индексов для единственного запроса базы данных NORTHWIND при снятом флажке **Keep all existing indexes**. Однострочные комментарии на русском языке добавлены вручную.

Листинг 5.3. Пример рекомендаций Мастера Настройки Индексов

```
/* Created by: Index Tuning Wizard  */
/* Date: 8/16/2004 *     /
/* Time: 4:50:56 PM *
/* Server Name: SERVER*  /
/* Database Name: Northwind  */
USE [Northwind]
Go
--Настройка параметров соединения, позволяющего создавать и удалять
-- индексы
SET QUOTED_IDENTIFIER ON
SET ARITHABORT ON
SET CONCAT_NULL_YIELDS_NULL ON
SET ANSI_NULLS ON
SET ANSI_PADDING ON
SET ANSI_WARNINGS ON
SET NUMERIC_ROUNDABORT OFF
go
DECLARE @bErrors as bit -- переменная для хранения кода ошибки
       -- транзакция 1: удаление ненужных индексов
BEGIN TRANSACTION
SET @bErrors = 0
DROP INDEX [dbo].[Employees].[PostalCode]
DROP INDEX [dbo].[Employees].[LastName]
IF( @bErrors = 0) COMMIT TRANSACTION
ELSE ROLLBACK TRANSACTION
    --транзакция 2: удаление двух ненужных индексов и создание вместо них
    --одного нужного индекса для проанализированного запроса
BEGIN TRANSACTION
SET @bErrors = 0
DROP INDEX [dbo].[OrderDetails].[OD_ProductID]
DROP INDEX [dbo].[OrderDetails].[OD_Quantity]
CREATE CLUSTERED INDEX [OrderDetails1]
ON [dbo].[OrderDetails] ([ProductID] ASC)
IF( @@error <> 0) SET @bErrors = 1
IF( @bErrors = 0) COMMIT TRANSACTION
ELSE ROLLBACK TRANSACTION
```

Из листинга 5.3 видно, что сгенерированный сценарий разбит на последовательность транзакций, каждая из которых объединяет изменения, относящиеся к одной таблице. Переменная `@bErrors` введена для организации обработки ошибок. В данном случае код ошибки анализируется только после выполнения операции создания индекса.

Замечание

При генерации рекомендаций Мастер Настройки Индексов присваивает создаваемым индексам имена, принятые SQL-сервером по умолчанию для вновь создаваемого индекса. Если ваша стратегия присвоения имен объектам базы данных отличается от стратегии, принятой по умолчанию, что обычно является разумным решением, то в сценарии выполнения рекомендаций Мастера Настройки Индексов нужно изменить имена индексов в соответствии с принятыми правилами.

Имеется существенное ограничение по использованию Мастера Настройки Индексов. Он не может давать рекомендации в тех случаях, когда недостаточно данных в таблицах. То есть для маленьких таблиц или в случаях, когда найденная мастером рекомендация не позволяет заметно повысить эффективность выполнения запроса по сравнению с существующими индексами, будет выдано сообщение об отсутствии рекомендаций. Если рекомендации сформулированы, то вместе с ними выводится значение, показывающее прирост эффективности в случае выполнения рекомендаций. Это значение вычисляется путем вычитания из единицы отношения стоимости запроса до применения рекомендаций мастера к стоимости запроса после их применения.

В дальнейшем будут приведены примеры использования Мастера Настройки Индексов для оптимизации различных запросов в базе данных `МАТ_УЧЕТ`.

5.4. Оптимизация поиска данных по заданному условию (опция *WHERE*)

Поиск данных по заданному условию является одной из наиболее часто выполняемых операций. Усилия по повышению быстродействия этой операции могут быть разделены на две категории:

1. Сам способ записи логического условия (предиката) в опции `WHERE` должен быть таким, чтобы его вычисление для конкретной записи выполнялось бы наиболее эффективно.
2. Крайне важен способ перебора массива записей, который должен быть таким, чтобы наиболее быстро выйти на проверку именно тех записей, которые удовлетворяют условию поиска, и по возможности вообще не проверять логическое условие для записей, которые ему удовлетворять не могут.

Первое требование достигается путем конструирования логических условий в опции `WHERE`, подчиняющихся определенным правилам, связанным с их вычислением.

Второе требование может быть выполнено, если удастся сконструировать так называемые *sargable* условия поиска (от псевдоакронима "SARG" или "*S*earch *ARG*ument"). Примерный перевод этого термина обозначает "пригодные к поиску", а смысл состоит в том, что для логических условий такого типа возможно использование индексного доступа к записям и выполнение бинарного поиска вместо полного сканирования таблицы.

5.4.1. Некоторые правила вычисления логических условий

Операторы сравнения, используемые в опции `WHERE`, обладают разным быстродействием. В списке, приведенном далее, они упорядочены по скорости выполнения, начиная с самого быстрого:

- =;
- >, >=, <, <=;
- `LIKE`;
- <>.

Чем более простые (короткие) операнды используются в операциях сравнения, тем выше скорость выполнения запроса. Сложные выражения с обилием скалярных функций сами по себе долго вычисляются, поэтому рекомендуется упрощать операнды сравнения везде, где только возможно.

Если в запросе имеется предложение `WHERE`, которое включает выражения, связанные несколькими конъюнкциями `AND`, то при отсутствии подходящих индексов и статистики эти выражения будут выполняться слева направо в том порядке, в котором они записаны, и никакие круглые скобки не могут изменить этого порядка. Принцип проверки последовательности конъюнкций таков, что если первое выражение окажется ложным, то остальные проверяться не будут. Поэтому целесообразно располагать первыми выражения, истинность которых наименее вероятна. Если для пары выражений вероятность истинности примерно одинакова, то первым нужно размещать наименее сложное из них.

Предикат `IN` оптимизатором преобразовывается к последовательности дизъюнкций `OR` со всеми вытекающими отсюда особенностями. Если имеется возможность использовать либо предикат `IN`, либо предикат `BETWEEN`, то `BETWEEN`, как правило, более эффективен.

Так, вместо

```
WHERE COLUMN IN (1000, 1001, 1002, 1003, 1004)
```

лучше написать

```
WHERE COLUMN BETWEEN 1000 AND 1004.
```

Если преобразование предиката IN к другому предикату невозможно, то список значений нужно упорядочить таким образом, чтобы наиболее часто встречающиеся значения оказались в начале списка. Эта мера может ускорить проверку, потому что предикат IN возвращает истину, как только любое из значений в списке совпадет с проверяемым параметром, поэтому чем скорее наступит совпадение, тем быстрее проверка завершится.

Если в опции WHERE часто выполняется поиск по шаблону с помощью предиката LIKE и из-за этого возникают проблемы быстродействия, то следует рассмотреть возможность использования службы полнотекстового поиска SQL-сервера под названием MicrosoftSearch, которая работает намного быстрее.

Вычисление скалярных функций всегда является долгой операцией, поэтому логические выражения должны быть как можно проще, с этой точки зрения. Например, для вычисления года по дате можно сконструировать громоздкое выражение

```
convert(int, substring(convert(varchar(50), GETDATE()),8,4))
```

Хотя общеизвестно, что существует стандартная функция для этой цели и гораздо разумнее эту задачу решать с ее помощью: year(GETDATE()).

Следует помнить, что бессмысленно употреблять функции LOWER и UPPER, если база данных сконфигурирована нечувствительной к регистру символов, как это принято по умолчанию. Если же в базе данных регистр символов учитывается, то вместо запроса в стиле

```
SELECT COLUMN FROM TABLE WHERE LOWER(COLUMN) = 'name'
```

гораздо эффективнее будет выполняться эквивалентный запрос

```
SELECT COLUMN FROM TABLE
WHERE COLUMN = 'NAME' OR COLUMN = 'name'
```

5.4.2. Опция *WHERE* и агрегирование

Инструкция группировки GROUP BY выполняется после выборок и соединений. Вместе с группировкой может быть использована инструкция HAVING, которая служит для задания логических условий на выражения, содержащие агрегатные функции. Однако в опции HAVING можно записать логическое условие относительно какого-нибудь из полей группировки и без агрегатной функции.

Рассмотрим, например, задачу подсчета суммарной стоимости каждой приходной накладной с номером в диапазоне от 30 до 100 для основного проектного решения базы данных MAT_УЧЕТ. Ее решение можно получить с помощью следующего оператора:

```
SELECT ПРИХОДЫ.N_прих,
SUM(ТОВАРЫ.Цена_учетн * ПРИХОДЫ.Кол_прих) AS Затраты
FROM ТОВАРЫ INNER JOIN ПРИХОДЫ ON ТОВАРЫ.Товар_Ид = ПРИХОДЫ.Товар_Ид
GROUP BY N_прих
HAVING N_прих between 30 and 100
```

Можно в этом операторе условие N_прих between 30 and 100 разместить в опции WHERE, а не в опции HAVING. План запроса от этого никак не изменится, все равно первой операцией в нем будет выборка по условию, затем соединение таблиц, после чего выполняется группировка с использованием алгоритма хеширования. Из этого примера видно, что логические условия без использования агрегатных функций эквивалентны, с точки зрения их размещения в опции WHERE или HAVING.

5.4.3. Использование индексов с опцией *WHERE*

Основной способ повышения быстродействия опции WHERE состоит в применении индексов для организации поиска, что позволяет использовать алгоритм бинарного поиска в сбалансированном B-дереве *(см. главу 4)* вместо полного сканирования таблицы.

Пригодными для индексного поиска, в первую очередь, считаются столбцы, используемые в логических выражениях с оператором сравнения по равенству, если они связаны функцией AND. Это наиболее типичная ситуация. Упомянутые столбцы отличаются друг от друга по степени селективности. Напомним, что высокой степенью селективности обладают столбцы с относительно большим количеством разных значений, поэтому выборка по строгому равенству для таких столбцов возвращает относительно малое количество записей. Рекомендуется, чтобы среди критериев поиска в предложении WHERE, по крайней мере, один был основан на высокоселективном столбце, который имеет индекс. Если столбца с достаточной степенью селективности не обнаружено, то можно рассмотреть создание составного покрывающего индекса *(см. разд. 5.3)*.

Но даже если необходимые индексы созданы, вполне возможно, что они не будут использоваться из-за того, что не все логические предикаты операций сравнения допускают использование индексов по своей сути. Индексы, как правило, не применяются для выполнения операции поиска со следующими предикатами: IS NULL, <>, !=, !>, !<, NOT, NOT EXISTS, NOT IN, NOT LIKE. Кроме

того, не может использовать индекс оператор LIKE, начинающийся с символа подстановки. Другими словами, условие поиска COLUMN LIKE 'П%' имеет потенциальную возможность использования индекса, а условие поиска COLUMN LIKE '%П' такой возможности не имеет, хотя можно вынудить сервер применить индекс и в этом случае с помощью подсказки оптимизатору *(см. разд. 5.6)*. Поэтому если в запросах используется поиск по шаблону, то нужно стараться записывать условие поиска с ведущих символов поиска, а не начинать сразу с символа подстановки.

Опция WHERE с логическим отрицанием NOT не использует поиск по индексу. Однако часто можно переписать условие поиска таким образом, чтобы операция NOT явно не использовалась. Простейший пример состоит в следующем: вместо WHERE NOT COLUMN > 5 написать WHERE COLUMN <= 5.

Особую ситуацию представляет использование скалярных функций в условии поиска и их взаимодействие с индексами. Для поиска по индексу непригодны выражения, содержащие скалярные функции, аргументами которых являются поля таблиц, даже если имеются индексы на сами эти поля. Кроме того, поиск по индексу не будет выполняться для выражений, содержащих одноименные столбцы слева и справа от операции сравнения. Например, для поиска по индексу фамилий, начинающихся с заданной буквы, вместо WHERE SUBSTRING(ФИО,1,1) = 'М' лучше написать WHERE ФИО LIKE 'М%'. Первое выражение не может использовать индекс по столбцу ФИО, а второе — может.

Другой пример сложнее. Рассмотрим таблицу с названием ПЕРСОНАЛ, содержащую сведения о людях, первичным ключом которой является идентификатор человека (поле PERSON_ID). Пусть в этой таблице имеется поле BIRTHDAY, содержащее дату рождения. Нам нужно получить сведения о людях, которые старше 60 лет. Запрос, решающий эту задачу, можно написать следующим образом:

```
SELECT * FROM ПЕРСОНАЛ
WHERE DATEDIFF(yy, BIRTHDAY,GETDATE()) > 60
```

Здесь скалярная функция DATEDIFF применяется непосредственно к полю BIRTHDAY, поэтому использование индекса по этому полю, даже если он и создан, невозможно. Если переписать этот запрос эквивалентным образом:

```
SELECT * FROM ПЕРСОНАЛ
WHERE BIRTHDAY < DATEDIFF(yy, -60, GETDATE()),
```

то препятствий для использования индекса нет.

Если в опции WHERE используется функция SOUNDEX, оптимизатор игнорирует любые доступные индексы, выполняя сканирование таблицы, что для большой таблицы крайне неэффективно. Если нужно выполнить исследования

с помощью SOUNDEX, то стоит предварительно вычислить код SOUNDEX для столбца, сохранить его в таблице и построить соответствующий индекс.

Интересные возможности дает использование индексов на вычислимые столбцы таблиц. Вычислимые столбцы на SQL-сервере могут быть проиндексированы, если они удовлетворяют следующим условиям:

- выражение вычислимого столбца детерминировано, что означает отсутствие в нем *недетерминированных* функций, т. е. таких, которые выдают разные значения при их неоднократном вызове;
- все столбцы в выражении для вычислимого столбца должны выбираться из одной и той же таблицы;
- недопустимо использование типа данных FLOAT;
- в выражении для вычислимого столбца нельзя использовать агрегатные функции;
- опция ANSI_NULL подключения к серверу была включена, когда таблица создавалась.

Если на вычислимом столбце создается кластерный индекс, то вычисленные значения сохраняются в таблице точно так же, как для любого кластерного индекса. Если создается некластерный индекс, вычисленные значения сохраняются в индексе, а не в фактической таблице.

Добавление индекса к вычислимому столбцу редко бывает полезным, т. к. модификация такого индекса по сложному выражению требует больших затрат ресурсов сервера.

Наиболее часто используется прием добавления индекса к вычисляемому столбцу для функции CHECKSUM, что является новой особенностью SQL-сервера 2000. Эта целочисленная функция (тип данных int) используется, чтобы создавать новое значение, которое представляет другие данные в таблице. Значение CHECKSUM является по сути хеш-индексом строкового выражения — аргумента этой функции. В большинстве случаев значение CHECKSUM будет намного короче, чем представляемое ею фактическое строковое значение. Обычно значения контрольной суммы, выдаваемые функцией CHECKSUM, уникальны, однако это не гарантируется во всех случаях. Возможно, что два слегка различных значения фактических данных производят идентичные значения CHECKSUM, но эта функция всегда возвращает тот же самый результат для одного и того же аргумента.

Как эта функция работает, покажем на примере базы данных о музыке, в которой нужно организовать поиск данных по названиям песен. Строки для поиска могут быть весьма длинными, например что-то вроде "Мой лучший друг мул из Миссури". Добавление индекса к столбцу заголовка песни сделало бы

этот индекс очень широким, поэтому лучше создать индекс по значению контрольной суммы названия песни. Но как вводить бессмысленные с точки зрения пользователя коды, выдаваемые функцией CHECKSUM, в качестве условия поиска песни? Рассмотрим код программы из листинга 5.4, в котором помимо всего прочего показано, как создавать индекс на вычислимый столбец.

Листинг 5.4. Функция CHECKSUM и индексы на вычислимые столбцы

```
--Создание индекса
ALTER TABLE SONGS ADD checksum_title AS checksum(title)
CREATE INDEX title_index ON SONGS (checksum_title)
GO
-- Запрос на поиск песни по названию
SELECT title, artist, composer FROM SONGS
WHERE title = 'Мой лучший друг мул из Миссури'
AND checksum_title = CHECKSUM('Мой лучший друг мул из Миссури')
GO
```

Запрос из последнего листинга выглядит странным — кажется, что два раза выполняется поиск по одинаковому условию. Однако на такую меру приходится идти, помня, что уникальные значения контрольной суммы не гарантируются. Работает этот запрос следующим образом. Оптимизатор исследует опцию WHERE, и, найдя высокоселективный индекс по столбцу checksum_title, решает его использовать. Поскольку используется индекс, диапазон поиска очень быстро сужается, и для выполнения первого сравнения опции WHERE нужно проверить совсем небольшое количество записей.

5.4.4. Примеры тестов для индексов с опцией *WHERE*

Приведенные выше голословные рекомендации по выполнению поиска с индексами и без них полезно проверить. Интересно рассмотреть влияние наличия индексов на способ выполнения поиска. Для простоты поиск будет выполняться по предикатам строгого равенства. В этом случае поведение оптимизатора наиболее предсказуемо.

Пример 1. Влияние селективности столбца на использование индекса

Создадим таблицу с именем ПРИХОДЫ_1, полностью идентичную таблице ПРИХОДЫ базы данных МАТ_УЧЕТ, сначала вообще без индексов и заполним ее данными из таблицы ПРИХОДЫ.

Проще всего это сделать с помощью оператора

```
if exists (select * from dbo.sysobjects
where id = object_id(N'[dbo].[ПРИХОДЫ_1]')
and OBJECTPROPERTY(id, N'IsUserTable') = 1)
drop table [dbo].[ПРИХОДЫ_1]
GO
SELECT * INTO ПРИХОДЫ_1 FROM ПРИХОДЫ
```

Высокоселективным в этом случае является столбец `N_прих`. Необходимо для таблицы `ПРИХОДЫ_1` проверить, присутствует ли в приходной накладной с заданным номером, равным 100, заданный товар с кодом, равным 11. Это делает запрос

```
SELECT * FROM ПРИХОДЫ_1 WHERE N_прих = 100 and Товар_Ид = 11
```

Без индексов, естественно, выполняется полное сканирование таблицы, на что затрачивается время порядка 50 мс. В соответствии с теорией создадим уникальный кластерный индекс по высокоселективному столбцу `N_прих`, участвующему в условии поиска. Этот индекс можно создать с помощью команды

```
CREATE UNIQUE CLUSTERED INDEX [IX_ПРИХОДЫ_1]
ON [dbo].[ПРИХОДЫ_1]([N_прих]) ON [PRIMARY]
GO
```

Запрос теперь выполняется около 1 мс, а его план изображен на рис. 5.4. На плане запроса из рис. 5.4 показана часть сведений об операции индексного поиска.

Кстати, если применить Мастер Настройки Индексов для анализируемого запроса, то будет сформулирована рекомендация о создании именно кластерного индекса на столбец `N_прих`, как это и сделано в последнем случае. Уникальность этого индекса не влияет на стоимость запроса.

Вместо кластерного индекса можно создать обыкновенный индекс с помощью оператора

```
CREATE INDEX [IX_ПРИХОДЫ_1] ON [dbo].[ПРИХОДЫ_1]([N_прих]) ON [PRIMARY]
GO
```

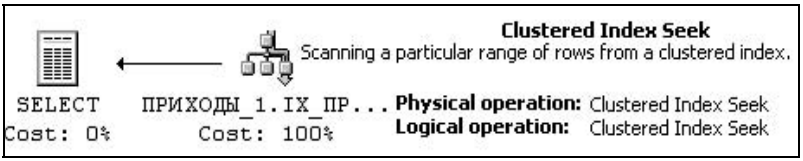


Рис. 5.4. План запроса при наличии кластерного уникального индекса по высокоселективному столбцу

В этом случае план запроса будет вместе с расшифровкой части сведений об операции индексного поиска соответствовать рис. 5.5, а время его выполнения возрастет. Элементарная операция `Filter` в плане запроса соответствует проверке предиката `Товар_Ид = 11`.

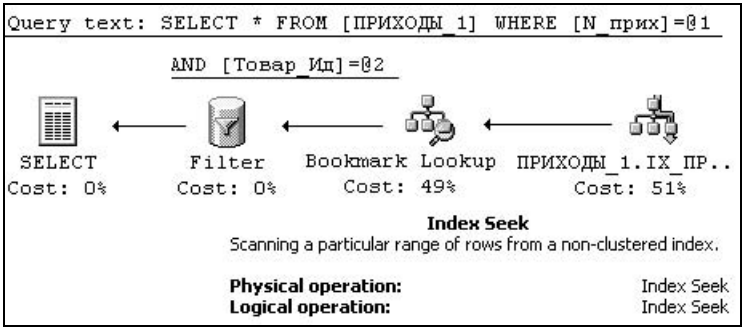


Рис. 5.5. План запроса при наличии обыкновенного индекса по высокоселективному столбцу

Замечание

Последний результат нельзя назвать непредсказуемым. И без Мастера Настройки Индексов можно было сообразить, что поиск значений всех полей таблицы (`SELECT*`) по полному совпадению со значением первичного ключа (поле `N_прих`) будет максимально эффективным с использованием именно кластерного индекса. Так происходит потому, что селективность критерия поиска в этом случае равна 100 %, а то, что используется кластерный индекс, гарантирует совпадение листовых страниц индекса со страницами данных, устраняя операцию поиска по закладке, которая, если создать вместо кластерного индекса некластерный, будет присутствовать в плане запроса, увеличив стоимость запроса примерно вдвое.

Теперь вместо индекса по полю `N_прих` создадим некластерный индекс по низкоселективному столбцу `Товар_Ид` и посмотрим на план запроса. Команда создания индекса теперь такая:

```
CREATE INDEX [IX_ПРИХОДЫ_1] ON [dbo].[ПРИХОДЫ_1]([Товар_Ид]) ON [PRIMARY]
GO
```

Этот индекс оказался бесполезным, с точки зрения оптимизатора, — вместо его использования выполняется полное сканирование таблицы, что занимает то же самое время, как и в случае полного отсутствия индексов. Попытка вынудить сервер использовать индекс для поиска по низкоселективному столбцу `Товар_Ид` путем применения подсказки привела к увеличению общего времени выполнения запроса примерно в полтора раза.

Можно по низкоселективному столбцу `Товар_Ид` создать кластерный индекс с помощью команды

```
CREATE CLUSTERED INDEX [IX_ПРИХОДЫ_1] ON [dbo].[ПРИХОДЫ_1]([Товар_Ид])
ON [PRIMARY]
GO
```

Будет использоваться созданный последней командой индекс (это показано в плане запроса на рис. 5.6). Время выполнения запроса в этом случае будет около 4 мс.

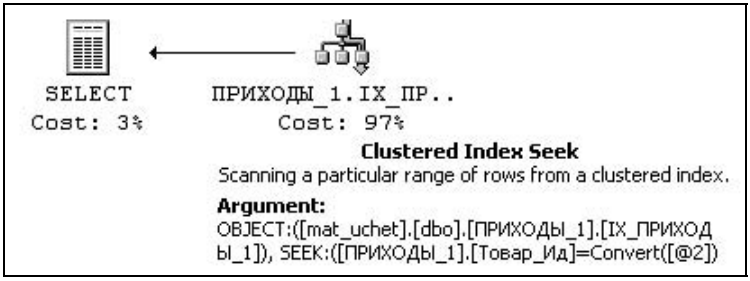


Рис. 5.6. План запроса при наличии кластерного индекса по низкоселективному столбцу

Замечание

На первый взгляд планы запросов для рис. 5.4 и 5.6 выглядят одинаково, но если просмотреть детальную информацию об операции индексного поиска, то можно увидеть, что эти операции имеют разный аргумент поиска.

Разобранный пример дает четкое представление о том, как используются индексы для поиска данных и насколько они полезны в том или ином случае.

Пример 2. Индексы по паре высокоселективных столбцов

Приведем еще пример плана запроса, в котором используются сразу два индекса для выполнения поиска. Этот пример получен на основе стандартной учебной базы данных `NORTHWIND`. Рассмотрим запрос

```
SELECT * FROM OrderDetails WHERE ProductID=10 and quantity=24
```

План этого запроса приведен на рис. 5.7.

Использованная в запросе таблица `OrderDetails` сконструирована специально в иллюстративных целях. Она является копией таблицы с именем `[Order Details]` базы данных `NORTHWIND`, только в ней убран первичный ключ и созданы индексы отдельно по столбцам `ProductID` и `quantity` с помощью операторов

```
CREATE INDEX [OD_ProductID] ON [dbo].[OrderDetails]([ProductID])
ON [PRIMARY]
GO
```

```
CREATE INDEX [OD_Quantity] ON [dbo].[OrderDetails]([Quantity])
ON [PRIMARY]
GO
```

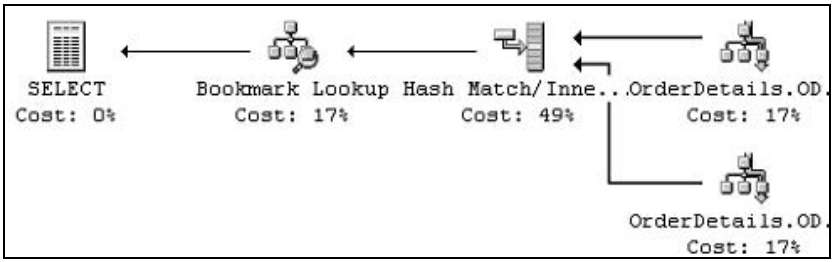


Рис. 5.7. План запроса, содержащего поиск сразу по двум индексам

Оба столбца, использованные для построения индексов, высокоселективны, поэтому оптимизатор использует сразу оба индекса для поиска в отдельности по каждому из условий, после чего результаты поисков соединяются с помощью `INNER JOIN` по закладке. Без индексов запрос будет выполняться путем полного сканирования таблицы, которое работает медленнее.

5.5. Соединение таблиц

Соединение таблиц является самой дорогостоящей операцией реляционной алгебры, не считая операции декартового произведения, поэтому она оказывает существенное влияние на быстродействие запроса в целом. Задается эта операция в опции `FROM` оператора `SELECT` в том случае, когда в запросе участвует более одной таблицы или таблица соединяется со своей копией.

В результате выполнения бинарной операции реляционной алгебры под названием *тета-соединение* создается результирующее отношение, которое содержит кортежи, удовлетворяющие заданному предикату, применяемому к декартовому произведению исходных отношений. Если этот предикат содержит только операции равенства, то такое соединение называется *соединением по эквивалентности* или *эквисоединением*. Эквисоединение двух отношений по всем полям, содержащим данные из одинаковых доменов, называется *естественным соединением*.

Общие рекомендации, позволяющие повысить эффективность выполнения операции соединения, в первую очередь связаны с правильным использованием индексов. Суть этих рекомендаций кратко можно сформулировать так: если нужно выполнять регулярные соединения таблиц, то эффективность в общем случае будет оптимальной при соединении столбцов одинакового типа данных, каждый из которых имеет собственный индекс. При этом считает-

ся, что кластерные индексы лучше некластерных, и чем выше селективность столбцов, по которым выполняется соединение, тем лучше.

На скорость соединения таблиц по любому из способов оказывает влияние тип данных полей связи. Считается, что наиболее быстро выполняется соединение для полей целочисленных типов данных. Если столбцы, которые часто участвуют в соединениях, недостаточно компактны, то стоит рассмотреть возможность введения суррогатных ключей. Время выполнения операции соединения для полей строкового типа данных зависит от реальной степени заполнения строк для типов данных `VARCHAR`.

Если запрос требует соединения более четырех таблиц, следует подумать о денормализации базы данных или о создании *индексированного представления* (Indexed View) для предварительного соединения таблиц *(см. разд. 5.12)*.

Следует избегать использования звездочки для возврата всех столбцов в обеих соединяемых таблицах. Это плохо по двум причинам. Во-первых, вряд ли возникнет ситуация, когда понадобятся все столбцы из всех таблиц, так что, скорее всего, звездочка ведет к передаче клиенту лишних данных. Во-вторых, уж точно ни к чему возвращать по два раза столбцы, по которым выполнялось эквисоединение.

Кроме того, эффективность выполнения операции соединения существенно зависит от того, сколько строк размещено в странице данных. Например, при соединении двух таблиц наиболее вероятно, что меньшая из них будет выбрана в качестве внутренней таблицы соединения, и тогда сервер должен поместить содержимое этой таблицы в буферный кэш. Если все нужные данные удастся сразу кэшировать, эффективность соединения будет существенно выше. Мораль, которая следует из этого, состоит в том, что нужно стремиться заполнять страницы максимально плотно. Этого можно достичь с помощью указания высокого значения `FILLFACTOR` для индексных страниц, а также путем оптимизации типов данных и размеров столбцов в таблицах. Кроме того, нужно не забывать своевременно перестраивать индексы.

Операцию соединения таблиц можно записывать, используя два различных типа синтаксиса: ANSI или Microsoft. Стиль ANSI является стандартным, и именно он реализован в визуальных средствах конструирования запросов, в то время как стиль Microsoft считается устаревшим. План запроса и способ его выполнения не зависят от выбранного стиля синтаксиса. Рекомендуется все же использовать синтаксис ANSI как более наглядный. Отличия двух способов записи легче всего показать на примере из листинга 5.5.

Листинг 5.5. Два синтаксиса записи операции соединения таблиц

```
--ANSI JOIN синтаксис
SELECT fname, lname, department
```

```
FROM names INNER JOIN departments
ON names.employeeid = departments.employeeid
--Старый Microsoft JOIN синтаксис
SELECT fname, lname, department
FROM names, departments
WHERE names.employeeid = departments.employeeid
```

5.5.1. Способы соединения таблиц

Далее перечислены основные способы соединения таблиц, реализованные на SQL-сервере, вместе с пиктограммами их графического представления в плане запроса.

`LOOP JOIN` — соединение блоками с применением вложенных циклов (с использованием или без использования индексов).

`MERGE JOIN` — соединение сортировкой-слиянием.

`HASH JOIN` — соединение посредством использования функции перемешивания (хеш-функции).

Подробнее ознакомиться с алгоритмами этих способов соединения можно в [13], здесь же будет приведено только словесное описание сути алгоритма.

Соединение блоками с использованием вложенных циклов является самым простым, и оптимизатор всегда начинает выбор способа соединения с него. По этому методу выполняется соединение по одному кортежу за цикл. Во внешнем цикле выполняется последовательный перебор кортежей первого отношения, а во внутреннем цикле последовательно просматриваются все кортежи второго отношения и в результирующий набор заносятся данные, удовлетворяющие предикату соединения. Легко показать, что для повышения быстродействия во внешнем цикле должны считываться кортежи таблицы, обладающей меньшим размером. Оптимизатор запроса сам определяет, какую из пары таблиц перебирать во внешнем цикле, а какую — во внутреннем, поэтому порядок указания таблиц в опции `FROM` оператора `SELECT` роли не играет. Если для соединяемых атрибутов внутреннего отношения существует индекс, то неэффективная операция сканирования внутренней таблицы может быть заменена поиском по индексу.

Если оба соединяемых отношения отсортированы по соединяемым атрибутам, то для выполнения эквисоединения наиболее эффективным является *соединение посредством сортировки-слияния.* Поскольку кортежи отношений расположены в порядке сортировки, то те из них, которые содержат одинаковые значения соединяемого атрибута, гарантированно будут размещаться рядом друг с другом. Таким образом, для выбранного кортежа первой таблицы

можно сразу присоединить все подходящие кортежи второй таблицы, а поиск по внутреннему циклу вообще оказывается ненужным. Но как сообщить оптимизатору, что данные отсортированы должным образом? Ответом на этот вопрос является наличие соответствующих индексов у полей связи соединяемых таблиц. При этом важно, какие это индексы — кластерные или нет.

Соединение посредством перемешивания является наиболее сложным типом соединения. Суть этого алгоритма состоит в разбиении соединяемых отношений на разделы в соответствии с некоторой функцией перемешивания, результаты вычисления которой равномерно распределены случайным образом. Каждый полученный раздел для каждого из отношений должен иметь одинаковые значения соединяемых атрибутов и может содержать больше одного их значения. Алгоритм проверяет эквивалентные разделы на наличие одинаковых значений, выполняя, таким образом, эквисоединение кортежей. Соединение посредством перемешивания обычно выбирается в ситуации, когда отсутствуют подходящие индексы, позволяющие повысить эффективность выполнения операции.

На практике наиболее часто соединяются таблицы, логически связанные ограничением внешнего ключа. Физически для реализации такой связи должен обязательно быть создан уникальный индекс по полю связи родительской таблицы. Наличие индекса в поле связи дочерней таблицы не является обязательным, и при создании ограничения внешнего ключа автоматически этот индекс не создается, но почти всегда его необходимо создать вручную, пусть даже он будет обладать малой селективностью. Во-первых, дочерняя таблица при эквисоединении с помощью вложенных циклов очень часто оказывается таблицей, обрабатываемой во внутреннем цикле, поэтому для нее наличие индекса для поиска по полю связи может существенно ускорить работу. Во-вторых, даже при малой селективности индекса поля связи дочерней таблицы его наличие позволяет оптимизатору использовать метод `MERGE JOIN`, который может оказаться наиболее эффективным.

Для пояснения сказанного рассмотрим простейший пример эквисоединения пары таблиц из базы данных `МАТ_УЧЕТ`, связанных ограничением внешнего ключа. Родительская таблица `ТОВАРЫ` по полю связи имеет уникальный кластерный индекс (ограничение первичного ключа), дочерняя таблица `ПРИХОДЫ` тоже проиндексирована по полю связи, но индекс здесь некластерный, потому что кластерный индекс использован для реализации ограничения первичного ключа дочерней таблицы. Никаких других индексов у этих таблиц нет. Оптимизатор выбирает соединение сортировкой-слиянием для запроса со следующим текстом:

```
SELECT ПРИХОДЫ.Товар_Ид, ТОВАРЫ.Товар_Ид, ТОВАРЫ.Цена_учетн
       FROM ТОВАРЫ INNER JOIN ПРИХОДЫ
       ON ТОВАРЫ.Товар_Ид = ПРИХОДЫ.Товар_Ид
```

Однако стоит добавить вывод еще каких-нибудь столбцов из таблицы ПРИХОДЫ, не входящих в кластерный индекс, как выполнение соединения сортировкой-слиянием становится невозможным, потому что требуется доступ к неупорядоченной информации — с помощью некластерного индекса данные из неключевых столбцов таблицы ПРИХОДЫ не могут быть упорядочены. Так, примерно вдвое дольше с помощью HASH JOIN выполняется запрос

```
SELECT ПРИХОДЫ.Товар_Ид, ТОВАРЫ.Товар_Ид,
       ТОВАРЫ.Цена_учетн*Кол_прих AS ОБЩ_ЦЕНА
       FROM ТОВАРЫ INNER JOIN ПРИХОДЫ
       ON ТОВАРЫ.Товар_Ид = ПРИХОДЫ.Товар_Ид
```

Стандартное ограничение внешнего ключа в ситуации, когда обе таблицы, как и положено, имеют ограничение первичного ключа, обычно приводит к созданию составного кластерного индекса для дочерней таблицы, физически реализующего ее первичный ключ. При создании этого индекса важен порядок полей. Так, для запроса

```
SELECT * FROM ТОВАРЫ INNER JOIN ПРИХОДЫ
ON ТОВАРЫ.Товар_Ид = ПРИХОДЫ.Товар_Ид
```

при создании индекса ограничения первичного ключа дочерней таблицы оператором

```
CREATE UNIQUE CLUSTERED INDEX [PK_ПРИХОДЫ]
ON [dbo].[ПРИХОДЫ] ([Товар_Ид], [N_прих])ON [PRIMARY]
```

соединение таблиц выполняется с помощью MERGE JOIN. Если же переставить поля в индексе PK_ПРИХОДЫ, то выполнение MERGE JOIN становится невозможным, оптимизатор выбирает соединение типа HASH JOIN, и для проведенного теста запрос выполняется дольше, чем в случае MERGE JOIN.

Если оптимизатор выбрал не самый лучший, с точки зрения пользователя, способ соединения таблиц, то его можно изменить с помощью *подсказки оптимизатору.* Синтаксис этой подсказки следующий:

```
FROM table_one INNER [LOOP | MERGE | HASH] JOIN table_two
```

Вместо внутреннего соединения (INNER JOIN) может быть указано также и любое из внешних соединений — подсказка в этом случае задается аналогично.

Если применена подсказка оптимизатору для указания способа соединения, то будет выполняться именно указанный тип соединения, что подтверждается соответствующим сообщением и отражается при просмотре графического плана выполненного запроса. Добавление подсказки соединения иногда может затронуть другие операции в плане запроса, например, использование

подсказки `MERGE` для таблиц без нужных индексов ведет к необходимости предварительного выполнения сортировки данных в память, что отражается дополнительной пиктограммой на плане запроса. Поэтому при анализе эффективности способа соединения необходимо учитывать суммарную стоимость как подготовительных операций, так и собственно самого соединения.

Общая рекомендация Microsoft по поводу использования подсказок состоит в том, что ими не следует особенно увлекаться — скорее всего, оптимизатор выберет наилучшую стратегию соединения самостоятельно. Однако экспериментальным путем иногда можно найти подходящую подсказку для медленного запроса. Для этого нужно просто попробовать применить подсказки с указанием другого способа соединения и либо просмотреть стоимость запроса, либо замерить время его выполнения. В качестве примера повышения эффективности выполнения запроса с помощью подсказки можно привести задачу из *разд. 5.6*.

5.6. Использование подсказок оптимизатору

В большинстве случаев оптимизатор запросов SQL-сервера правильно оценивает запрос и составляет наиболее оптимальный план его выполнения. Но бывают ситуации, когда возможно повысить эффективность выполнения запроса путем использования так называемых *подсказок оптимизатору*, иначе называемых *хинтами* (от англ. *hint — подсказка*), с помощью которых можно внести определенные изменения в план выполнения запроса, сгенерированный оптимизатором.

Базовая рекомендация Microsoft по поводу использования подсказок оптимизатору заключается в том, чтобы оставить все как есть, предоставив оптимизатору запросов автоматически генерировать планы выполнения. Применять подсказки рекомендуется только опытным программистам, понимающим суть алгоритмов, стоящих за этими подсказками, после того как такие меры повышения эффективности запроса, как модификация статистики таблиц, перетрансляция хранимых процедур для внутренних запросов, добавление или изменение индексов и пересмотр записи выражений для поиска, не дали ощутимого результата.

В этом случае, если идентифицирован запрос, эффективность выполнения которого критична и недостаточна, имеет смысл попробовать использование подсказок оптимизатору.

Подсказки оптимизатору могут быть разделены на пять различных категорий.

- *Подсказки таблицы* (индексные подсказки), которые служат для принудительного использования выбранного индекса.

- *Подсказки запроса*, предназначенные для управления процессом выполнения запроса.
- *Подсказки соединения*, которые могут изменить способ соединения таблиц *(см. разд. 5.5)*.
- *Подсказки блокировки*, предназначенные для избежания нежелательных блокировок, о которых будет рассказано в *главе 6*.
- *Подсказки представления*, служащие для определения индексов в индексированных представлениях *(см. разд. 5.11)*.

Одна из проблем, связанная с использованием подсказок, состоит в том, что подсказка, найденная в каком-то состоянии базы данных, может со временем перестать быть полезной. Поэтому рекомендуется предусмотреть какую-то процедуру, позволяющую периодически убедиться в том, что подсказки продолжают оставаться эффективными.

В зависимости от обстоятельств имеется быстрый способ выключить все подсказки для отдельного подключения пользователя или для всего SQL-сервера в целом без изменения кода запросов путем использования средства консоли команд базы данных `DBCC` с соответствующими флагами трассировки. Например:

- `DBCC TRACEON (8602)` — эта команда используется, чтобы выключить все индексные подсказки для текущего подключения;
- `DBCC TRACEON (8755)` — эта команда используется, чтобы выключить все подсказки блокировки для текущего подключения;
- `DBCC TRACEON (8722)` — эта команда используется, чтобы выключить все остальные подсказки для текущего подключения.

Эти команды могут быть объединены в одну, например команда

```
DBCC TRACEON (8602, 8755, 8722)
```

выключает все типы подсказок для текущего подключения.

Для отключения подсказок для всего SQL-сервера необходимо добавить "–1" к любому из вышеупомянутых примеров. Так командой

```
DBCC TRACEON (8602, 8755, 8722, -1)
```

подсказки выключаются для всего сервера.

Как же можно использовать эти команды в реальном мире? Один из способов состоит в применении утилиты SQL Profiler для типичной рабочей нагрузки с сортировкой запросов по их продолжительности. Нужно получить два следа трассировки — один для режима с подсказками, а другой при отключенных подсказках, после чего можно сравнивать продолжительности различных за-

просов, чтобы видеть, имеются ли какие-либо существенные различия, и делать выводы об эффективности подсказок.

Следует отдавать себе отчет в том, что возможна ситуация, когда заданная подсказка оптимизатором реально не используется. Например, если добавлена подсказка на применение определенного индекса для таблицы, а оптимизатор нашел и использовал подходящее индексированное представление, то, естественно, подсказка по поводу индекса будет игнорироваться. В большинстве случаев если оказывается, что подсказка не используется, наиболее вероятно, что допущена ошибка и следует проверить код задания подсказки.

Далее в этом разделе приведено описание первых двух типов подсказок оптимизатору из их полного списка, а именно подсказки таблицы и подсказки запроса. Описание остальных типов подсказок вынесено в соответствующие темы, на что указывают ссылки полного списка подсказок.

5.6.1. Подсказки таблицы

Подсказки этого типа используются, чтобы определить, какие индексы нужно принудительно использовать для выполнения запроса. Синтаксис для подсказки таблицы имеет следующий вид:

```
SELECT column_list FROM table_name WITH (INDEX (index_name) [, ...])
```

Здесь `index_name` — это имя индекса или его системный идентификатор `(INDID)`, который можно узнать из системной таблицы `sysidexes`, о чем рассказывается в *главе 4*. Хотя системные идентификаторы индексов можно использовать наряду с их именами, это не очень удачная идея, т. к. эти идентификаторы могут измениться при удалении и пересоздании индексов. Если нужно использовать несколько индексов, то их имена перечисляются через запятую в списке опции `WITH`. Если нужно, наоборот, запретить использование индексов и принудительно выполнять запрос путем полного сканирования таблицы, то следует задать подсказку в виде `WITH (INDEX(0))`.

Как и другие подсказки, индексные подсказки должны использоваться только в том случае, когда оптимизатор запроса по какой-то причине не выбирает доступный индекс, а использование индекса обеспечивает увеличение эффективности работы запроса.

Примером подсказки оптимизатору, которую удалось найти на основе теории, для искусственно созданной задачи в рамках базы данных материального учета служит следующее.

Сконструирована таблица, аналогичная таблице `РАСХОДЫ_ТОВАРЫ` базы данных `МАТ_УЧЕТ`, с именем `РАСХОДЫ_1`, и для нее задан единственный индекс (листинг 5.6).

Листинг 5.6. Таблица для демонстрации подсказки оптимизатору

```
CREATE TABLE [dbo].[РАСХОДЫ_1] (
        [N_расх] [int] NOT NULL,
-- Тип данных следующего поля не совпадает с типом данных таблицы РАСХОДЫ
        [Товар] [varchar] (50) COLLATE Cyrillic_General_CI_AS NOT NULL,
        [Кол_прих] [int] NULL,
        [Дата_прих] [datetime] NULL) ON [PRIMARY]
GO
CREATE INDEX [IX_РАСХОДЫ_1_N_расх] ON [РАСХОДЫ_1]([N_расх])ON [PRIMARY]
GO
```

Никаких ограничений для таблицы `РАСХОДЫ_1` не создано. В эту таблицу путем автогенерации занесено около 160 000 записей таким образом, что в каждой расходной накладной имеется либо две, либо три товарные позиции. Селективность поля `N_расх`, вычисленная по методу листинга 5.1, равна 28 %. То есть индекс `IX_РАСХОДЫ_1_N_расх`, согласно теории, является низкоселективным. Необходимо при этих условиях найти для таблицы `РАСХОДЫ_1` расходные накладные с номером, меньше 100, в которых присутствует товар с названием "Товар2". Простой запрос, дающий ответ на вопрос задачи, выглядит так:

```
SELECT * FROM РАСХОДЫ_1 WHERE N_расх < 100 and Товар = 'Товар2'
```

Этот запрос выполняется путем полного сканирования таблицы, потому что оптимизатор не счел полезным использование низкоселективного индекса `IX_РАСХОДЫ_1_N_расх`. При этом время выполнения запроса оказалось порядка 50 мс. Добавление подсказки об использовании индекса `IX_РАСХОДЫ_1_N_расх` представлено в листинге 5.7.

Листинг 5.7. Задание индексной подсказки оптимизатору

```
-- Подсказка с использованием явного имени индекса
SELECT * FROM РАСХОДЫ_1 WITH (INDEX (IX_РАСХОДЫ_1_N_расх))
WHERE N_расх < 100 and Товар = 'Товар2'
-- Та же подсказка с использованием системного идентификатора индекса
SELECT * FROM РАСХОДЫ_1 WITH (INDEX (2))
WHERE N_расх < 100 and Товар = 'Товар2'
```

Замечание

Следует обратить внимание на то, что при задании подсказки с использованием идентификатора индекса нужно этот идентификатор определить, пользуясь системной таблицей `sysidexes`. Почему в нашем примере написано `WITH (INDEX (2))`, объясняется в *главе 4.*

Применение подсказки привело к тому, что был использован индексный поиск вместо сканирования таблицы, и время выполнения запроса уменьшилось примерно в 10 раз. Интересно, что стоимость запроса с подсказкой оказалась чуть выше, чем без нее, — поэтому, кстати, оптимизатор и не выбрал индекс самостоятельно.

Разумеется, последний рассмотренный пример искусственный, т. к. в нем используется таблица без кластерного индекса, что редко является разумным. Составной кластерный индекс, реализующий ограничение первичного ключа, как это и сделано в базе данных `МАТ_УЧЕТ` для таблицы `РАСХОДЫ_ТОВАРЫ`, оптимизатор запроса будет использовать без всяких подсказок.

Еще одна рекомендация по использованию подсказок оптимизатору состоит в следующем. Если оказалось, что SQL-сервер выполняет сканирование таблицы вместо индексного поиска для предиката `IN` или дизъюнкции `OR` в опции `WHERE` при наличии покрывающего индекса, то применение подсказки, требующей использования этого индекса, может быть эффективным. Пусть имеются два запроса, приведенные в листинге 5.8.

Листинг 5.8. Задание индексной подсказки оптимизатору

```
-- Первый запрос
SELECT * FROM tblTaskProcesses
WHERE nextprocess = 1 AND processid IN (8,32,45)
-- Второй запрос с подсказкой
SELECT * FROM tblTaskProcesses (INDEX = IX_ProcessID)
WHERE nextprocess = 1 AND processid IN (8,32,45)
```

Утверждается, что первый из этих запросов выполняется около трех секунд, в то время как второй — около одной секунды.

5.6.2. Подсказки запроса

SQL-сервер предлагает ряд различных подсказок для запросов, которые управляют широким спектром задач. Использовать эти подсказки нужно только опытному разработчику в специфических ситуациях. Подсказки запроса задаются с помощью предложения `OPTION`. В нем можно ввести несколько подсказок, но запрос имеет только одно предложение `OPTION`. Предложение `OPTION` добавляется в конце запроса, и его синтаксис таков:

```
OPTION (hint 1 [, ...hint n])
```

Здесь `hint 1` или `hint n` — это одна или более подсказок, разделенных запятыми. Далее перечисленные некоторые из возможных подсказок.

- `EXPAND VIEWS` запрещает оптимизатору запроса использовать любые индексированные представления.

- `FAST n` требует возвратить указанное число строк результата запроса с максимально возможной скоростью, даже если это снижает общую эффективность запроса, создавая тем самым иллюзию быстродействия для пользователя. Как только указанное число строк возвращено, остающиеся строки возвращаются с обычной скоростью.
- `FORCE ORDER` требует выполнять соединение таблиц в том порядке, как это указано в запросе, а не пробовать переупорядочить таблицы оптимальным образом.
- `{HASH | ORDER} GROUP` требует выполнять групповые операции `GROUP BY`, `DISTINCT` или `COMPUTE` указанным методом.
- `KEEPFIXED PLAN` сообщает оптимизатору, что не нужно перетранслировать план выполнения запроса при изменениях в статистике или в индексных столбцах. Это означает, что запрос будет перетранслирован, только если изменена схема таблиц или для таблицы явно выполнена системная процедура `sp_recompile`.
- `{LOOP | MERGE | HASH} JOIN` указывает способ соединения таблиц и может быть задано в опции `FROM`.
- `{MERGE | HASH | CONCAT} UNION` указывает способ выполнения операции `UNION`.
- `ROBUST PLAN` требует создавать план выполнения запроса, который работает для максимального размера строки, даже если это означает, что общая эффективность будет снижена. В редких случаях, когда таблицы имеют очень широкие столбцы типа `VARCHAR`, оптимизатор запроса может создавать план выполнения, который генерирует промежуточные таблицы. Если имеются любые внутренние операции, которые сохраняют и обрабатывают строки в этих таблицах, возможно, что будет выдана ошибка о превышении предела размера строки, а запрос не выполнится. При наличии подсказки `ROBUST PLAN` оптимизатор не будет генерировать никаких планов, которые могли бы потенциально вызвать эту проблему.

5.7. Хранимые процедуры и производительность SQL-сервера

Хранимая процедура — это предварительно скомпилированный набор операторов языка T-SQL, который хранится на SQL-сервере. Хранимые процедуры являются основным способом оформления часто решаемых задач, обеспечивающим их эффективное выполнение. Краткий синтаксис оператора создания хранимой процедуры с именем `procedure_name` выглядит следующим образом:

```
CREATE PROC[EDURE ] procedure_name
[ @parameter data_type [ VARYING ] [ = default ] [ OUTPUT ],...]
[ WITH { RECOMPILE | ENCRYPTION} ]
AS
операторы
[RETURN]
```

Кратко опишем назначение синтаксических конструкций оператора создания хранимой процедуры.

- `@parameter data_type` — это параметр процедуры указанного типа данных. Значение каждого объявленного параметра должно быть задано пользователем при выполнении процедуры, если значение по умолчанию `(= default)` для параметра не определено. Допустимо не более 2 100 параметров. Параметры могут быть выходными, признак этого — опция `OUTPUT`.
- `VARYING` определяет выходной параметр типа курсора.
- `RECOMPILE` указывает, что SQL-сервер не кэширует план этой процедуры, а процедура перекомпилируется каждый раз во время выполнения. Используется для оптимизации быстродействия в специфических случаях, о чем будет рассказано подробнее.
- `ENCRYPTION` указывает, что SQL-сервер должен шифровать информацию системной таблицы `syscomments`, содержащей текст инструкции `CREATE PROCEDURE`, предотвращая тем самым просмотр текста хранимой процедуры посторонними лицами.
- `RETURN` задает передаваемое вызывающему приложению значение, которым обычно является код возврата, используемый для анализа ошибок.

Основные преимущества использования хранимых процедур заключаются, во-первых, в снижении загрузки сети, а во-вторых, в использовании *буферного кэша*. Всякий раз, когда прикладная программа должна послать запрос SQL-серверу, его рекомендуется оформлять в виде хранимой процедуры, а не в виде текста на языке T-SQL, внедренного в код клиентского приложения. Такой способ программирования позволяет существенно уменьшить сетевой трафик и время ожидания, увеличивая прикладную эффективность. Когда прикладная программа выполняет хранимую процедуру, то происходит очень компактный RPC (Remote Procedure Call, удаленный вызов процедуры), посылаемый от пользователя на SQL-сервер, т. к. он включает только единственную команду вызова процедуры. Но если прикладная программа вместо хранимой процедуры использует внедренный T-SQL, то приходится пересылать по сети весь этот код, после чего его нужно каждый раз заново компилировать и оптимизировать.

Дополнительные преимущества использования хранимых процедур следующие:

- планы выполнения процедур могут многократно использоваться без перекомпиляции и повторной оптимизации запросов, оставаясь в буферном кэше SQL-сервера;
- хранимые процедуры являются средством повторного использования кода, что может увеличивать производительность разработчиков, сокращая время написания и отладки программы;
- хранимые процедуры инкапсулируют бизнес-логику на стороне SQL-сервера, позволяя вносить изменения в код на клиентской стороне без существенных затрат времени разработчика;
- хранимые процедуры обеспечивают лучшую и более гибкую защиту данных. Пользователям можно разрешить выполнение хранимой процедуры, даже если у них нет прав на доступ к объектам, с которыми эта процедура работает.

5.7.1. Управление перекомпиляцией и кэшированием

Кэширование — это процесс занесения в специально выделенное место оперативной памяти информации, которая обычно хранится в базе данных на диске. Кэшированные данные хранятся в кэше определенное время и могут быть считаны оттуда по мере необходимости гораздо быстрее, чем с диска. Для хранимых процедур характерно, что перед непосредственным их выполнением в буферный кэш заносится соответствующий план выполнения процедуры. В кэше всегда будет, по крайней мере, один скомпилированный план для каждой уникальной комбинации хранимой процедуры и среды.

Под *средой* понимают определенные параметры конфигурации подключения, задаваемые командой `SET`, например изменение языка или формата даты, которые влияют на процесс оптимизации запросов. Кроме того, в формирование понятия среды входит механизм неявного владения объектами. Очевидно, что чем меньше сгенерировано различных сред, тем лучше. Поэтому для повышения быстродействия все объекты, которые используются в пределах одной хранимой процедуры, должны принадлежать ее владельцу, которым предпочтительно является владелец базы данных `dbo`. В противном случае тратится достаточно много времени на проверку прав доступа. Удобнее всего создавать хранимую процедуру от имени владельца базы данных и назначать другим пользователям права на ее выполнение везде, где это возможно. Имена объектов при написании хранимой процедуры лучше задавать в полном формате, а именно `object_owner.object_name`. Параметры соединения с SQL-

сервером должны быть согласованы, установлены в начале подключения и впоследствии не изменяться.

Что же делать, если конкретная хранимая процедура все-таки работает медленно? Первое мероприятие, которое нужно предпринять, — это принудительная перекомпиляция хранимой процедуры, потому что возможно, что найденный в кэше план ее выполнения не соответствует изменившейся ситуации. Перекомпиляцию можно осуществить с помощью системной процедуры `sp_recompile` или же добавив опцию `WITH RECOMPILE` при выполнении процедуры.

Если перекомпиляция помогла, то возможны две ситуации. В базе данных могли произойти изменения, которые влияют на план выполнения процедуры, например, добавлены индексы, полезные для выполнения запросов из этой хранимой процедуры. Перекомпиляция привела к тому, что теперь эти индексы стали использоваться, и быстродействие возросло. Это наиболее легкая ситуация, потому что делать ничего больше не нужно, — осталось только убедиться, что при дальнейшей работе снижения быстродействия не произойдет. Если это не так, т. е. имеет место ситуация, когда принудительная перекомпиляция лишь временно позволит уменьшить время выполнения процедуры, а потом оно снова возрастает, то такая хранимая процедура является кандидатом на создание ее с опцией `WITH RECOMPILE`. Если же перекомпиляция процедуры не помогает вообще, то не остается ничего другого, как анализировать ее код с целью его оптимизации.

Но возможна и противоположная ситуация. Если хранимая процедура слишком часто, по вашему мнению, перекомпилируется, то это также может приводить к проблеме снижения быстродействия. В этом случае, конечно, план выполнения запросов процедуры будет находиться все время в более или менее оптимальном состоянии, но затраты на его поддержание могут перевесить выгоду от его использования.

Поскольку неадекватная перекомпиляция может служить серьезной причиной понижения быстродействия при использовании хранимых процедур, то нужно отчетливо понимать, в каких случаях SQL-сервер автоматически перекомпилирует хранимую процедуру. Это происходит в следующих случаях:

- хранимая процедура создана или запущена на выполнение с опцией `WITH RECOMPILE` — такие процедуры вообще не кэшируются;
- выполняется команда `sp_recompile` для любой таблицы или представления, упомянутых в хранимой процедуре;
- произошло изменение схемы для любого объекта, упомянутого в хранимой процедуре, включая изменение правил, значений по умолчанию или ограничений базы данных;

- сгенерирована новая дистрибутивная статистика для любого объекта, упомянутого в хранимой процедуре;
- база данных, которая включает хранимую процедуру или любой из объектов, на который ссылается хранимая процедура, подвергалась процессу восстановления;
- план выполнения хранимой процедуры был удален из буферного кэша SQL-сервера по причине превышения времени его хранения;
- индекс, используемый планом выполнения хранимой процедуры, был удален;
- число операций модификации данных для таблицы, упомянутой в хранимой процедуре, оказалось большим;
- хранимая процедура включает как `DDL`, так и `DML` инструкции, и они взаимодействуют друг с другом;
- хранимая процедура использует временные таблицы.

Не существует прямой возможности запретить выполнение автоматической перекомпиляции хранимых процедур. Можно только определенными косвенными действиями повлиять на ситуации, которые к ней приводят. Так, можно отключить на перегруженном SQL-сервере опцию **Auto Update Statistics**. Это ведет ко многочисленным сложным последствиям *(см. разд. 5.2)*, но перекомпиляции, вызванные обновлением статистики, происходить перестанут. Использование `sp_executesql` вместо `EXECUTE` также может улучшить ситуацию (объяснение приведено в BOL). Не стоит смешивать `DDL` и `DML` инструкции в одной и той же хранимой процедуре, если это возможно.

При использовании временных таблиц можно применить подсказку оптимизатору `KEEP PLAN`, использование которой, впрочем, полезно, только если временные таблицы редко подвержены изменениям. Кроме того, при использовании временных таблиц полезно придерживаться следующих правил:

- любые ссылки на временные таблицы в хранимой процедуре должны относиться только к таблицам, созданным в ней же без использования команд `sp_executesql` или `EXECUTE`;
- все операторы работы с временными таблицами должны быть расположены между операторами `CREATE TABLE` и `DROP TABLE` для этой временной таблицы;
- не нужно объявлять никаких курсоров, которые обращаются к временной таблице.

Как уже было сказано, перед выполнением запросы, включенные в код хранимой процедуры, анализируются оптимизатором с целью построения оптимального плана их выполнения. Оптимизированные планы выполнения за-

просов являются составными частями откомпилированного плана выполнения всей процедуры, который кэшируется в буфере SQL-сервера.

Планы выполнения из буферного кэша доступны только для чтения и могут использоваться любым количеством пользователей. Правда, для каждого пользователя при этом инициализируется дополнительная область памяти, называемая *контекстом выполнения*, которая содержит, например, значения параметров процедуры, связанные с этим пользователем. SQL-сервер может выбирать из буферного кэша планы выполнения отдельных запросов, если они там уже есть. С этой точки зрения удобно, если при каждом выполнении хранимой процедуры запросы в ней не меняются. Однако управляющие структуры T-SQL могут приводить к тому, что при разных значениях входных параметров будут выполняться совершенно разные запросы. Кроме того, запрос внутри хранимой процедуры может оказаться динамическим, и если при его конструировании использованы входные параметры процедуры, то этот запрос в принципе не может быть оптимизирован один раз, поэтому вся хранимая процедура будет перекомпилироваться постоянно, а это очень снижает производительность.

Рассмотрим сначала примеры хранимых процедур, порождающих эту проблему, а потом дадим объяснения способов ее преодоления. В листинге 5.9 приведены две простейшие хранимые процедуры, одна из которых содержит управляющую структуру, а другая — динамический SQL.

Листинг 5.9. Хранимые процедуры

```
-- Процедура с динамическим SQL
CREATE PROCEDURE dbo.Test1 @par1 VARCHAR(10) AS
DECLARE @text VARCHAR(100)
SET @text='SELECT * FROM '+@par1
EXEC(@text)
GO
-- Процедура с управляющей структурой
CREATE PROCEDURE dbo.Test2 (@query bit) AS
IF @query = 0
SELECT * FROM Table1
ELSE
SELECT * FROM Table2
GO
```

Вызовем процедуру `Test2` с параметром `@query = 0`. В буферный кэш заносится план выполнения процедуры с запросом `SELECT * FROM Table1`, который совершенно не годится для выполнения той же самой хранимой процедуры с параметром `@query= 1`, т. к. в этом случае нужно выполнять совершенно дру-

гой запрос. В результате SQL-сервер вынужден создать и оптимизировать план выполнения второго запроса с самого начала. Суть возникающей проблема состоит в том, что SQL-сервер генерирует и кэширует план выполнения хранимой процедуры только для того пути управляющих структур T-SQL, который имел место при первом выполнении процедуры, а не для всех возможных путей.

Рекомендуемый в данном случае совет заключается в том, чтобы изменить код хранимой процедуры `Test2`, разбив его на две процедуры так, как это сделано в листинге 5.10.

Листинг 5.10. Разбивка кода хранимой процедуры `Test2`

```
-- Отдельная процедура для первого запроса
CREATE PROCEDURE dbo.Test2_Table1 AS
SELECT * FROM Table1
GO
-- Отдельная процедура для второго запроса
CREATE PROCEDURE dbo.Test2_Table2 AS
SELECT * FROM Table2
GO
-- Вызывающая процедура
CREATE PROCEDURE dbo.Test_Call (@query bit) AS
IF @query = 0
EXEC Test1
ELSE
EXEC Test2
GO
```

Результат этой реструктуризации приведет к тому, что в буферном кэше будут одновременно храниться оптимизированные планы обеих операторов `SELECT`, т. к. они разнесены в разные хранимые процедуры. Конечно, данный пример является только иллюстрацией — ведь затраты на перекомпилирование такого короткого запроса, какой использован в хранимой процедуре `Test2`, не будут особенно заметными. Имеются сообщения о том, что используя эту методику для сложных запросов в задаче, реализующей элементы полнотекстового поиска для генератора отчетов, удалось добиться примерно десятикратного (!) увеличения эффективности.

Единственным недостатком этого метода является то, что после его использования нужно управлять тремя хранимыми процедурами вместо одной. Это не очень затруднительно, поскольку `Test_Call` является единственной хранимой процедурой, непосредственно вызываемой пользователем, поэтому фактически управлять правами доступа нужно только для нее.

5.7.2. Другие рекомендации по оптимизации хранимых процедур

По умолчанию каждый раз при выполнении хранимой процедуры SQL-сервер посылает клиенту сообщение, указывающее число строк, которые обработаны хранимой процедурой. Эта информация редко полезна для клиента, и ее можно не пересылать по сети без надобности. Отключить ее на уровне соединения можно командой `SET NOCOUNT ON`, которую также допускается вставлять в начало каждой пользовательской хранимой процедуры.

Для всего SQL-сервера сообщение о количестве обработанных строк отключается командой

```
SP_CONFIGURE 'user options', 512
RECONFIGURE
```

Если создается хранимая процедура, которую не предполагается сохранять в базе данных `MASTER`, то не нужно использовать префикс "`sp_`" в ее названии, т. к. этот специальный префикс предназначен для системных хранимых процедур и поиск такой процедуры начинается в системной базе данных `MASTER`, а не в текущей. На этот поиск впустую тратится время.

Когда нужно выполнять строку кода T-SQL многократно или использовать динамический SQL, то предпочтительно применять хранимую процедуру `sp_executesql` вместо инструкции `EXECUTE`. Сначала рассмотрим разницу в синтаксисе этих двух способов вызова процедуры на примере простейшей задачи динамического SQL, которая для двух параметров определяет, имеется ли в заданной таблице (параметр `@table_name`) заданный товар (параметр `@par1`) в столбце `Товар` (листинг 5.11).

Листинг 5.11. Использование `sp_executesql` и `EXECUTE`

```
-- Объявляем переменные
DECLARE @par1 VARCHAR(10), @table_name sysname
-- Строки для формирования динамического оператора SQL
DECLARE @sql_text1 VARCHAR(50),@sql_text2 NVARCHAR(50)
-- Использование EXECUTE
SET @par1='Товар 19'
SET @table_name='ТОВАРЫ'
SET @sql_text1='SELECT * FROM '+@table_name+ '
WHERE Товар = ''' +@par1+''''
-- Вывод сформированной строки динамического SQL
SELECT @sql_text1
```

```
-- Выполнение
EXEC(@sql_text1)
-- Использование sp_executesql — приведено два способа вызова
SET @sql_text2='SELECT * FROM '+@table_name+ ' WHERE Товар = @par1'
-- Первый способ
EXEC sp_executesql @sql_text2,N'@table_name sysname,
@par1 VARCHAR(10)', @table_name=N'ТОВАРЫ',@par1=N'Товар 19'
-- Второй способ
EXEC sp_executesql @sql_text2,N'@table_name sysname,@par1 VARCHAR(10)',
N'ТОВАРЫ',N'Товар 19'
```

Из листинга 5.11 видно, что основная разница между `sp_executesql` и `EXECUTE` состоит в том, что для `sp_executesql` допустимы параметры при создании динамической строки SQL. Это различие приводит к следующим преимуществам. Во-первых, поддержка замены параметров допускает более гибкое программирование. Во-вторых, и это главное, для повышения эффективности процедура `sp_executesql` создает планы выполнения запроса, которые с большей вероятностью будут повторно использоваться SQL-сервером, потому что наличие параметров в случае `sp_executesql` позволяет не создавать заново всю строку динамического оператора SQL.

В случае если хранимая процедура должна возвращать единственное значение, а не набор данных, рассматривается возможность передачи его клиенту с помощью опции `RETURN`. Хотя традиционно эта опция используется для проверки кода ошибки, но это не является обязательным требованием, и если `RETURN` использовать как обычный выходной параметр, это будет быстрее.

5.7.3. Пример использования утилиты SQL Profiler для мониторинга хранимых процедур

Описание использования утилиты SQL Profiler в плане ее общих возможностей приведено в *приложении 1.* Без понимания того, как пользоваться этой утилитой, разобраться в примере из данного раздела достаточно сложно.

Хранимые процедуры являются наглядным объектом для демонстрации возможностей утилиты SQL Profiler, поэтому и приведен пример по трассировке вызовов хранимой процедуры, содержащей простейшие операторы динамического SQL, аналогичные листингу 5.10.

На основании таблицы следа трассировки под названием `trace_book` создан запрос, который расшифровывает коды событий трассировки и представляет ее результаты в более компактном и наглядном виде. Таблица `Events_Id` *(см. приложение 1)*, использованная в этом запросе, содержит расшифровку кодов событий следа трассировки. Текст запроса приведен в листинге 5.12.

Таблица 5.5. *Трассировка последовательности вызовов хранимой процедуры*

RowNumber	EventClass	Комментарии	Description	TextData	id	name	type	reads	CPU	Duration
Создана и первый раз выполнена процедура `Test_Dyn`										
3	34		SP:CacheMiss							
4	34		SP:CacheMiss							
5	35	В кэш вставлен план выполнения процедуры `Test_Dyn`	SP:Cache Insert		942626401	`Test_Dyn`	P			
6	42	`Test_Dyn` стартовала	SP: Starting	`Test_Dyn 'R1_1'`	942626401	`Test_Dyn`	P			
7	44		SP:Stmt Starting	`-- Test_Dyn @text='SELECT * FROM ' +@par1`	942626401	`Test_Dyn`	P			
8	45		SP:Stmt Completed	`-- Test_Dyn @text='SELECT * FROM '+@par1`	942626401	`Test_Dyn`	P	2	0	0
9	44		SP:Stmt Starting	`-- Test_Dyn EXEC(@text)`	942626401	`Test_Dyn`	P			
10	35	В кэш вставлен план выполнения запроса	SP:Cache Insert	`()SELECT * FROM [R1_1]`	78426227					

Таблица 5.5 *(продолжение)*

RowNumber	EventClass	Комментарии	Description	TextData	id	name	type	reads	CPU	Duration
11	44	Запрос начал выполняться	SP:Stmt Starting	`-- Dynamic SQL SELECT * FROM R1_1`	21489264					
12	38	Процедура найдена в кэше и план запроса тоже	SP:Cache Hit		78426227					
13	45	Запрос выполнился	SP:Stmt Completed	`-- Dynamic SQL SELECT * FROM R1_1`	21489264			1290	421	5037
14	45	Оператор завершился	SP:Stmt Completed	`-- Test_Dyn EXEC(@text)`	942626401	`Test_Dyn`	P	1303	421	5037
15	43	`Test_Dyn` завершилась	SP:Completed	`Test_Dyn 'R1_1'`	942626401	`Test_Dyn`	P			5037
На выполнение послан запрос `SELECT * FROM R2_1`										
16	35	Запрос кэшируется	SP:Cache Insert	`()SELECT * FROM [R2_1]`	887763486					
17	35		SP:Cache Insert	`SELECT * FROM R2_1`	702168809					
18	38		SP:Cache Hit		887763486					

Таблица 5.5 *(продолжение)*

RowNumber	EventClass	Комментарии	Description	TextData	id	name	type	reads	CPU	Duration
На выполнение послана процедура `Test_Dyn` *с параметром* `'inserted'` *. В этом случае в теле процедуры будет совершенно новый запрос, который еще не кэширован*										
19	35	План выполнения процедуры кэшируется	SP:Cache Insert	`Test_Dyn 'inserted'`						
20	39	В кэше найден другой план хранимой процедуры для другого контекста	SP:Exec ContextHit		942626401	`Test_Dyn`	P			
21	42		SP:Starting	`Test_Dyn 'inserted'`	942626401	`Test_Dyn`	P			
22	44		SP:Stmt Starting	`-- Test_Dyn @text='SELECT * FROM '+@par1`	942626401	`Test_Dyn`	P			
23	45		SP:Stmt Completed	`-- Test_Dyn @text='SELECT * FROM '+@par1`	942626401	`Test_Dyn`	P	2	0	0
24	44		SP:Stmt Starting	`-- Test_Dyn EXEC(@text)`	942626401	`Test_Dyn`	P			
25	35	Кэшируется запрос	SP:Cache Insert	`()SELECT * FROM [inserted]`	875636990					

Таблица 5.5 *(продолжение)*

RowNumber	EventClass	Комментарии	Description	TextData	id	name	type	reads	CPU	Duration
26	44		SP:Stmt Starting	-- Dynamic SQL SELECT * FROM inserted	498590061					
27	38		SP:Cache Hit		875636990					
28	45		SP:Stmt Completed	-- Dynamic SQL SELECT * FROM inserted	498590061			15	0	0
29	45		SP:Stmt Completed	-- Test_Dyn EXEC(@text)	942626401	Test_Dyn	P	28	0	0
30	43	Процедура завершена	SP:Completed	Test_Dyn 'inserted'	942626401	Test_Dyn	P			0
*На выполнение снова послан запрос "SELECT * FROM R2_1", он уже есть в кэше, т. к. выполнялся (см. строку следа с номером 16). Запрос найден в кэше, событие SP:CacheInsert не возникает*										
31	38	Запрос и контекст найдены в кэше	SP:Cache Hit		702168809					
32	39		SP:ExecContext Hit		887763486					
*На выполнение первый раз послан запрос "SELECT * FROM tr1"*										
33	35	Запрос кэшируется	SP:Cache Insert	()SELECT * FROM [tr1]	694263914					

Таблица 5.5 *(продолжение)*

RowNumber	EventClass	Комментарии	Description	TextData	id	name	type	reads	CPU	Duration
На выполнение послана процедура `Test_Dyn` с параметром `'tr1'` . В этом случае в теле процедуры возникнет запрос, который уже кэширован, SQL-сервер найдет план выполнения процедуры с подходящим контекстом и воспользуется им										
34	38	В кэше найден другой план хранимой процедуры для другого контекста	SP:Cache Hit							
35	39		SP:Exec ContextHit		942626401	`Test_Dyn`	P			
36	42		SP:Starting	`Test_Dyn 'tr1'`	942626401	`Test_Dyn`	P			
37	44		SP:Stmt Starting	`-- Test_Dyn @text='SELECT * FROM '+@par1`	942626401	`Test_Dyn`	P			
38	45		SP:Stmt Completed	`-- Test_Dyn @text='SELECT * FROM '+@par1`	942626401	`Test_Dyn`	P	2	0	0
39	44		SP:Stmt Starting	`-- Test_Dyn EXEC(@text)`	942626401	`Test_Dyn`	P			
40	44		SP:Stmt Starting	`-- Dynamic SQL SELECT * FROM tr1`	314541204					

Таблица 5.5 *(окончание)*

RowNumber	EventClass	Комментарии	Description	TextData	id	name	type	reads	CPU	Duration
41	39		SP:Exec ContextHit		694263914					
42	45		SP:Stmt Completed	`-- Dynamic SQL SELECT * FROM tr1`	314541204			915	0	50
43	45		SP:Stmt Completed	`-- Test_Dyn EXEC(@text)`	942626401	`Test_Dyn`	P	919	0	50
44	43	Процедура завершилась	SP:Completed	`Test_Dyn 'tr1'`	942626401	`Test_Dyn`	P			50

Листинг 5.12. Работа с таблицей следа трассировки событий

```
SELECT RowNumber,t.EventClass,Description,TextData,
t.objectid,o.name,o.type,reads,CPU,Duration
FROM (trace_book t left join sysobjects o on t.objected = o.id)
INNER JOIN Events_Id e on t.EventClass = e.EventClass
ORDER BY RowNumber
```

После выделения необходимых данных, связанных с работой интересующей нас хранимой процедуры с именем `Test_Dyn`, результаты работы запроса преобразованы в таблицу в текстовом формате, а затем в нее вручную вставлены комментарии с описанием тех действий, которые породили возникновение соответствующих событий в следе. Последняя таблица представлена в табл. 5.5.

Из данного следа трассировки отчетливо видно, что SQL-сервер, занося в кэш план выполнения запроса, присваивает ему объектный идентификатор, которого, конечно же, нет в системной таблице `sysobjects`. По этому идентификатору SQL-сервер может находить информацию в кэше и повторно ее использовать.

Изучать табл. 5.5 подробно саму по себе нет необходимости. Она приведена только как иллюстрация возможности получения подробной информации о процессе многократного выполнения и кэширования хранимых процедур.

5.8. Автогенерация данных

Оценка производительности выполнения запросов как с помощью значения параметра стоимости, так и способом прямого замера времени на стадии разработки приложения редко может быть адекватной. Основная причина этого состоит в том, что у разработчика отсутствуют реальные данные, которые накопятся в базе данных возможно только через месяцы, а то и годы ее эксплуатации. Поэтому выполнять создание и настройку индексов можно только на основе теоретических рекомендаций. На таблицах небольшого размера мастер настройки индексов вообще не выдает никаких подсказок — запросы хоть с индексами, хоть без индексов выполняются очень быстро, разница не ощущается, и не за что особенно бороться.

Для практической оценки быстродействия нужны тестовые таблицы с большим объемом данных. На стадии разработки приложения данные для этих таблиц можно генерировать автоматически с использованием недетерминированных скалярных функций. К этим функциям относятся, например, `RAND()`, `GETDATE()` и другие функции, значения которых не повторяются при каждом вызове. Основная идея такой автогенерации состоит в использовании цикли-

чески выполняемого оператора вставки INSERT, в списке VALUES которого присутствуют выражения, содержащие недетерминированные функции. Для получения наиболее достоверных результатов тестирования при задании этих выражений нужно постараться, чтобы селективность автоматически сгенерированных значений в полях таблиц была приближена к реально ожидаемой.

Речь здесь идет вот о чем. Предположим, что необходимо исследовать быстродействие запроса, в котором участвует таблица с миллионами записей, имеющая неключевое поле Товар_Ид, содержащее код товара. В запросе по этому полю предполагается выполнять соединение таблиц с последующей группировкой для вычисления итогов. Для достижения адекватной селективности нужно, чтобы в тестовой таблице в поле Товар_Ид оказалось k различных значений кодов товаров. Напомним, что величина k носит название кардинальности. Причем это значение k должно примерно соответствовать действительно ожидаемой кардинальности выборки по атрибуту. То есть если предполагается, что значений кодов товаров будет около сотни ($k = 100$), то эту сотню значений с повторениями и нужно распределить по миллиону записей. Далее приведены листинги программ, которые реализуют автогенерацию данных для *целевых таблиц* с заданным числом записей. Под термином *целевые таблицы* подразумеваются те таблицы, в которые выполняется автогенерация данных. В данном разделе все целевые таблицы не являются таблицами основного проектного решения для базы данных МАТ_УЧЕТ, описанного в самом начале текущей главы. Рассмотрены наиболее типичные примеры ограничений целостности данных.

5.8.1. Пример 1. Одна таблица с автоинкрементным ключом

Будем создавать таблицу с числовым автоинкрементным первичным ключом, распределяя в ней заданное количество кодов товаров по произвольному числу записей. Это можно сделать, используя функцию RAND(), которая при вызове без аргумента выдает действительное случайное число в диапазоне от 0 до 1. Если результат вызова функции RAND() умножить на $k + 1$ и присвоить целочисленной переменной, то получатся как раз те значения, которые можно использовать для генерации последовательности кодов товаров.

Описанный прием позволяет без труда генерировать не только числовые, но строковые последовательности с нужной степенью селективности для полей соответствующего типа. Чтобы это проиллюстрировать, в первом примере для поля Товар_Ид специально выбран строковый тип данных, хотя это и отличается от общепринятых проектных решений. Данные, сгенерированные подобным образом, не будут иметь смысла — ведь строки создаются на основе цифр, но для тестирования запросов они вполне пригодны. Далее приве-

ден листинг 5.13, с помощью которого можно сгенерировать заданное количество записей для целевой таблицы ПРИХОДЫ_ПРИМЕР.

Листинг 5.13. Автогенерация данных для одной таблицы с автоинкрементным ключом

```
-- Создание целевой таблицы
if exists (SELECT * FROM dbo.sysobjects WHERE
id = object_id(N'[dbo].[ПРИХОДЫ_ПРИМЕР]') and
OBJECTPROPERTY(id, N'IsUserTable') = 1)
DROP TABLE [dbo].[ПРИХОДЫ_ПРИМЕР]
CREATE TABLE [dbo].[ПРИХОДЫ_ПРИМЕР] (
        [N_прих] [INT] IDENTITY (1, 1) NOT NULL PRIMARY KEY CLUSTERED,
        [Товар_Ид] [varchar](15) NOT NULL,
        [Кол_прих] [INT] NOT NULL,
        [Дата_прих] [DATETIME] NOT NULL
) ON [PRIMARY]
GO
-- Объявляем переменные
DECLARE @Counter INT, @End INT, @p INT, @k INT,
@f_Товар varchar(100), @f_Кол_прих INT, @f_Дата_прих DATETIME
-- Задаем кардинальность выборки по атрибуту Товар,
-- т. е. 8 кодов товаров будут распределяться по записям таблицы
SET @k=8
-- Задаем начальное значение счетчика цикла
SELECT @Counter=1
 -- Задаем количество вставляемых записей
SELECT @End=33
-- Основной цикл
       WHILE(@Counter <= @End)
       BEGIN
-- Генерация переменных, содержащих значения полей записи целевой таблицы
       SELECT @f_Кол_прих = 100 * RAND()+10,
       @f_Дата_прих = GetDate() + 10 * RAND(),
       @f_Товар='Тест'+Convert(varchar(100), Convert(INT, (@k+1)*RAND()))
-- Вставка записи в целевую таблицу
       INSERT INTO ПРИХОДЫ_ПРИМЕР(Товар_Ид, Кол_прих, Дата_прих)
       VALUES (@f_Товар, @f_Кол_прих, @f_Дата_прих)
       SET @Counter = @Counter + 1
END
```

Замечание

Перед автогенерацией данных бывает полезно по возможности удалить ограничения и индексы целевых таблиц, потому что вставка записей в таблицы в

этом случае выполняется быстрее. После заполнения целевой таблицы данными можно создать необходимые индексы и ограничения. Разумеется, при этом все равно нужно разрабатывать код программы таким образом, чтобы на сгенерированных с ее помощью записях задавались все логически необходимые ограничения целостности данных. Для листинга 5.13 единственное такое ограничение — это ограничение уникальности поля N_прих, выполнение которого достигается заданием свойства счетчика (IDENTITY) для данного поля.

Если данные, которые нужно заносить в целевую таблицу сервера, имеются в текстовом формате, то самым быстрым способом их загрузки будет выполнение команды массового копирования BULK INSERT. Тестовый пример выполнялся следующим образом. Сначала из таблицы ПРИХОДЫ с помощью задачи экспорта данных мастера DTS (Data Transformation Services, службы преобразования данных) был получен текстовый файл ПРИХОДЫ.TXT, содержащий 10 000 записей. Для простоты, чтобы не использовать средства управления форматированием команды BULK INSERT, полученный текстовый файл был отформатирован текстовым редактором. Правда, нужно иметь в виду, что текстовый редактор для работы с подобными данными должен быть достаточно мощным и обладать развитыми средствами макропрограммирования — попробуйте без этих средств вручную хотя бы убрать лишние пробелы из 10 000 строк текста. В нашем случае был выбран текстовый редактор MultiEdit, и с его помощью данные были отформатированы под простейший вид команды BULK INSERT, когда строка текста соответствует записи целевой таблицы, а столбцы отделяются друг от друга одним пробелом. Команда массового копирования (листинг 5.14) выполнила вставку 10 000 записей в целевую таблицу ПРИХОДЫ_BU_INS меньше, чем за 300 мс.

Листинг 5.14. Простейшая автогенерация данных с помощью BULK INSERT

```
-- Создание целевой таблицы
if exists (SELECT * FROM dbo.sysobjects
WHERE id = object_id(N'[dbo].[ПРИХОДЫ_BU_INS]')
and OBJECTPROPERTY(id, N'IsUserTable') = 1)
DROP TABLE [dbo].[ПРИХОДЫ_BU_INS]
GO
CREATE TABLE [dbo].[ПРИХОДЫ_BU_INS]([N_прих] [INT] NOT NULL,
                                    [Товар_Ид] [varchar](15) NOT NULL,
                                    [Кол_прих] [float] NOT NULL,
                                    [Дата_прих] [DATETIME] NOT NULL
) ON [PRIMARY]
GO
BULK INSERT ПРИХОДЫ_BU_INS
FROM 'D:\ПРИХОДЫ.TXT' WITH (FIELDTERMINATOR=' ', CODEPAGE = 'ACP')
```

Замечание

Команда BULK INSERT не выполняет никаких проверок целостности данных, и это одна из основных предпосылок того, что массовое копирование является наиболее быстрым способом закачки данных на сервер. Поэтому при создании целевой таблицы для BULK INSERT никаких ограничений целостности данных задавать не требуется. Текстовый файл для закачки в целевую таблицу обычно готовится клиентским приложением, в задачу которого как раз и входит контролировать целостность данных.

5.8.2. Пример 2. Пара таблиц, связанных FK-ограничением

Если генерация данных для автоматического заполнения таблиц на основе случайных или недетерминированных значений по каким-либо причинам неудобна, а селективность значений в столбце невелика, то можно ввести нужные данные без повторений в отдельную небольшую таблицу, а потом из нее заполнить большую тестовую таблицу. При этом данные в маленькой и большой таблице окажутся связанными ограничением внешнего ключа по соответствующему столбцу, что позволит тестировать запросы с учетом этого ограничения, как это и имеет место в реальной ситуации. Проиллюстрируем сказанное на примере, в котором имеется пара таблиц, связанных ограничением внешнего ключа, как показано на рис. 5.8.

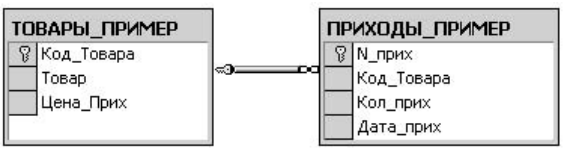


Рис. 5.8. Пара таблиц, связанных FK-ограничением

Пусть в поле первичного ключа Код_Товара таблицы ТОВАРЫ_ПРИМЕР предполагается иметь k различных значений, и они уже введены. Нужно автоматически заполнить поле Код_Товара таблицы ПРИХОДЫ_ПРИМЕР значениями из одноименного поля таблицы ТОВАРЫ_ПРИМЕР, распределив их произвольным образом более или менее равномерно на любое количество записей (в наших обозначениях на @End записей). Взяв за основу предыдущую программу, нужно добиться того, чтобы при каждом выполнении цикла переменная @f_Товар принимала значения, случайным образом выбираемые из поля Код_Товара таблицы ТОВАРЫ_ПРИМЕР. Для этого на основе таблицы ТОВАРЫ_ПРИМЕР организуется курсор с опцией SCROLL, допускающий произвольную адресацию (FETCH ABSOLUTE). В цикле происходит многократная выборка данных из курсора в переменную @f_Товар по случайно задаваемой позиции @POS. Ограничение

уникальности поля N_прих, как и в предыдущем примере, достигается использованием свойства IDENTITY. Ограничения и ключи для таблиц перед автогенерацией данных не создаются (листинг 5.15).

Листинг 5.15. Автогенерация данных для FK-ограничения с использованием курсора

```
-- Создание целевой дочерней таблицы, содержащей внешний ключ
if exists (SELECT * FROM dbo.sysobjects
WHERE id = object_id(N'[dbo].[ПРИХОДЫ_ПРИМЕР]') and
OBJECTPROPERTY(id, N'IsUserTable') = 1)
DROP TABLE [dbo].[ПРИХОДЫ_ПРИМЕР]
GO
CREATE TABLE [dbo].[ПРИХОДЫ_ПРИМЕР] (
        [N_прих] [INT] IDENTITY (1, 1) NOT NULL,
        [Код_Товара] INT NULL,
        [Кол_прих] [INT] NULL,
        [Дата_прих] [DATETIME] NULL
) ON [PRIMARY]
GO
/* Создание родительской таблицы и копирование в нее данных
 с целью имитации ввода данных в родительскую таблицу */
if exists (SELECT * FROM dbo.sysobjects
WHERE id = object_id(N'[dbo].[ТОВАРЫ_ПРИМЕР]') and
OBJECTPROPERTY(id, N'IsUserTable') = 1)
DROP TABLE [dbo].[ТОВАРЫ_ПРИМЕР]
GO
SELECT Товар_Ид AS Код_Товара, Товар, Цена_Учетн AS Цена_Прих
INTO ТОВАРЫ_ПРИМЕР FROM ТОВАРЫ
GO
-- Объявляем переменные
DECLARE @Counter INT, @End INT, @Кол_Тов INT, @POS INT,
@f_Товар varchar(15), @f_Кол_прих INT, @f_Дата_прих DATETIME,
@f_Код_Товара INT
-- Вычисляем кардинальность выборки по атрибуту Товар
SELECT @Кол_Тов = count(*) FROM ТОВАРЫ_ПРИМЕР
-- Задаем начальное значение счетчика цикла
SELECT @Counter=1
-- Задаем количество вставляемых записей
SELECT @End=33
-- Создание курсора
DECLARE ТОВАРЫ_ПРИМЕР SCROLL CURSOR FOR
SELECT Код_Товара FROM ТОВАРЫ_ПРИМЕР
```

```
-- Основной цикл
WHILE(@Counter <= @End)
BEGIN
-- Случайная выборка строки из курсора
              OPEN ТОВАРЫ_ПРИМЕР
              SELECT @POS = RAND()*(@Кол_Тов)+1
              FETCH ABSOLUTE @POS FROM ТОВАРЫ_ПРИМЕР INTO @f_Код_Товара
              SELECT @@FETCH_STATUS
              CLOSE ТОВАРЫ_ПРИМЕР
SELECT @f_Кол_прих = 100 * RAND() + 10,
       @f_Дата_прих = GetDate() + 10 * RAND()
-- вставка записи в целевую таблицу
INSERT INTO ПРИХОДЫ_ПРИМЕР(Код_Товара, Кол_прих, Дата_прих)
VALUES (@f_Код_Товара, @f_Кол_прих, @f_Дата_прих)
SET @Counter = @Counter + 1
END
DEALLOCATE ТОВАРЫ_ПРИМЕР
-- подсчет количества записей в целевой таблице
SELECT count(*) as ВСТАВЛЕНО_ЗАПИСЕЙ FROM ПРИХОДЫ_ПРИМЕР
```

Замечание

Выражение для вычисления переменной `@POS` написано так, что будут выбираться все записи курсора, но если значение `RAND()` окажется равным единице, то `@POS` выйдет за пределы диапазона записей в курсоре. Это нестрашно — выборки из курсора в этом случае не происходит, о чем сигнализирует значение переменной `@@FETCH_STATUS`, равное `–1`, а значение переменной `@f_Код_Товара` просто останется старым, и вставка записи в таблицу `ПРИХОДЫ_ПРИМЕР` все равно произойдет благополучно.

Если нужны даты без времени (со значением времени равным 00:00:00), то для сгенерированных данных можно выполнить команду, которая усекает переменную типа `DATETIME`, удаляя из нее символы, указывающие время, например:

```
UPDATE ПРИХОДЫ_ПРИМЕР SET
Дата_прих = convert(DATETIME, left(convert(varchar(50), Дата_прих), 11))
```

Разумеется, усечение даты можно сразу выполнять при генерации значения переменной `@f_Дата_прих`.

После выполнения программы из листинга 5.15 можно создать ограничения целостности данных для обеих таблиц согласно рис. 5.8. Это осуществляется, например, с помощью операторов, которые могут считаться продолжением листинга 5.15.

Листинг 5.16. Создание ограничений (окончание)

```
ALTER TABLE [dbo].[ТОВАРЫ_ПРИМЕР] WITH NOCHECK ADD
       CONSTRAINT [PK_ТОВАРЫ_ПРИМЕР] PRIMARY KEY CLUSTERED
       ([Код_Товара]) ON [PRIMARY]
GO
ALTER TABLE [dbo].[ПРИХОДЫ_ПРИМЕР] WITH NOCHECK ADD
       CONSTRAINT [PK_ПРИХОДЫ_ПРИМЕР] PRIMARY KEY CLUSTERED
       ([N_прих]) ON [PRIMARY]
GO
ALTER TABLE [dbo].[ПРИХОДЫ_ПРИМЕР] ADD
       CONSTRAINT [FK_ПРИХОДЫ_ПРИМЕР_ТОВАРЫ_ПРИМЕР] FOREIGN KEY
([Код_Товара]) REFERENCES [dbo].[ТОВАРЫ_ПРИМЕР] ( [Код_Товара])
ON DELETE CASCADE ON UPDATE CASCADE
GO
```

Ясно, что код из листинга 5.16 работает медленно, ведь курсор — это вообще стратегически медленный прием, а в последнем примере пришлось использовать курсор еще и самого медленного типа (`SCROLL CURSOR`) для обеспечения произвольного доступа к строкам таблицы. Тест листинга 5.16 выдал следующий результат: 10 000 записей было сгенерировано приблизительно за 30 секунд.

Гораздо быстрее автогенерация данных для примера 2 будет выполняться, если в основу программы положить идею использования временной таблицы для того, чтобы в этой таблице выполнять случайный индексный поиск по ее автоинкрементному целому первичному ключу. Тест показал, что в этом случае 10 000 записей генерируется примерно за 8 секунд (листинг 5.17). Поскольку программа из этого листинга решает ту же самую задачу, что и программа из листинга 5.14, в ней опущены операторы создания целевых таблиц.

Листинг 5.17. Автогенерация данных для FK-ограничения с использованием индексированной временной таблицы

```
-- Создание временной таблицы
CREATE TABLE #Table1
       (Номер INT NOT NULL IDENTITY (1, 1)PRIMARY KEY,
       Код_Товара INT NOT NULL)
GO
-- Вставка записей во временную таблицу
INSERT INTO #Table1(Код_Товара) SELECT Код_Товара FROM ТОВАРЫ_ПРИМЕР
GO
-- Объявляем переменные
DECLARE @Counter INT, @End INT, @p INT, @Кол_Тов INT, @POS INT,
@f_Код_Товара INT, @f_Кол_прих INT, @f_Дата_прих DATETIME
```

```
-- Сколько значений товаров
SELECT @Кол_Тов = count(*) FROM #Table1
-- Задаем начальное значение счетчика цикла и количество
-- вставляемых записей
SELECT @Counter=1, @End=10000
-- Основной цикл
WHILE(@Counter <= @End)
BEGIN
SELECT @f_Кол_прих = 100 * RAND()+10
SELECT @f_Дата_прих = GetDate() + 10 * RAND()
-- Случайный выбор кода товара из временной таблицы
SELECT @POS=RAND()*(@Кол_Тов)+1
SELECT @f_Код_Товара = Код_Товара FROM #Table1 WHERE Номер = @POS
-- Вставка записи в целевую таблицу
INSERT INTO ПРИХОДЫ_ПРИМЕР(Код_Товара, Кол_прих, Дата_прих)
VALUES (@f_Код_Товара, @f_Кол_прих, @f_Дата_прих)
SET @Counter = @Counter + 1
END
-- Подсчет количества записей в целевой таблице
SELECT count(*) as ВСТАВЛЕНО_ЗАПИСЕЙ FROM ПРИХОДЫ_ПРИМЕР
-- Удаление временной таблицы
DROP TABLE #TABLE1
```

5.8.3. Пример 3. Дочерняя таблица с составным первичным ключом и с парой FK-ограничений

Более сложная задача автогенерации данных при наличии ограничений внешнего ключа может возникнуть, когда имеются две родительские таблицы и одна дочерняя, причем дочерняя таблица имеет составной первичный ключ, каждое из двух полей которого удовлетворяет FK-ограничению для соответствующей родительской таблицы. Выглядит все это, как показано на рис. 5.9.

Пусть данные в родительские таблицы уже введены. Нужно сгенерировать дочернюю таблицу таким образом, чтобы выполнялись все ограничения. Для решения этой задачи можно руководствоваться примером 2, усложнив его так, чтобы случайный выбор значений полей связи происходил поочередно из двух родительских таблиц, формируя таким образом текущую запись целевой таблицы. В этом случае соблюдение ограничения внешнего ключа гарантировано. Но выполнение ограничения первичного ключа для дочерней таблицы окажется под вопросом, т. к. при случайном поиске по родительским таблицам вполне может сгенерироваться два раза одна и та же комбинация значений (`N_расх`, `Товар_Ид`), хотя это и маловероятно.

Начало таблицы #ТОВ_ПОЗ

Номер	N_расх
1	1
2	1
3	1
4	2
5	2
6	3
7	3
8	3
9	4
10	4

Рис. 5.9. Дочерняя таблица с составным первичным ключом и с парой FK-ограничений

Предлагаемая программа (листинг 5.18) позволяет генерировать данные для целевой таблицы, удовлетворяющие ограничениям как внешнего, так и первичного ключа. Идея заключается в том, что сначала во временную таблицу заносятся упорядоченные значения номеров расходных накладных, причем заранее известно максимальное количество товарных позиций в расходной накладной, в нашем случае равное трем. Тогда при случайном выборе кода товара для текущего номера расходной накладной достаточно проследить, чтобы он не совпал с двумя предыдущими кодами товаров. Таким образом, два конечных значения последовательности кодов товаров постоянно сохраняются, и процесс генерации нового значения кода товара продолжается до тех пор, пока оно не будет отличным от двух предыдущих значений.

Во временную таблицу вставляются записи, которые представляют собой заготовки товарных позиций расходной накладной. Номера расходных накладных, взятые из таблицы `РАСХОДЫ`, нужно распределить по нескольким записям. Для этого используется вспомогательная таблица `НОМЕРА`, в единственном столбце которой введены числа от одного до пяти. Способ соединения таблиц в операторе вставки `INSERT INTO #ТОВ_ПОЗ` из листинга 5.18 основан на равенстве остатков от деления на два для пары значений. В результате получается, что четные номера расходных накладных дублируются два раза, а нечетные — три раза. Этот способ позволяет создать нужное количество расходных накладных с двумя или тремя товарными позициями, и работает он несколько быстрее, чем случайный выбор значения номера расходной наклад-

ной из ключевого поля таблицы РАСХОДЫ так, как это реализовано в листинге 5.17. Далее, имея таблицу #ТОВ_ПОЗ, можно будет просто в цикле перебрать ее записи по порядку. Начало таблицы #ТОВ_ПОЗ показано на рис. 5.9.

Листинг 5.18. Автогенерация данных для трех таблиц, связанных двумя ограничениями внешнего ключа

```
-- Создание целевой таблицы
if exists (SELECT * FROM dbo.sysobjects
WHERE id = object_id(N'[dbo].[ПОЗ_РАСХ_НАКЛ]')
and OBJECTPROPERTY(id, N'IsUserTable') = 1)
DROP TABLE [dbo].[ПОЗ_РАСХ_НАКЛ]
GO
CREATE TABLE [dbo].[ПОЗ_РАСХ_НАКЛ] ([N_расх] [INT] NOT NULL,
        [Товар_Ид] [INT] NOT NULL,
        [Кол_расх] [float] NOT NULL)
ON [PRIMARY]
GO
ALTER TABLE [dbo].[ПОЗ_РАСХ_НАКЛ] WITH NOCHECK ADD
        CONSTRAINT [PK_ПОЗ_РАСХ_НАКЛ] PRIMARY KEY CLUSTERED
        ([N_расх], [Товар_Ид]) ON [PRIMARY]
GO
 -- Создание временной таблицы
CREATE TABLE #ТОВ_ПОЗ
        (Номер INT NOT NULL IDENTITY (1, 1)PRIMARY KEY,
        N_расх INT NOT NULL)
GO
-- Вставка записей во временную таблицу
INSERT INTO #ТОВ_ПОЗ (N_расх)
SELECT N_расх FROM РАСХОДЫ t1, НОМЕРА t2 WHERE t1.N_расх%2=t2.НОМЕР%2
ORDER BY t1.N_расх
GO
-- Объявляем переменные
DECLARE @Counter INT, @End INT, @Кол_Тов INT, @f_Товар varchar(15),
@f_Кол_расх INT, @f_Код_Товара INT, @f1 INT, @f2 INT, @f_N_расх INT
-- Сколько значений товаров
SELECT @Кол_Тов = count(*) FROM ТОВАРЫ_ПРИМЕР
-- Начальные значения последовательности кодов товаров
SELECT @f1 =max(Товар_Ид)FROM ТОВАРЫ
SELECT @f2=@f1-2
-- Задаем начальное значение счетчика цикла
SELECT @Counter=1
```

```
-- Задаем конечное значение счетчика цикла, равное количеству записей
-- целевой таблицы
SELECT @End=count(*) FROM #ТОВ_ПОЗ
-- Основной цикл
WHILE(@Counter <= @End)
BEGIN
-- Выбор кода товара текущей записи целевой таблицы
SELECT @f_Код_Товара=RAND()*(@Кол_Тов)+1
              WHILE @f_Код_Товара=@f1 or @f_Код_Товара=@f2
              SELECT @f_Код_Товара=RAND()*(@Кол_Тов)+1
-- Случайное значение для количества товара
SELECT @f_Кол_расх = 100 * RAND()+10
-- Последовательный перебор записей временной таблицы
SELECT @f_N_расх = N_расх FROM #ТОВ_ПОЗ WHERE Номер = @Counter
-- Вставка записи в целевую таблицу
INSERT INTO ПОЗ_РАСХ_НАКЛ(N_расх, Товар_Ид, Кол_расх)
VALUES (@f_N_расх, @f_Код_Товара, @f_Кол_расх)
SET @Counter = @Counter + 1
-- Сдвиг последовательности кодов товаров
SELECT @f1=@f2, @f2=@f_Код_Товара
END
-- Подсчет количества записей в целевой таблице
SELECT count(*) as ВСТАВЛЕНО_ЗАПИСЕЙ FROM ПОЗ_РАСХ_НАКЛ
-- Удаление временной таблицы
DROP TABLE #ТОВ_ПОЗ
```

Тестирование этой программы показало, что 250 000 записей было занесено в целевую таблицу `ПОЗ_РАСХ_НАКЛ` меньше, чем за четыре минуты, что является вполне приемлемым результатом. При этом генерация данных занимает в среднем около 20 с, а остальное время тратится на выполнение оператора вставки в целевую таблицу.

5.8.4. Пример 4. Работа с графическими данными

Графические изображения, как известно, можно хранить на сервере в полях с типом данных `IMAGE`. Простейший способ вставки картинки из файла в такое поле — это использование команды `BULK INSERT` с форматным файлом, в котором нужно указать соответствующий тип столбца целевой таблицы и размер файла, содержащего изображение в байтах. Примером является программа из листинга 5.19.

Листинг 5.19. Занесение графического файла в базовую таблицу SQL-сервера

```
-- Создание целевой таблицы
if exists (SELECT * FROM dbo.sysobjects
```

```
WHERE id = object_id(N'[dbo].[IMAGE_TABLE]')
and OBJECTPROPERTY(id, N'IsUserTable') = 1)
DROP TABLE [dbo].[IMAGE_TABLE]
GO
CREATE TABLE [dbo].[IMAGE_TABLE] ([c1] [image] NULL)
ON [PRIMARY] TEXTIMAGE_ON [PRIMARY]
GO
-- Выполнение массового копирования
BULK INSERT IMAGE_TABLE FROM 'd:\sql_SERVER\agent.gif'
WITH (FORMATFILE = 'd:\sql_SERVER\graph.fmt')
```

Форматный файл с именем graph.fmt для этой команды имеет вид

```
8.0
1
1 SQLIMAGE 0 2144 "" 1 c1  ""
```

Данная команда вставляет в столбец `c1` таблицы `IMAGE_TABLE` изображение, находящееся в файле с именем agent.gif, причем размер файла agent.gif равен 2 144 байтам.

Однако возникает вопрос, как пользоваться подобным способом вставки на практике. Если на каждую картинку вручную создавать форматный файл, то шансов довести дело до конца имеется немного. Кроме того, как увидеть, что изображения действительно попали в целевую таблицу сервера в неискаженном виде? Ведь если при создании форматного файла ошибиться в задании размера файла изображения, то сама команда `BULK INSERT` сработает вроде бы успешно, во всяком случае, сообщение "`The command(s) completed successfully`" будет получено. Но при этом результат вставки будет либо искаженным (если размер файла изображения ошибочно задан меньше действительного), либо вставка не выполнится вовсе (если по ошибке размер файла изображения оказался большим, чем на самом деле).

Сразу следует сказать, что SQL-сервер предназначен только для хранения графической информации, а ее интерпретация — это дело клиента. Все современные среды разработки программного обеспечения, которые могут использовать SQL-сервер в качестве СУБД, имеют специальные объекты и средства работы с графическими данными, хранящимися на сервере. Сам же SQL-сервер для просмотра картинок определенно не предназначен. Однако кое-что можно увидеть и реализовать простейшими средствами.

Клиентское приложение, которое всегда под рукой и которое способно просматривать картинки, — это интернет-браузер. Поэтому если с помощью соответствующего мастера утилиты Enterprise Manager создать на основе таблицы с графическими данными статическую интернет-страницу, то все кар-

тинки будут видны. Более того, упомянутый мастер сгенерирует и набор файлов в общепринятом графическом формате, каждый из которых содержит картинку, соответствующую одному значению IMAGE-столбца. Нужный мастер называется Web Assistant Wizard, и получить доступ к нему можно через опции **Tools\Wizards...\Management**.

По поводу автоматизации вставки графических данных в таблицу SQL-сервера можно предложить следующее. Например, пусть имеется каталог, в котором находятся только графические файлы, подлежащие загрузке на сервер. Если удастся передать на сервер имя каждого из этих файлов и их общее количество, а также создать для каждого из них соответствующий форматный файл, то можно используя средства динамического SQL вызывать команду BULK INSERT в цикле, осуществляя тем самым нужные вставки записей в целевую таблицу.

Сначала создадим текстовый файл с именем f1.txt, с данными о содержимом каталога graph, в котором размещены графические файлы. Для этого выходную информацию команды DIR перенаправим в файл, послав из командной строки команду

```
DIR /-C >d:\ graph\f1.txt
```

Опция /-C здесь нужна, чтобы убрать символ-разделитель при выводе размера файла.

После выполнения этой команды в текстовом файле f1.txt, начиная с восьмой строки, будут находиться записи о файлах, изображения из которых мы хотим загрузить на SQL-сервер. Выглядеть эти сведения будут примерно так:

```
29-03-98 08:30p 2144 AGENT.GIF
29-03-98 08:30p 19916 ALLBXO.GIF
29-03-98 08:30p 740 APLIC.GIF
29-03-98 08:30p 753 BACK.GIF
29-03-98 08:30p 1982 CLIPB.GIF
29-03-98 08:30p 1441 COM_ENG.GIF
29-03-98 08:30p 1270 COM_RUS.GIF
29-03-98 08:30p 217 DOOR.GIF
```

Замечание

Если файл f1.txt создается в том же каталоге, что и графические файлы, то этот файл будет содержать строку с информацией о нем самом. Перед выполнением программы листинга 5.19 эту строку нужно удалить.

Далее передаем эту информацию на сервер, загружая содержимое текстового файла f1.txt в таблицу IMAGE_FILES, готовим форматные файлы, как это описано далее, и запускаем на выполнение программу, приведенную в листин-

ге 5.20. Эта программа осуществляет загрузку графических данных в целевую таблицу IMAGE_TABLE, структура которой приведена в предыдущем листинге 5.19.

Листинг 5.20. Загрузка картинок из общего каталога на SQL-сервер

```
-- Целевая таблица создана в листинге 5.19.
-- Создание таблицы для занесения в нее информации о графических файлах
if exists (SELECT * FROM dbo.sysobjects
WHERE id = object_id(N'[dbo].[IMAGE_FILES]')
and OBJECTPROPERTY(id, N'IsUserTable') = 1)
DROP TABLE [dbo].[IMAGE_FILES]
GO
CREATE TABLE [dbo].[IMAGE_FILES] ([Номер] [INT] IDENTITY (1, 1) NOT NULL,
        [Инф_файл] [varchar] (100) COLLATE Cyrillic_General_CI_AS NULL
) ON [PRIMARY]
GO
-- Копирование данных из текстового файла f1.txt на сервер
-- Текстовая строка данных целиком заносится в поле Инф_файл
BULK INSERT IMAGE_FILES FROM 'd:\graph\f1.txt'
WITH (FORMATFILE = 'd:\IM_FILES.FMT', FIRSTROW =8)
/* Удаляем последние две записи, т. к. они содержат итоговые сведения,
-- а не имя файла */
DELETE FROM IMAGE_FILES WHERE Инф_файл LIKE '% bytes%'
-- Выделяем из текстовой строки имя файла и заносим его на старое место
UPDATE IMAGE_FILES SET Инф_файл =
ltrim(reverse(left(reverse(Инф_файл), charindex(' ',reverse(Инф_файл)))))
-- Объявляем переменные
DECLARE @text nvarchar(200), @Counter INT, @End INT,
@Имя_файла varchar(50)
-- Задаем начальное и конечное значение счетчика цикла
SELECT @Counter=1
SELECT @End=count(*) FROM IMAGE_FILES
-- Основной цикл
WHILE(@Counter <= @End)
BEGIN
/* Формируем строку, содержащую текст команды массового копирования для
текущего имени файла */
SELECT @Имя_файла=Инф_файл FROM IMAGE_FILES
       WHERE Номер=@Counter
SELECT @TEXT= 'BULK INSERT IMAGE_TABLE FROM
 ''d:\GRAPH\'+@Имя_файла+'''
```

```
WITH (FORMATFILE = ''d:\GRAPH\'+
left(@Имя_файла, charindex('.', @Имя_файла))+'fmt'')'
-- SELECT @TEXT
/* Выполняем оператор динамического SQL, осуществляющий вставку строки
в целевую таблицу */
exec sp_executesql @TEXT
SELECT @Имя_файла
SET @Counter = @Counter + 1
END
-- Подсчет количества записей в целевой таблице
SELECT count(*) as ВСТАВЛЕНО_ЗАПИСЕЙ FROM IMAGE_TABLE
```

Форматный файл с именем im_files.fmt имеет вид

```
8.0
1
1 SQLCHAR 0 100 "\r\n" 2 Инф_файл Cyrillic_General_CI_AS
```

Для работы программы из листинга 5.20 важно, чтобы имя форматного файла совпадало с именем соответствующего графического файла.

Осталось только описать способ создания форматных файлов. Если данных немного, то форматные файлы можно создать вручную в любом текстовом редакторе. Но можно этот процесс и запрограммировать, используя любое доступное средство разработки программ. Наличие файла f1.txt, а он нужен обязательно, делает создание программы генерации форматных файлов тривиальной задачей. Так как все нужные нам сведения уже занесены в текстовый файл, то нет нужды при написании программы обращаться к средствам работы с файловой системой компьютера. Достаточно открыть файл f1.txt, перебрать его строки, выделить из каждой строки имя файла и его размер, после чего создать форматный файл с соответствующим именем и расширением, подставив в нужное место размер графического файла. Для примера приведена программа на простейшем языке Turbo Pascal (листинг 5.21), которая решает эту задачу.

Листинг 5.21. Форматирование форматного файла

```
program FMT_PREPARE;
var f, f1:text; S, name, len, filename:string; i: INTeger;
 begin
 Assign(f1, 'd:\gala\SQL_SERV\GRAPH\f1.txt');
 reset(f1);
 for i:=1 to 8 do readln(f1, S);
{ Цикл перебора строк файла f1.txt начиная с восьмой строки
продолжается до тех пор, пока не считается строка, содержащая в начале
пробел - в ней уже находится итоговая информация }
```

```
 WHILE not (copy(S, 1, 1)=' ') do
 begin
{выделение имени файла и размеров из считанной строки}
        delete(S, 1, 16);
 WHILE S[1]=' ' do delete(S, 1, 1);
 name:=copy(s, pos(' ', s)+1, length(s));
 name:=copy(name, 1, length(name)-3);
 len:=copy(s, 1, pos(' ', s)-1);
 filename:='d:\gala\SQL_SERV\GRAPH\'+name+'fmt';
 writeln(filename);
{создание форматного файла с текущим именем}
 Assign(f, filename);
 rewrite(f);
 writeln(f, '8.0');
 writeln(f, '1');
 writeln(f, '1 SQLIMAGE 0 '+len+' "" 1 c1 ""');
 close (f);
{чтение следующей строки файла f1.txt}
 readln(f1, S);
 END;
 close (f1);
 END.
```

5.9. Оптимизация быстродействия для некоторых типичных запросов

5.9.1. Пример 1. Поиск максимального значения

Рассмотрим классический пример использования скалярного подзапроса — поиск записей, у которых значение в некотором поле максимально (или минимально) среди всех значений этого поля. Например, необходимо найти сведения о приходных накладных, выписанных в самую позднюю дату. Или, еще проще, определить название самого дорогого товара. Будем ставить эксперименты на таблице ТОВАРЫ, которая приведена в проектном решении для задачи материального учета. Эта таблица содержит 20 записей и имеет кластерный индекс, реализующий ограничение первичного ключа. Напомним структуру этой таблицы (рис. 5.10).

Задачу определения записей таблицы ТОВАРЫ, соответствующих максимальной цене, решает запрос (листинг 5.22), план выполнения которого приведен на рис. 5.11.

ТОВАРЫ *

Column Name	Data Type	Length	Allow Nulls
Товар_Ид	int	4	
Товар	varchar	15	
Цена_учетн	money	8	
Остаток	int	4	

Рис. 5.10. Структура таблицы ТОВАРЫ

Листинг 5.22. Поиск сведений о товаре с максимальной ценой первым способом

```
SELECT Товар_Ид, Товар, Цена_Учетн
FROM ТОВАРЫ
WHERE Цена_Учетн =(SELECT MAX(Цена_Учетн)FROM ТОВАРЫ)
```

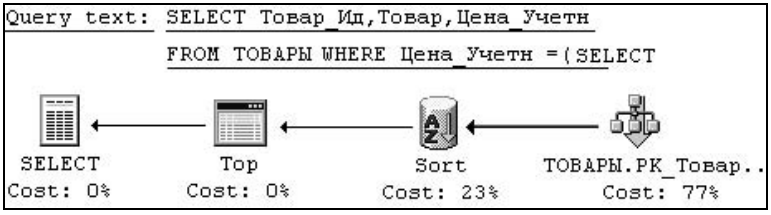


Рис. 5.11. План запроса, который определяет сведения о товаре с максимальной ценой

Данный запрос выполняется следующим образом. Таблица ТОВАРЫ сканируется по первичному ключу и сортируется по убыванию значений в поле Цена_Учетн, после чего результатом запроса будут верхние строки отсортированного набора с одинаковой (максимальной) ценой.

Перепишем предыдущий запрос с использованием локальной переменной:

```
DECLARE @C INT
SELECT @C=(SELECT MAX(Цена_Учетн)FROM ТОВАРЫ)
SELECT Товар_Ид, Товар, Цена_Учетн FROM ТОВАРЫ
WHERE Цена_Учетн = @C
```

Если после этого просмотреть графический план выполнения запроса, то окажется, что в нем приходится выполнять два оператора SELECT, каждый из которых занимает около 50 % от общих накладных расходов пакета. В каждом из этих операторов SELECT самой ресурсоемкой операцией, дающей практически 100 % вклада в величину стоимости, является сканирование кластерного индекса. При первом способе решения задачи сканирование кластерного индекса выполняется только один раз, на что выпадает примерно три четвер-

ти расходов. Если стоимость операции сканирования кластерного индекса принять за x, то стоимость решения задачи без использования локальной переменной можно принять за $1{,}25x$, а при ее использовании суммарная стоимость будет равна $2x$. Таким образом, наглядно подтверждена рекомендация о том, что локальных переменных лучше по возможности избегать.

Интересно, что если разместить подзапрос не в опции `WHERE`, а в опции `FROM`, применив *внутреннее представление* (inline view), то план запроса и стоимость практически не меняются. По эквивалентному плану (рис. 5.11) выполняется запрос

```
SELECT Товар_Ид, Товар, Цена_Учетн
FROM ТОВАРЫ AS T INNER JOIN
(SELECT MAX(Цена_Учетн) AS C FROM ТОВАРЫ) AS M ON T.Цена_Учетн = M.C
```

Замечание

Для всех приведенных способов решения задачи поиска сведений о товаре с максимальной ценой важно, чтобы в поле `Цена_Учетн` были запрещены `NULL`-значения, иначе в плане выполнения запроса будет присутствовать еще одна операция, которая заключается в использовании фильтрации для проверки того факта, что максимум не равен `NULL`-значению.

Эквивалентность планов выполнения для уже рассмотренных способов решения задачи поиска сведений о товаре с максимальной ценой не вызывает удивления — все три запроса используют один и тот же принцип сравнения значения поля, по которому производится поиск максимума с результатами скалярного подзапроса, и отличаются только способом размещения этого подзапроса в программном коде. Однако оказалось, что эквивалентный план выполнения (рис. 5.11) на таблице малого размера имеет запрос, приведенный в листинге 5.23.

Листинг 5.23. Поиск сведений о товаре с максимальной ценой вторым способом

```
SELECT TOP 1 WITH TIES Товар_Ид, Товар, Цена_Учетн
FROM ТОВАРЫ ORDER BY Цена_Учетн DESC
```

Этот запрос также определяет сведения о самом дорогом товаре, но он запрограммирован с использованием совершенно другого подхода. Таблица `ТОВАРЫ` сортируется по убыванию цены товара, после чего записи с нужными товарами попадают наверх. Опция `TOP 1 WITH TIES` указывает, что нужно вернуть все записи, а не одну, если окажется, что имеется несколько записей с совпадающим максимальным значением цены товара.

Итак, имеются два эквивалентных способа запрограммировать решение задачи поиска сведений о товаре с максимальной ценой. Если для написанных

запросов и таблицы ТОВАРЫ применить Мастер Настройки Индексов, то он не выдаст никаких рекомендаций. Казалось бы, оптимизация завершена, и то, как написан запрос, роли не играет. Но подобный вывод оказывается совершенно неверным, как только размеры таблиц существенно увеличиваются.

Для проведения тестов нужна таблица большого размера, в которой можно менять индексы. Создадим, например, таблицу с именем ПРИХОДЫ_1, которая по структуре соответствует таблице ПРИХОДЫ из задачи материального учета с помощью оператора

```
if exists (select * from dbo.sysobjects
where id = object_id(N'[dbo].[ПРИХОДЫ_1]')
and OBJECTPROPERTY(id, N'IsUserTable') = 1)
drop table [dbo].[ПРИХОДЫ_1]
GO
SELECT * INTO ПРИХОДЫ_1 FROM ПРИХОДЫ
```

Никаких индексов в таблице ПРИХОДЫ_1 пока нет, а NULL-значения, что существенно, запрещены во всех ее столбцах, потому что так было в таблице ПРИХОДЫ. Для таблицы ПРИХОДЫ_1 создадим кластерный индекс, соответствующий ее первичному ключу, с помощью оператора

```
ALTER TABLE [dbo].[ПРИХОДЫ_1] WITH NOCHECK ADD
       CONSTRAINT [PK_ПРИХ_СОРТ] PRIMARY KEY CLUSTERED
       ([N_прих]
       ) ON [PRIMARY]
GO
```

Для таблицы ПРИХОДЫ_1 будем решать задачу, аналогичную предыдущей, т. е. будем искать сведения о приходных накладных, в которых производилась закупка товара в максимальном количестве. Эту задачу, как уже известно, можно решить двумя способами (листинги 5.24 и 5.25). Какой же из этих способов лучше? Можно ли найти такие индексы, которые повысят быстродействие при работе с таблицей большого размера?

Листинг 5.24. Поиск максимума с помощью подзапроса в логическом условии

```
SELECT *
FROM ПРИХОДЫ_1
WHERE Кол_Прих=(SELECT MAX(Кол_Прих)FROM ПРИХОДЫ_1)
```

Листинг 5.25. Поиск максимума с помощью сортировки

```
SELECT TOP 1 WITH TIES *
       FROM ПРИХОДЫ_1
       ORDER BY Кол_прих DESC
```

Для запроса из листинга 5.24 план выполнения с добавленными комментариями приведен на рис. 5.12. Если сравнить этот план выполнения с планом выполнения запроса из листинга 5.22, то видно, что оптимизатор изменил план выполнения запроса только из-за того, что вырос размер таблицы.

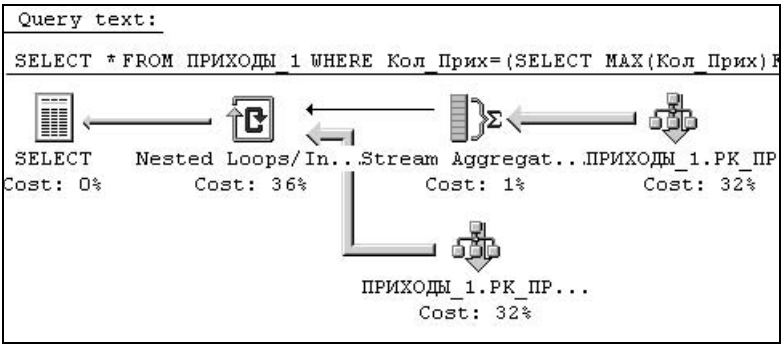


Рис. 5.12. План запроса для листинга 5.24

План запроса из листинга 5.25 (рис. 5.13) остается таким же, каким он был для маленькой таблицы, т. е. совпадает с планом, приведенным на рис. 5.11.

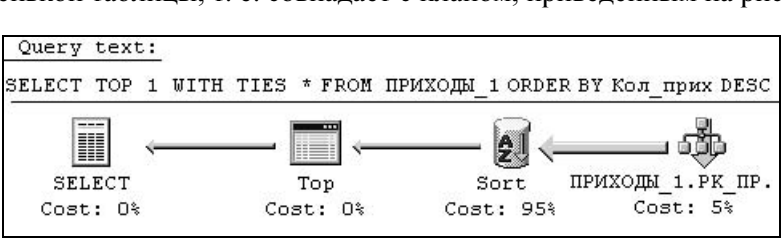


Рис. 5.13. План запроса для листинга 5.25

Но теперь нужно выполнять полную сортировку для таблицы, которая имеет не 20 записей, а в 10 000 раз больше. Относительная стоимость операции сортировки значительно возрастает, а общая стоимость выполнения запроса возрастает примерно в 7 раз, и задержка времени выполнения по сравнению с запросом из листинга 5.24 ощущается без всяких специальных измерений. Таким образом, проведение простейшего теста показало, что от того, как написан запрос, скорость его выполнения может изменяться в несколько раз.

Рассмотрим, как на создавшуюся ситуацию повлияет добавление индекса. Сначала нужно выяснить, какой индекс будет максимально полезным. Ответ на этот вопрос очевиден — необходим тот индекс, который поможет быстро найти максимальное значение в поле `Кол_Прих`, т. е. индекс на столбец `Кол_Прих`. Эту же рекомендацию выдаст и мастер настройки индексов, если его применить. Создаем нужный индекс, выполнив команду:

```
CREATE INDEX [IX_ПРИХОДЫ_1] ON [dbo].[ПРИХОДЫ_1]([Кол_прих])
ON [PRIMARY]
GO
```

После создания этого индекса планы запросов изменились коренным образом. Для листинга 5.24 план запроса теперь показан на рис. 5.14, а для листинга 5.25 — на рис. 5.15.

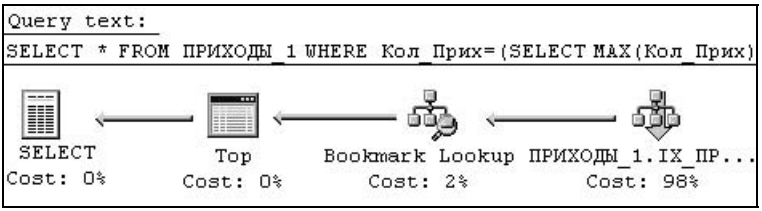


Рис. 5.14. План запроса для листинга 5.24 после добавления индекса по столбцу Кол_прих

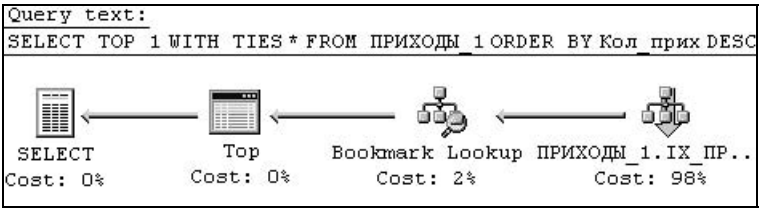


Рис. 5.15. План запроса для листинга 5.25 после добавления индекса по столбцу Кол_прих

Мы видим, что после добавления полезного индекса планы запросов почти совпадают. Теперь оба запроса начинаются со сканирования индекса IX_ПРИХОДЫ_1 и определения по закладке записей, соответствующих максимальному значению Кол_Прих. Первые три операции в обоих планах запроса полностью совпадают. Стоимость обоих способов выполнения запроса уменьшается примерно в 38 раз по сравнению с ситуацией, когда отсутствует индекс IX_ПРИХОДЫ_1. Практически неощутимая разница в стоимости возникает из-за того, что во втором запросе (см. листинг 5.25) элементарная операция SELECT имеет меньшую стоимость при наличии параметра WITH TIES.

5.9.2. Пример 2. Поиск значений, отсутствующих во внешнем ключе

Для пары таблиц, связанных ограничением внешнего ключа, ставится задача нахождения тех значений, которые присутствуют в поле связи родительской таблицы, но отсутствуют в поле связи дочерней таблицы. При этом оговаривается, допустимы ли NULL-значения в поле связи дочерней таблицы. Для

предметной области материального учета товаров эту задачу можно, например, сформулировать, как задачу нахождения товаров, которые ни разу не закупались, но присутствуют в таблице ТОВАРЫ. NULL-значения в поле связи дочерней таблицы в нашем случае запрещены. Родительская таблица по полю связи имеет уникальный кластерный индекс, реализующий ограничение первичного ключа, дочерняя таблица также проиндексирована по полю связи, т. е. созданы индексы в соответствии со стандартной рекомендацией для FK-ограничения. В родительской таблице записей намного меньше, чем в дочерней таблице, а именно в данном конкретном случае имеется около 20 записей в родительской таблице и около 200 000 в дочерней.

Для тестирования быстродействия разных способов решения этой задачи возьмем пару таблиц ТОВАРЫ и ПРИХОДЫ, связанных FK-ограничением по полю Товар_Ид, со структурой, приведенной на рис. 5.16.

Рис. 5.16. Пара таблиц с FK-ограничением

Данную задачу можно решить тремя способами, представленными на листингах 5.26, 5.27 и 5.28 соответственно. После каждого листинга приведены стоимость и время выполнения запроса на тестовых данных. Эти величины имеют смысл только при их сравнении друг с другом для определения самого быстрого способа решения задачи.

Листинг 5.26. Использование `Left Join`

```
SELECT DISTINCT ТОВАРЫ.Товар_Ид
FROM ТОВАРЫ LEFT OUTER JOIN ПРИХОДЫ
ON ТОВАРЫ.Товар_Ид = ПРИХОДЫ.Товар_Ид
WHERE (ПРИХОДЫ.Товар_Ид IS NULL)
```

Стоимость запроса составляет 0,338, а время выполнения — порядка 156 мс.

План запроса из листинга 5.26 приведен на рис. 5.17. Из него видно, что этот способ выполнения запроса будет работать потенциально медленно, потому что для формирования условия фильтрации здесь нужно, чтобы в соединении LEFT OUTER JOIN участвовали все 200 000 записей большой таблицы, что показано жирной стрелкой. Дальше мы увидим, что ни один из двух других планов запросов не требует выполнения соединения такого большого количества записей.

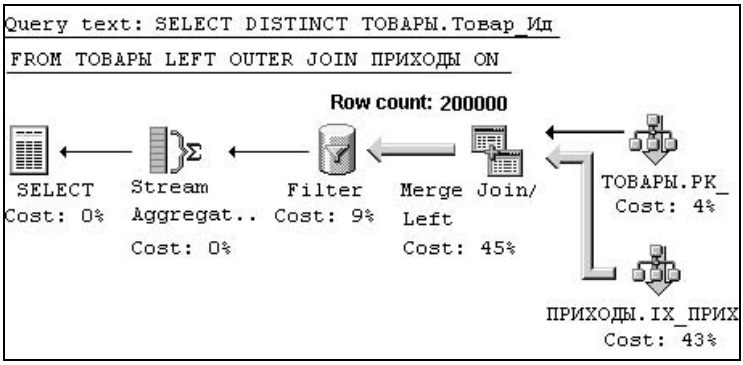


Рис. 5.17. План запроса с использованием `LEFT OUTER JOIN`

Листинг 5.27. Использование коррелированного подзапроса с предикатом `EXISTS`

```
SELECT Товар_Ид FROM ТОВАРЫ WHERE not EXISTS
(SELECT Товар_Ид FROM ПРИХОДЫ WHERE ТОВАРЫ.Товар_Ид = ПРИХОДЫ.Товар_Ид)
```

Стоимость запроса равна 0,047, время выполнения — порядка 2 мс.

План выполнения этого запроса приведен на рис. 5.18.

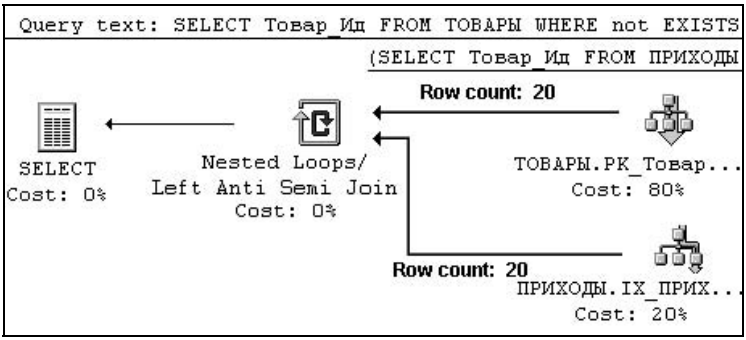


Рис. 5.18. План запроса с использованием коррелированного подзапроса и предиката `EXISTS`

Листинг 5.28. Использование подзапроса в логическом условии

```
-- С предикатом IN
SELECT ТОВАРЫ.Товар_Ид FROM ТОВАРЫ
WHERE ТОВАРЫ.Товар_Ид NOT IN (SELECT DISTINCT Товар_Ид FROM ПРИХОДЫ)
```

```
-- С предикатом SOME
SELECT ТОВАРЫ.Товар_Ид FROM ТОВАРЫ
WHERE not(ТОВАРЫ.Товар_Ид = SOME (SELECT DISTINCT Товар_Ид
FROM ПРИХОДЫ))
```

Стоимость запроса равна 0,047. Время выполнения — порядка 5 мс.

План выполнения последнего запроса одинаков как для предиката `IN`, так и для предиката `SOME`. Этот план выполнения содержит две операции со стоимостью, отличной от нуля, и они в точности совпадают с операциями сканирования индексов в плане запроса для листинга 5.27, поэтому и стоимость выполнения запросов листинга 5.27 и листинга 5.28 одинакова. Однако план выполнения запроса из листинга 5.28 содержит помимо сканирования индексов еще множество других операций со стоимостью, практически равной нулю, среди которых два соединения с вложенными циклами. Этот план слишком громоздок, и из-за этого его рисунок не приводится. Время же выполнения запроса из листинга 5.27 примерно в два раза меньше, чем у запроса из листинга 5.28, что иллюстрирует общую рекомендацию о преимуществе использования коррелированного подзапроса и предиката `EXISTS` в ситуации, когда они взаимозаменяемы. В табл. 5.6 приведены сводные результаты тестов для примера о поиске отсутствующих во внешнем ключе значений родительской таблицы.

Таблица 5.6. *Сравнение параметров быстродействия запросов*

Способ реализации запроса	С индексом на поле внешнего ключа		Без индекса на поле внешнего ключа	
	Стоимость	Время	Стоимость	Время
Листинг 5.26. Использование `Left Join`	0,338	156 мс	2,59	302 мс
Листинг 5.27. Использование коррелированного подзапроса с предикатом `EXISTS`	0,047	2 мс	0,572	80 мс
Листинг 5.28. Использование подзапроса в логическом условии	0,047	5 мс	1,53	152 мс

Из рассмотренного примера следует общепринятая рекомендация, суть которой состоит в следующем: если при написании запроса имеется альтернатива между `EXISTS` и `IN` (`NOT IN`), то для повышения быстродействия предпочтительнее использование `EXISTS`.

5.9.3. Пример 3. Поиск максимума от агрегатной функции

Рассмотрим решение задачи поиска расходной накладной (накладных) с максимальной суммарной выручкой на примере таблиц из базы данных материального учета, связанных ограничением внешнего ключа (рис. 5.19).

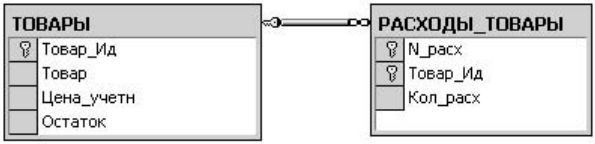


Рис. 5.19. Таблицы для поиска расходных накладных с максимальной суммарной выручкой

Задачу можно решать двумя способами на базе подхода, рассмотренного в первом примере. Использование способа решения задачи на основе опции `TOP 1 WITH TIES` приводит к запросу, текст которого приведен в листинге 5.29.

Листинг 5.29. Поиск максимума от агрегатной функции первым способом

```
SELECT TOP 1 WITH TIES A1.N_расх,
SUM(A1.Кол_расх * A2.Цена_Учетн * 1.1) AS МАКС_ВЫРУЧКА
FROM РАСХОДЫ_ТОВАРЫ A1 INNER JOIN
ТОВАРЫ A2 ON (A1.Товар_Ид = A2.Товар_Ид)
GROUP BY A1.N_расх
ORDER BY МАКС_ВЫРУЧКА DESC
```

Этот запрос является прямой аналогией запроса из листинга 5.23 и особых пояснений не требует. На таблицах из нашей базы данных он выполняется около трех миллисекунд и имеет стоимость порядка 10.

Сложнее запрограммировать решение задачи с использованием подзапроса. Проблема состоит в том, что на языке SQL нельзя напрямую написать вложенные агрегатные функции. Например, конструкция `MAX(SUM(ИМЯ_ПОЛЯ))` является недопустимой. Поэтому приходится дважды применять одно и то же внутреннее представление, которое вычисляет суммарную выручку для каждой расходной накладной (листинг 5.30). При изучении листинга 5.30 следует обратить внимание на правильное задание алиаса внутреннего представления и алиаса столбца для вывода максимальной выручки.

Листинг 5.30. Поиск максимума от агрегатной функции вторым способом

```
SELECT N_расх, МАКС_ВЫРУЧКА
FROM
-- Внутреннее представление, его алиас P
       (SELECT РАСХОДЫ_ТОВАРЫ.N_расх,
       SUM(РАСХОДЫ_ТОВАРЫ.Кол_расх * ТОВАРЫ.Цена_Учетн * 1.1)
       AS МАКС_ВЫРУЧКА FROM РАСХОДЫ_ТОВАРЫ INNER JOIN ТОВАРЫ
       ON РАСХОДЫ_ТОВАРЫ.Товар_Ид = ТОВАРЫ.Товар_Ид
       GROUP BY РАСХОДЫ_ТОВАРЫ.N_расх) P
WHERE МАКС_ВЫРУЧКА=
        (SELECT MAX(МАКС_ВЫРУЧКА) FROM
-- Внутреннее представление, его алиас P
              (SELECT РАСХОДЫ_ТОВАРЫ.N_расх,
              SUM(РАСХОДЫ_ТОВАРЫ.Кол_расх * ТОВАРЫ.Цена_Учетн * 1.1)
              AS МАКС_ВЫРУЧКА FROM РАСХОДЫ_ТОВАРЫ INNER JOIN ТОВАРЫ
              ON РАСХОДЫ_ТОВАРЫ.Товар_Ид = ТОВАРЫ.Товар_Ид
              GROUP BY РАСХОДЫ_ТОВАРЫ.N_расх) P)
```

Запрос из листинга 5.29 выполняется также около трех миллисекунд и имеет примерно такую же стоимость, как и запрос из листинга 5.28. Планы выполнения обоих запросов также практически одинаковы — сначала выполняется соединение таблиц по их внешнему ключу с помощью `HASH JOIN`, далее происходит агрегация и сортировка данных, после чего выводятся результаты. Индекс внешнего ключа не используется, если же его навязать с помощью подсказки оптимизатору, то запрос будет работать медленнее.

Если пытаться проводить оптимизацию решения последней задачи по схеме из примера 1, то нужно создать индекс для вычисления максимума, т. е. проиндексировать выражение `Кол_расх*Цена_Учетн*1.1`. Физически этот индекс можно привязать только к индексированному представлению *(см. разд. 5.12)*. Такой индекс можно создать для целочисленного типа выражения `Кол_расх*Цена_Учетн*1.1`, но оказывается, что он не помогает.

Мастер Настройки Индексов для запроса из листинга 5.29 в режиме **Medium** выдал рекомендацию создать *индексированное представление (см. разд. 5.12)*. Сокращенный код этой рекомендации с добавленными комментариями приведен в листинге 5.31. Сокращения кода касаются удаления из полного текста рекомендаций операторов, связанных с настройками соединения (`SET`). Имена вновь создаваемых объектов оставлены такими, какими их назначил Мастер Настройки Индексов.

Листинг 5.31. Индексированное представление для запроса из листинга 5.29

```
-- Проверка существования объекта представления перед его созданием
if exists (SELECT * FROM dbo.sysobjects
WHERE id = object_id(N'[dbo].[_hypmv_0]') and
OBJECTPROPERTY(id, N'IsView') = 1)
DROP VIEW [dbo].[_hypmv_0]
GO
-- Создание представления
CREATE VIEW [dbo].[_hypmv_0] WITH SCHEMABINDING AS
       SELECT [dbo].[РАСХОДЫ_ТОВАРЫ].[n_расх] AS _hypmv_0_col_1,
        [dbo].[РАСХОДЫ_ТОВАРЫ].[кол_расх] AS _hypmv_0_col_2,
        [dbo].[товары].[цена_учетн] AS _hypmv_0_col_3,
        [dbo].[товары].[Товар_Ид] AS _hypmv_0_col_4,
        [dbo].[РАСХОДЫ_ТОВАРЫ].[Товар_Ид] AS _hypmv_0_col_5
       FROM [dbo].[товары], [dbo].[РАСХОДЫ_ТОВАРЫ]
       WHERE ( [dbo].[товары].[Товар_Ид] =
       [dbo].[РАСХОДЫ_ТОВАРЫ].[Товар_Ид])
GO
-- Создание уникального кластерного индекса для представления
CREATE UNIQUE CLUSTERED INDEX [_hypmv_01]
ON [dbo].[_hypmv_0]([_hypmv_0_col_1], [_hypmv_0_col_5]) ON [PRIMARY]
GO
```

Запрос из листингов 5.29 и 5.30 при наличии представления из листинга 5.31 выполняется примерно в два раза быстрее и имеет соответственно меньшую стоимость.

Замечание

С точки зрения общих рекомендаций *(см. разд. 5.12)*, задача сложного агрегирования для соединенных таблиц является типичной, когда применение индексированного представления может оказаться полезным. Однако достаточно сложно определить конкретную структуру этого представления и его индексов. Поэтому использование Мастера Настройки Индексов является хорошей мерой.

5.9.4. Пример 4. Сложное агрегирование

Рассмотрим задачу вычисления остатков каждого товара в ситуации, когда не используется триггер на поле `Остаток` таблицы `ТОВАРЫ` *(см. разд. 5.11)*. Пусть нам необходимо вычислить остатки каждого товара на текущий момент времени. Предполагается, что все сведения в базе данных корректны, т. е. ни разу не было отпуска товара по расходной накладной, когда этого товара не

было в наличии в нужном количестве. В этом случае из информации, содержащейся в приходных и расходных накладных, можно вычислить остатки любого товара на любую дату. Будем для простоты вычислять эти остатки в предположении, что обработаны все внесенные в базу данных приходные и расходные накладные, т. е. условие по диапазону дат не задается.

Итак, необходимо для основного проектного решения базы данных материального учета подсчитать количество товаров каждого типа, оставшихся на складе после выполнения производственных операций по всем приходным и расходным накладным, в предположении, что все расходные накладные корректны. Причем следует учесть возможность того, что существуют товары, которые закупались, но ни разу не были проданы.

Первый способ решения задачи основан на внешнем соединении запросов, в которых по отдельности вычислены суммы закупок и суммы продаж для каждого товара. Эта программа приведена в листинге 5.32.

Листинг 5.32. Вычисление остатков с помощью внешнего соединения подзапросов

```
SELECT КУПИЛИ.ТОВАР_Ид, КУПИЛИ.КОЛ_КУПЛ, ПРОДАЛИ.КОЛ_ПРОД,
-- Вычитание общей суммы продаж из общей суммы закупок
       CASE WHEN ПРОДАЛИ.ТОВАР_Ид IS NULL
              THEN КУПИЛИ.КОЛ_КУПЛ
              ELSE КУПИЛИ.КОЛ_КУПЛ – ПРОДАЛИ.КОЛ_ПРОД
              END
       AS ОСТАТОК
FROM
-- Суммарное количество закупок каждого товара
       (SELECT ПРИХОДЫ.ТОВАР_Ид, SUM(ПРИХОДЫ.Кол_прих) AS КОЛ_КУПЛ
       FROM ПРИХОДЫ
       GROUP BY ПРИХОДЫ.ТОВАР_Ид)AS КУПИЛИ
LEFT JOIN
-- Суммарное количество продаж каждого товара
       (SELECT РАСХОДЫ_ТОВАРЫ.ТОВАР_Ид,
       SUM(РАСХОДЫ_ТОВАРЫ.Кол_расх) AS КОЛ_ПРОД
       FROM РАСХОДЫ_ТОВАРЫ
       GROUP BY РАСХОДЫ_ТОВАРЫ.ТОВАР_Ид) AS ПРОДАЛИ
ON КУПИЛИ.ТОВАР_Ид = ПРОДАЛИ.ТОВАР_Ид
ORDER BY КУПИЛИ.ТОВАР_Ид
```

Второй способ решения задачи состоит в использовании операции объединения таблиц с товарными позициями приходных и расходных накладных. В этом случае опция `UNION` применяется для запросов, в которых вычислено

суммарное количество закупок и продаж для каждого товара, причем закупки берутся со знаком плюс, а продажи — со знаком минус. Программа приведена в листинге 5.33.

Листинг 5.33. Вычисление остатков товаров с помощью объединения

```
SELECT ТОВАР_Ид, SUM(РАЗНИЦА)AS ОСТАТОК
FROM
-- Суммарное количество закупок каждого товара
       (SELECT ПРИХОДЫ.ТОВАР_Ид, SUM(ПРИХОДЫ.Кол_прих) AS РАЗНИЦА
 FROM ПРИХОДЫ
       GROUP BY ПРИХОДЫ.ТОВАР_Ид
       UNION ALL
-- Суммарное количество продаж каждого товара
       SELECT РАСХОДЫ_ТОВАРЫ.ТОВАР_Ид, - SUM(РАСХОДЫ_ТОВАРЫ.Кол_расх)
       AS КОЛ_ПРОД
       FROM РАСХОДЫ_ТОВАРЫ
       GROUP BY РАСХОДЫ_ТОВАРЫ.ТОВАР_Ид)AS T
GROUP BY ТОВАР_Ид
ORDER BY ТОВАР_Ид
```

Оба запроса из листингов 5.32 и 5.33 имеют в случае нашей базы данных одинаковую стоимость и выполняются около одной миллисекунды.

Рекомендация Мастера Настройки Индексов для листинга 5.33, которая приведена в листинге 5.34, позволила снизить стоимость запроса примерно вдвое, и более чем вдвое уменьшилось время его выполнения.

Листинг 5.34. Индексированное представление для запроса из листинга 5.32

```
-- Проверка существования объекта представления перед его созданием
if exists (SELECT * FROM dbo.sysobjects WHERE
id = object_id(N'[dbo].[_hypmv_2]') and
OBJECTPROPERTY(id, N'IsView') = 1)
DROP VIEW [dbo].[_hypmv_2]
GO
-- Суммарное количество закупок каждого товара
CREATE VIEW [dbo].[_hypmv_2] WITH SCHEMABINDING AS
       SELECT [dbo].[РАСХОДЫ_ТОВАРЫ].[Товар_Ид] AS _hypmv_2_col_1,
       SUM([mat_uchet].[dbo].[РАСХОДЫ_ТОВАРЫ].[Кол_расх]) AS
       _hypmv_2_col_2,
       COUNT_BIG(*) AS _hypmv_2_col_3
       FROM [dbo].[РАСХОДЫ_ТОВАРЫ]
       GROUP BY [dbo].[РАСХОДЫ_ТОВАРЫ].[Товар_Ид]
```

```
GO
-- Создание уникального кластерного индекса для представления
CREATE UNIQUE CLUSTERED INDEX [_hypmv_24]
ON [dbo].[_hypmv_2]([_hypmv_2_col_1]) ON [PRIMARY]
GO
```

5.10. Использование курсоров

Курсоры являются стандартным средством *построчной обработки* табличных данных. Основная рекомендация по поводу курсоров состоит в том, что по возможности их следует избегать, т. к. курсоры используют много ресурсов SQL-сервера, уменьшая тем самым эффективность и масштабируемость прикладных программ. Всегда стоит попытаться найти альтернативное решение, основанное на табличной обработке, вместо выполнения лобовой построчной операции. Возможно вместо построчного перебора таблицы рассмотреть создание запросов с временными или производными таблицами или с использованием коррелированных подзапросов. Вместо курсора может быть использована последовательность операторов `SELECT`, связанных с помощью табличных переменных. Или же более эффективным решением вообще окажется перенесение построчной обработки на клиента. Этот прием является особенно удачным в том случае, когда набор данных для построения курсора невелик по объему и его передача по сети клиенту не слишком накладна. Однако стоит избегать клиентских курсоров, если число обрабатываемых строк велико или данные являются статическими. В этом случае рекомендуется рассмотреть использование серверного `KEYSET`-курсора вместо клиентского курсора. Эффективность обычно увеличивается из-за сокращения сетевого движения между пользователем и сервером. Для достижения оптимальной эффективности, вероятно, придется попробовать оба типа курсоров на более или менее реальных данных, чтобы определить, какой из них является лучшим для конкретной среды.

Если все же альтернативы использованию серверного курсора найти не удалось, тогда следует подумать о типе используемого курсора. Лучше начинать выбор типа курсора с быстрых курсоров, допускающих только последовательную обработку, т. е. с курсоров типа `FORWARD_ONLY`, `FAST_FORWARD` или `READ_ONLY`. Эти курсоры допускают только чтение данных. При работе с такими курсорами предпочтительно использование опции `FAST_FORWARD` вместо опции `FORWARD_ONLY`. Если невозможно использовать быстрые курсоры, тогда следует попробовать курсоры других типов, которые перечислены в порядке характеристик их эффективности от самого быстрого до самого медленного: *динамический*, *статический*, *ключевой*. Для курсора любого типа нужно стараться организовывать цикл перебора записей так, чтобы при первой возможности выходить из цикла и закрывать курсор.

Для своей работы все типы курсоров используют базу данных TEMPDB, поэтому если не удалось найти программной альтернативы интенсивному использованию курсоров, рекомендуется пробовать расположить базу данных TEMPDB на отдельном физическом устройстве для повышения эффективности обмена данными с диском.

Использование курсоров может уменьшить параллелизм и привести к ненужным блокировкам *(см. главу 6)*. Чтобы избежать этого, нужно использовать опцию курсора READ_ONLY, если это допустимо. Если курсор применяется для модификации данных, то можно попробовать использование опции OPTIMISTIC. Эта опция изменяет способ блокировки данных, применяемых курсором, на *оптимистический* вместо принятого по умолчанию *пессимистического* способа блокировки. Рекомендуется избегать использования опции курсора SCROLL_LOCKS, которая уменьшает параллелизм. Кроме того, для сокращения времени блокировок полезно сводить к минимуму количество строк, изменяемых с помощью курсора.

Когда курсор отработал, его не только нужно закрыть (CLOSE), но и освободить (DEALLOCATE). Освобождение требуется, чтобы вернуть серверу ресурсы, используемые курсором, т. к. закрытие курсора только снимает блокировки, но не освобождает ресурсы сервера. Следует помнить, что локальные курсоры освобождаются автоматически по завершению породившего их пакета, в то время как глобальные курсоры нужно освобождать явно. Глобальные курсоры полезны, если один и тот же курсор используется в пакете больше одного раза. Не закрывая и не освобождая глобальный курсор раньше, чем это нужно, можно выиграть то время, которое затрачивается на эти операции.

Если для прикладной программы это подходит, то лучше загружать курсор как можно скорее, сразу перемещаясь на его последнюю строку. Это снимает разделяемые блокировки, возникающие при создании курсора.

Если в операторе SELECT курсора нужно выполнять операцию соединения (JOIN), то для этого случая курсоры с опциями KEYSET и STATIC обычно работают быстрее.

Если ожидается, что набор результатов будет очень большим, можно рассмотреть использование асинхронных курсоров. Это позволяет прикладной программе начать обрабатывать данные, в то время как курсор все еще заполняется. Такой прием фактически не ускоряет обработку, но создает на клиенте иллюзию быстродействия.

5.10.1. Примеры использования курсоров

Характерный пример использования курсора приведен в *разд. 5.8* (листинг 5.15). Там же сделан вывод о том, что рассмотренная задача с помощью курсора решается крайне неэффективно. Далее в листинге 5.17 предложен

способ замены курсора, выполняющего построчную обработку набора данных с равномерно случайным доступом, табличной обработкой с помощью оператора SELECT и даже с использованием индекса. Результаты тестирования времени выполнения обоих вариантов решения задачи говорят сами за себя. Очевидно, что использование приема из листинга 5.17 позволяет избежать применения курсоров для достаточно широкого круга задач.

Пример 1. Использование курсора для задачи администрирования

Можно, однако, привести пример безусловно полезного использования курсора, который иллюстрирует общий способ автоматизации широкого круга административных задач. Идея заключается в создании курсоров на основе системных таблиц для получения различных метаданных, которые комбинируются в исполняемые сервером команды с помощью динамического SQL. В подобных задачах редко возникает необходимость обработки слишком больших наборов данных. Кроме того, эти задачи, как правило, не нужно решать в моменты, когда сервер перегружен, а время их выполнения редко бывает критичным.

Рассмотрим задачу назначения права выполнения оператора SELECT пользователю с заданным именем на все таблицы, имеющиеся в базе данных. Если эту задачу решать средствами статического SQL, то придется писать множество операторов GRANT, меняя код программы каждый раз, когда появляется новая таблица. Визуальные средства администрирования тоже не удастся применять не задумываясь. В листинге 5.35 приведен код хранимой процедуры, которая принимает входным параметром имя пользователя и назначает ему заданный перечень прав.

Листинг 5.35. Процедура назначения прав пользователю

```
CREATE PROCEDURE GRANT_SELECT @par1 SYSNAME
AS
SET ANSI_DEFAULTS ON
-- Объявление переменной для хранения имени текущей базовой таблицы
DECLARE @au_table SYSNAME
-- Объявление курсора
DECLARE ALL_USER_TABLES CURSOR
       FOR SELECT name FROM sysobjects WHERE type='U'
-- Открытие курсора
OPEN ALL_USER_TABLES
-- Выборка первой записи курсора
FETCH NEXT FROM ALL_USER_TABLES INTO @au_table
```

```
-- Выполнение динамического оператора присвоения прав первый раз
EXEC('GRANT SELECT ON '+ @au_table +' to ' + @par1)
-- Основной цикл
WHILE @@FETCH_STATUS = 0
       BEGIN
       -- Выборка текущей записи курсора
       FETCH NEXT FROM ALL_USER_TABLES INTO @au_table
       -- Выполнение динамического оператора присвоения прав
       EXEC('GRANT SELECT ON '+@au_table+' to ' + @par1)
       SELECT 'GRANT SELECT ON '+ @au_table +' to ' + @par1
       END
-- Закрытие и освобождение курсора
CLOSE ALL_USER_TABLES
DEALLOCATE ALL_USER_TABLES
```

Замечание

Для работы программы из последнего листинга важно, как настроены параметры соединения, поэтому нужный оператор их настройки `SET ANSI_DEFAULTS ON` вставлен в начало процедуры.

Пример 2. Замена курсора табличной обработкой

В этом примере показана замена курсора табличной обработкой для задачи, в которой нужно выполнять сравнение данных из разных записей одной и той же таблицы в ситуации, когда невозможно выполнить эквисоединение указанной таблицы с ее копией. При такой формулировке задачи использование курсора и сохранение выбранных из него записей в переменных кажется очевидным решением. Рассмотрим это решение на конкретном примере, а затем попытаемся найти способ обойтись без курсора.

Пусть нужно найти учетную цену заданного товара на заданную дату с учетом возможности изменения цен. Сведения об изменении цен на товары занесены в таблицу `ИСТОРИЯ_ЦЕН` (см. основное проектное решение для базы данных `МАТ_УЧЕТ`). Структура таблицы `ИСТОРИЯ_ЦЕН` приведена на рис. 5.20.

Чтобы понять, что же нужно определить в этом примере, рассмотрим такое рассуждение. Предположим, что с бананами начали работать 01-Jan-00. В таблицу `ИСТОРИЯ_ЦЕН` заносятся сведения о цене бананов на указанную дату, которая равна, например, 3 руб. Цена на бананы повышается таким образом, что 01-Jan-00 учетная цена бананов равна 3 руб., 02-Jan-00, 03-Jan-00, 04-Jan-00 и 05-Jan-00 она уже равна 4 руб., а начиная с 06-Jan-00 цена бананов достигла значения 5 руб. и больше в течение января не менялась, в феврале же опять дважды возросла. Эти сведения также приведены на рис. 5.20. Считаем, что идентификатор бананов равен двум, — это значение и будут выводить запросы.

Товар_Ид	Дата_цена	Цена_учетн
2	01-Jan-00	3
2	02-Jan-00	4
2	06-Jan-00	5
2	12-Feb-00	5,5
2	28-Feb-00	5,8

Рис. 5.20. Данные и структура таблицы ИСТОРИЯ_ЦЕН

Из условия задачи следует, что для таблицы ИСТОРИЯ_ЦЕН, отсортированной по первичному ключу (Товар_Ид, Дата_цена), нужно найти такую пару записей, для которой введенная дата больше или равна значению поля Дата_цена для первой записи, но меньше значения поля Дата_цена для второй записи пары. Тогда искомая учетная цена определяется полем Цена_учетн первой записи найденной пары записей. Напрашивается решение задачи с помощью курсора, приведенное в листинге 5.36.

Листинг 5.36. Поиск текущей учетной цены с помощью курсора

```
SET ANSI_DEFAULTS ON
DECLARE @Товар_Ид INT, @ЗАД_ДАТА DATETIME
DECLARE @P1 MONEY, @P2 MONEY, @D1 DATETIME, @D2 DATETIME
SET @Товар_Ид=2
SET @ЗАД_ДАТА='2000-Jan-12'
DECLARE WORK_CURSOR CURSOR FAST_FORWARD FOR
       SELECT Дата_цена, Цена_учетн
       FROM ИСТОРИЯ_ЦЕН
       WHERE Товар_Ид = @Товар_Ид
       ORDER BY 1, 2
OPEN WORK_CURSOR
FETCH NEXT FROM WORK_CURSOR INTO @D1, @P1
WHILE @@FETCH_STATUS = 0
BEGIN
FETCH NEXT FROM WORK_CURSOR INTO @D2, @P2
       IF @D2 > @ЗАД_ДАТА
       BREAK
       ELSE
       BEGIN
       SET @P1 = @P2
       END
END
```

```
SELECT @Товар_Ид AS Товар_Ид, @ЗАД_ДАТА AS ЗАД_ДАТА, @P1 AS УЧЕТН_ЦЕНА
CLOSE WORK_CURSOR
DEALLOCATE WORK_CURSOR
-- Ответ
Товар_Ид            ЗАД_ДАТА                        УЧЕТН_ЦЕНА
2                   2000-01-12 00:00:00.000         5.0000
```

Казалось бы, без курсора обойтись в данной задаче нельзя — ведь нужна проверка условия одновременно для двух разных записей, к чему оператор SELECT в принципе не приспособлен. Тем более что про созданный в листинге 5.35 курсор ничего плохого сказать нельзя — он не велик по объему, т. к. условие отбора по заданному товару присутствует сразу при создании курсора, а перебор всех его записей будет происходить редко. Кроме того, удалось воспользоваться опцией FAST_FORWARD, создав тем самым курсор самого быстрого типа.

И все же задачу можно решить без курсора. Переформулируем условие для поиска цены следующим образом: в отсортированной таблице нам нужна запись, у которой значение поля Дата_цена максимально возможное среди всех значений этого поля, меньших или равных заданному значению даты. Запрос, приведенный в листинге 5.37, выдаст значение поля Дата_цена, которое определяет запись, содержащую искомую учетную цену для заданного товара.

Листинг 5.37. Подзапрос для поиска текущей учетной цены с помощью оператора SELECT

```
DECLARE @Товар_Ид INT, @ЗАД_ДАТА DATETIME
SET @Товар_Ид = 2
SET @ЗАД_ДАТА ='2000-Jan-12'
SELECT max(Дата_цена)AS Дата_цена
FROM ИСТОРИЯ_ЦЕН WHERE Товар_Ид = @Товар_Ид and Дата_цена <= @ЗАД_ДАТА
-- Ответ
Дата_цена
2000-01-06 00:00:00
```

Но как получить остальные поля из записи, определяющей цену товара? Еще лучше было бы получить точно такой же выходной набор данных как в решении с курсором. Воспользуемся результатом предыдущего запроса в качестве условия отбора строк из таблицы. Полное решение задачи вместе с вводом параметров приведено в листинге 5.38.

Листинг 5.38. Поиск текущей учетной цены с помощью одного оператора SELECT

```
DECLARE @Товар_Ид INT, @ЗАД_ДАТА DATETIME
SET @Товар_Ид = 2
SET @ЗАД_ДАТА ='2000-Jan-12'
```

```
SELECT Товар_Ид, @ЗАД_ДАТА AS ЗАД_ДАТА, Цена_учетн AS УЧЕТН_ЦЕНА
       FROM ИСТОРИЯ_ЦЕН
       WHERE Товар_Ид = @Товар_Ид and
       Дата_цена = (SELECT max(Дата_цена) AS Дата_цена
                    FROM ИСТОРИЯ_ЦЕН
                    WHERE Товар_Ид = @Товар_Ид and
                    Дата_цена <= @ЗАД_ДАТА)
-- Ответ
Товар_Ид            ЗАД_ДАТА                    УЧЕТН_ЦЕНА
2                  2000-01-12 00:00:00       5.0000
```

Если сравнить планы выполнения для решения задачи с курсором и без него, то можно увидеть, что выполнение только одного оператора `FETCH NEXT` для курсора по стоимости примерно в три раза больше, чем весь запрос из листинга 5.38.

5.11. Использование триггеров

Триггеры — это специальный тип хранимых процедур, присоединяемых к базовым таблицам или представлениям, которые выполняются при срабатывании операторов `INSERT`, `DELETE`, `UPDATE` для соответствующей таблицы или представления. Указанные операторы представляют собой транзакции, при работе которых должны выполняться определенные бизнес-правила по согласованию данных.

Триггеры применяются для следующих целей:

- обеспечение сложных деловых правил, порождающих сложные виды целостности базы данных, например вычисление полей, содержащих итоговые значения;
- выбор сложных значений по умолчанию;
- сбор информации о производимых изменениях.

Основная рекомендация по поводу триггеров состоит в том, что т. к. они в целом существенно замедляют быстродействие, то их нужно использовать только при действительной необходимости. Если имеется возможность реализовать бизнес-правило с помощью ограничений целостности, то обычно именно так и нужно поступать, потому что ограничения работают гораздо эффективнее. Например, каскадные изменения и удаления данных предпочтительнее реализовывать с помощью соответствующих опций ограничения внешнего ключа, а не триггера. И уж конечно, не нужно дублировать триггером уже существующие значения по умолчанию, правила и ограничения.

Нововведением SQL-сервера 2000 является возможность создания нескольких триггеров на один оператор. При этом можно указать, какой триггер будет работать первым, а какой последним. Предпочтительно, чтобы первым выполнялся триггер, у которого наибольшая вероятность отката транзакции.

5.11.1. Пример 1. Триггер на вычисление итоговых значений

В базе данных МАТ_УЧЕТ описаны требования на осуществление двух транзакций вместе с простейшими бизнес-правилами, которые должны при этом выполняться. Первая транзакция состоит в реализации операции по закупке товара. В этой транзакции участвуют две таблицы — ТОВАРЫ и ПРИХОДЫ, связанные FK-ограничением. Триггер здесь нужен для вычисления значения в поле Остаток таблицы ТОВАРЫ так, чтобы данные оставались согласованными. Предполагается, что могут происходить как повторные закупки товаров, для которых уже имеется Остаток в таблице ТОВАРЫ, так и закупки новых товаров, для которых, кроме ввода новой записи в таблицу ПРИХОДЫ, нужно еще добавить запись в таблицу ТОВАРЫ. Триггер для этой ситуации приведен в листинге 5.39.

Листинг 5.39. Пример триггера на вычисление итоговых значений

```
CREATE TRIGGER T_ПРИХОДЫ ON ПРИХОДЫ
INSTEAD OF INSERT AS
DECLARE @ОСТАТОК INT, @Товар_Ид INT, @Кол_прих INT, @Дата_прих DATETIME
--Проверяется наличие закупаемого товара в родительской таблице
IF EXISTS(SELECT I.Товар_Ид FROM INSERTED I INNER JOIN ТОВАРЫ T
 ON I.Товар_Ид = T.Товар_Ид)
/* Закупаемый товар присутствует в родительской таблице, в этом случае
просто вставляется запись в дочернюю таблицу, а в родительской таблице
изменяется значение остатка */
       BEGIN
       UPDATE ТОВАРЫ SET ОСТАТОК = ОСТАТОК + I.Кол_прих
       FROM ТОВАРЫ T INNER JOIN INSERTED I ON I.Товар_Ид = T.Товар_Ид
       INSERT INTO ПРИХОДЫ(Товар_Ид, Кол_прих, Дата_прих)
       SELECT Товар_Ид, Кол_прих, Дата_прих FROM INSERTED
       END
ELSE
       BEGIN
/* Закупается новый товар, которого нет в родительской таблице */
       SELECT @ОСТАТОК = Кол_прих FROM INSERTED
```

```
/* В родительскую таблицу вставляется запись, значение остатка равно
количеству закупки*/
       INSERT INTO ТОВАРЫ (Товар, Цена_Учетн, Остаток)
              VALUES('новый товар', 1000, @ОСТАТОК)
       SET @Товар_Ид = @@IDENTITY
/* Вставляется запись в дочернюю таблицу*/
       SELECT @Кол_прих = Кол_прих, @Дата_прих = Дата_прих FROM INSERTED
       INSERT INTO ПРИХОДЫ(Товар_Ид, Кол_прих, Дата_прих)
              VALUES(@Товар_Ид, @Кол_прих, @Дата_прих)
       END
```

В листинге 5.39 выполнен ввод данных в две таблицы, связанные FK-ограничением по полю со свойством `IDENTITY` для родительской таблицы. Причем, собственно, вводимые записи попадают в дочернюю таблицу `ПРИХОДЫ`, а родительская таблица `ТОВАРЫ` изменяется из `INSTEAD OF INSERT` триггера, который как бы обходит ограничение внешнего ключа.

Замечание

Чтобы не привязывать пример триггера к оператору `INSERT`, который вызовет его срабатывание, значения для названия нового товара и учетной цены в случае нового товара введены просто как константы — `VALUES('новый товар', 1000, @ОСТАТОК)`. На самом деле эти данные должны быть переданы с клиента. В следующем примере аналогичный момент разбирается подробнее.

5.11.2. Пример 2. Триггер на представление

Вторая транзакция базы данных `МАТ_УЧЕТ` заключается в выполнении операции по отпуску товара. В этой транзакции участвуют три таблицы — `ТОВАРЫ`, `РАСХОДЫ` и `РАСХОДЫ_ТОВАРЫ`, связанные двумя FK-ограничениями. При продаже товара добавляется несколько записей в таблицу `РАСХОДЫ_ТОВАРЫ` и одна запись в таблицу `РАСХОДЫ`, а также изменяется количество остатков для проданных товаров в таблице `ТОВАРЫ`.

Бизнес-правило, которое должно реализовываться при выполнении транзакции, состоит в том, что товары можно продавать только таким образом, чтобы их остаток после продажи не был отрицательным.

Для реализации этой транзакции создано представление с именем `VIEW_ВСТАВКА_РАСХОДА`. Разумеется, задача может быть решена и без использования представления, но именно этот способ выбран, прежде всего, для демонстрации примера триггера на представление. Кроме того, в случае использования представления вся бизнес-логика может быть собрана в одном месте, и легко выполнить FK-ограничение по полю со свойством счетчика. Представление используется еще и потому, что для оформления расходной

накладной клиентское приложение должно сформировать данные для двух базовых таблиц. Например, пусть оно формирует следующую последовательность операторов вставки для создания расходной накладной из двух товарных позиций:

```
INSERT INTO РАСХОДЫ (Дата_расх) VALUES('01-01-04')
SELECT @N = @@IDENTITY
INSERT INTO РАСХОДЫ_ТОВАРЫ(N_расх,Товар_Ид, Кол_расх) VALUES(@N, 25, 200)
INSERT INTO РАСХОДЫ_ТОВАРЫ(N_расх,Товар_Ид, Кол_расх) VALUES(@N, 26, 100)
```

Пусть создано представление `VIEW_ВСТАВКА_РАСХОДА` с помощью оператора

```
CREATE VIEW dbo.VIEW_ВСТАВКА_РАСХОДА AS
SELECT dbo.РАСХОДЫ.N_расх, dbo.РАСХОДЫ_ТОВАРЫ.Товар_Ид,
dbo.РАСХОДЫ_ТОВАРЫ.Кол_расх, dbo.РАСХОДЫ.Дата_расх
FROM dbo.РАСХОДЫ_ТОВАРЫ INNER JOIN dbo.РАСХОДЫ ON
dbo.РАСХОДЫ_ТОВАРЫ.N_расх = dbo.РАСХОДЫ.N_расх
```

Тогда три оператора вставки, в которых использованы две разные таблицы, можно заменить следующим образом:

```
INSERT INTO VIEW_ВСТАВКА_РАСХОДА (N_расх, Товар_Ид, Кол_расх, Дата_расх)
       VALUES(0, 25, 200, '01-01-04')
INSERT INTO VIEW_ВСТАВКА_РАСХОДА (N_расх, Товар_Ид, Кол_расх)
       VALUES(0, 26, 100)
```

Замечание

Разумеется, порядок операторов вставки в любом случае должен быть корректным, с точки зрения ограничения внешнего ключа, т. е. сначала вставка выполняется в таблицу `РАСХОДЫ`, а потом в таблицу `РАСХОДЫ_ТОВАРЫ`. В данном примере операторы вставки никак не объединены в транзакции, без чего реальное решение поставленной задачи является не совсем корректным. Средства управления транзакциями, с которыми очень тесно связана работа триггеров, рассмотрены в *главе 6*. Пока каждый оператор `INSERT` порождает отдельную транзакцию, как это принято по умолчанию.

При вставке в представление в списке `VALUES` первое значение равно нулю. Это заглушка, т. к. для поля `N_расх` родительской таблицы все равно в триггере будет сформировано значение, выбранное счетчиком. Нельзя написать оператор

```
INSERT INTO VIEW_ВСТАВКА_РАСХОДА (Товар_Ид, Кол_расх, Дата_расх)
       VALUES(25, 200, '01-01-04')
```

Это объясняется тем, что нельзя вставить `NULL`-значение в поле `N_расх` представления, потому что за этим полем стоит первичный ключ со свойством счетчика для базовой таблицы.

Текст триггера для второй транзакции приведен в листинге 5.40.

Листинг 5.40. Пример триггера на представление

```
CREATE TRIGGER T_VIEW ON VIEW_ВСТАВКА_РАСХОДА
INSTEAD OF INSERT
AS
DECLARE @ОСТАТОК INT, @N_расх INT, @Товар_Ид INT, @Кол_расх INT
-- Сколько осталось продаваемого товара
SELECT @ОСТАТОК = T.ОСТАТОК - I.Кол_расх
       FROM ТОВАРЫ T INNER JOIN INSERTED I ON I.Товар_Ид = T.Товар_Ид
/* Если для товарной позиции остаток товара меньше, чем нужно продать,
   то товарная позиция откатывается */
IF @ОСТАТОК < 0
       BEGIN
       RAISERROR ('ОСТАТОК ОТРИЦАТЕЛЕН ДЛЯ ВВЕДЕННОГО ТОВАРА', 16, 1)
       ROLLBACK TRANSACTION
       END
ELSE
/* Остаток товара больше, чем нужно продать, вставка товарной позиции
 выполняется */
       BEGIN
       RAISERROR ('МОЖНО ВСТАВИТЬ', 16, 1)
/* Проверяется, существует ли уже расходная накладная */
       IF EXISTS(SELECT I.N_расх FROM INSERTED I INNER JOIN РАСХОДЫ R
       ON I.N_расх= R.N_расх)
-- Вставка товарной позиции в дочернюю таблицу
              INSERT INTO РАСХОДЫ_ТОВАРЫ (N_расх, Товар_Ид, Кол_расх)
              SELECT N_расх, Товар_Ид, Кол_расх FROM INSERTED
       ELSE
       BEGIN
-- Вставка новой расходной накладной
       INSERT INTO РАСХОДЫ (Дата_расх) SELECT Дата_расх FROM INSERTED
-- Согласование FK-ограничения для поля типа счетчика
       SET @N_расх = @@IDENTITY
-- Вставка первой товарной позиции в новую расходную накладную
       SELECT @Кол_расх = Кол_расх, @Товар_Ид = Товар_Ид FROM INSERTED
       INSERT INTO РАСХОДЫ_ТОВАРЫ (N_расх, Товар_Ид, Кол_расх)
       VALUES(@N_расх, @Товар_Ид, @Кол_расх)
       END
-- Согласование остатков товара
       UPDATE ТОВАРЫ SET ОСТАТОК = ОСТАТОК - I.Кол_расх
       FROM ТОВАРЫ T INNER JOIN INSERTED I ON I.Товар_Ид = T.Товар_Ид
      END
```

Замечание

Разумеется, последний пример является демонстрационным. В реальном триггере бессмысленно использовать оператор генерации сообщения RAISERROR. Но в учебном примере может быть полезным увидеть, как выполнялись проверки.

5.11.3. Пример 3. Триггер для сохранения данных о сделанных изменениях

Таблица ИСТОРИЯ_ЦЕН создана в базе данных для того, чтобы хранить сведения об изменении учетной цены на товары для каждого товара. Эти данные должны заноситься в указанную таблицу в момент изменения (UPDATE ТОВАРЫ) любого значения учетной цены. В качестве даты изменения цены берется текущая дата. Текст программы приведен в листинге 5.41.

Листинг 5.41. Создание таблицы ИСТОРИЯ_ЦЕН

```
CREATE TRIGGER T_ИСТОРИЯ_ЦЕН ON ТОВАРЫ
FOR UPDATE
AS
DECLARE @Цена_учетн MONEY, @Товар_Ид INT
-- Проверка того, что учетная цена изменилась
IF UPDATE (Цена_учетн)
       BEGIN
       -- Выбираем старую цену для товара, у которого она изменена
       SELECT @Товар_Ид = D.Товар_Ид, @Цена_учетн = D.Цена_учетн
       FROM DELETED D INNER JOIN ТОВАРЫ T ON D.Товар_Ид = T.Товар_Ид
-- Вставка записи в таблицу ИСТОРИЯ_ЦЕН
       INSERT INTO ИСТОРИЯ_ЦЕН (Товар_Ид, Дата_цена, Цена_учетн)
       VALUES(@Товар_Ид, GETDATE(),@Цена_учетн)
END
```

5.12. Использование представлений

Представление дает возможность сохранить определенный ранее запрос в виде объекта базы данных для использования в будущем. То есть представление является некоторым логическим видом данных, хранящихся в базе. Создается виртуальная таблица, которая представляет данные из одной или нескольких базовых таблиц в нужном виде. Другими словами, представление — это просто сохраненный в базе данных оператор SELECT.

Представления полезны в следующих ситуациях:

- когда нужно сконцентрировать данные для пользователя, скрыв внутреннюю сложность базы данных;
- когда нужно осуществлять сложное управление доступом к данным, в том числе и на уровне столбцов;
- когда нужно подготовить данные, содержащиеся в нескольких базовых таблицах, для экспорта в другое приложение.

Использование представлений является обычной практикой, но в целом они замедляют производительность сервера. Правда, такое замедление редко бывает критичным, и преимущества перевешивают указанный недостаток. Все же кратко укажем некоторые причины его возникновения. Рассмотрим, например, два простейших запроса:

```
SELECT * FROM table_name
SELECT * FROM view_name
```

Какой из них быстрее? Утверждается, что тот, который использует таблицу, а не представление, хотя план выполнения у них будет эквивалентным. Разница в скорости выполнения возникает потому, что при работе представления тратится время на различные дополнительные действия, например на просмотр системных таблиц, и эти затраты времени не учитываются при подсчете стоимости запроса.

Представления, как и хранимые процедуры, при первом выполнении подвергаются оптимизации, после чего они кэшируются в буферный кэш для повторного использования. И все же если основной целью является достижение максимальной производительности, то лучше использовать хранимые процедуры. Нужный набор данных, являющийся выходным для хранимой процедуры, может использоваться для тех же целей, что и набор данных, генерируемый представлением.

5.12.1. Индексированные представления

Одним из самых серьезных нововведений для SQL-сервера 2000 является возможность создания индексов для представлений. По сравнению с обычным представлением *индексированное представление* является совершенно другим объектом, с точки зрения его физической реализации. Ведь индекс имеет смысл, только если отсортированные данные физически сохранены в базе данных. Поэтому индексированное представление называют еще *материализованным представлением*, т. к. его выходной набор физически сохраняется в базе данных и проиндексирован. Правильное применение индексированных представлений позволяет добиться в определенных случаях суще-

ственного роста производительности выполнения запросов, связанного с возможностью использования индекса для доступа к данным.

Чтобы обычное представление могло быть индексированным, для него нужно выполнение следующих условий:

- создавать индексированное представление можно только на основе тех таблиц, для которых тот, кто его создает, является их владельцем;
- нельзя использовать звездочку в операторе `SELECT` представления, а нужно явно указывать все столбцы;
- представление должно быть создано с опцией `WITH SCHEMABINDING`, которая запрещает изменение схемы базовых таблиц, использованных для построения представления;
- первый индекс, который создается для представления, обязательно должен быть уникальным кластерным индексом, следовательно, логическая структура оператора `SELECT` представления должна обеспечить в нем наличие данных, удовлетворяющих ограничению первичного ключа;
- если в операторе `SELECT` индексированного представления присутствует опция `GROUP BY`, то в нем обязательно должен также присутствовать столбец, создаваемый с помощью выражения `COUNT_BIG(*)`, хотя индекс на него не создается.

Индексированные представления рекомендуется применять для повышения производительности критических запросов, в которых присутствует соединение таблиц вместе со сложными условиями поиска и группировками. Следует помнить, что на индексированные представления в полной мере оказывают влияние все соображения, связанные с выбором компромисса между полезностью индекса и накладными расходами на его поддержание (*см. главу 4*).

Вручную подобрать подходящую схему индексированного представления и структуру его индексов обычно достаточно сложно. Рекомендуется использовать для этого Мастер Настройки Индексов в соответствующем режиме. Примеры индексированных представлений для сложных запросов приведены в *разд. 5.9*.

ГЛАВА 6

Транзакции и производительность

6.1. Основные понятия, связанные с транзакциями

Как известно, *транзакция* — это неделимая последовательность операторов манипулирования данными, выполняющаяся *как единое целое* (все или ничего) и переводящая базу данных *из одного целостного состояния в другое целостное состояние*.

Транзакции также являются *единицами восстановления* данных после сбоев, т. е. восстанавливаясь, система ликвидирует следы транзакций, не успевших завершиться в результате программного или аппаратного сбоя.

Кроме того, в многопользовательских системах транзакции служат для обеспечения *изолированной* работы отдельных пользователей, т. е. пользователям, одновременно работающим с одной базой данных, кажется, что они работают как бы в однопользовательской системе и не мешают друг другу.

Транзакция обладает четырьмя важными свойствами, известными по первым четырем буквам как *свойства АСИД* (ACID, Atomicity, Consistency, Isolation, Durability).

- *Атомарность* (Atomicity). Транзакция выполняется как атомарная операция — либо выполняется вся транзакция целиком, либо она целиком не выполняется.
- *Согласованность* (Consistency). Транзакция переводит базу данных из одного целостного состояния в другое целостное состояние. Внутри транзакции согласованность базы данных может нарушаться.
- *Изолированность* (Isolation). Транзакции различных пользователей не должны мешать друг другу.

- *Долговечность* (Durability). Если транзакция выполнена, то результаты ее работы должны сохраниться в базе данных, даже если в следующий момент произойдет сбой системы.

Более подробно вопросы, связанные с реализацией указанных свойств транзакций, рассматриваются далее.

6.1.1. Атомарность и согласованность транзакций

Свойство атомарности транзакций является вместе со свойством согласованности фактически ключевым в определении понятия транзакции. Поэтому это свойство представляет собой критерий, на основании которого процесс непрерывных изменений базы данных должен быть разделен на отдельные транзакции. Безусловной рекомендацией при этом считается то, что чем более короткие последовательности изменений удастся выделить и квалифицировать как атомарные, тем лучше, с точки зрения производительности. Далее будет объяснено, почему это так.

Свойство согласованности, или непротиворечивости, транзакций определяется понятием целостности базы данных, которое, как правило, поддерживается на уровне ограничений различных типов. Ограничение целостности — это некоторое утверждение, которое может быть истинным или ложным в зависимости от состояния базы данных. Ограничения целостности обычно вытекают из бизнес-правил предметной области. Примерами ограничений целостности могут служить следующие утверждения:

- возраст сотрудника должен находиться в диапазоне от 18 до 65 лет, если этот сотрудник является мужчиной, и от 18 до 60 лет, если этот сотрудник является женщиной;
- сотрудник обязан числиться в одном отделе;
- сумма накладной должна равняться сумме всех товарных позиций, входящих в накладную.

Некоторые из ограничений целостности являются ограничениями реляционной модели данных, например, такие как целостность сущностей, ссылочная целостность, в то время как другие ограничения целостности являются достаточно произвольными утверждениями. Но любое ограничение целостности является *семантическим* понятием, т. е. появляется как следствие определенных свойств объектов предметной области и/или их взаимосвязей.

База данных находится в *согласованном (целостном) состоянии*, если выполнены все ограничения целостности, определенные для базы данных. Таким образом, согласованность базы данных является ее формальным свойством. База данных понимает под "смыслом" хранимых данных весь набор ограни-

чений целостности. Если все ограничения выполнены, то СУБД считает, что данные корректны.

В данном определении важно подчеркнуть, что должны быть выполнены не вообще все ограничения предметной области, а только те, которые *определены* в базе данных. Для этого необходимо, чтобы СУБД *обладала* развитыми средствами поддержки ограничений целостности.

Вместе с понятием целостности базы данных возникает понятие *реакции системы на попытку нарушения целостности*. Имеется два типа реакции на попытку транзакции нарушить целостность базы данных:

- *отказ* выполнить "незаконную" операцию, при этом может генерироваться какое-нибудь сообщение для пользователя;
- выполнение компенсирующих действий, которые заключаются в том, что в транзакцию пользователя принудительно включается дополнительный код, поддерживающий согласованность данных. Для пользователя компенсирующие действия означают следующее: когда он изменяет, например, единственное значение в базовой таблице, при этом в базе данных может, скрыто от него, выполняться множество всяких проверок и изменений данных в других базовых таблицах.

В некоторых случаях система может не выполнять проверку нарушения ограничений, а сразу выполнять компенсирующие операции, если заранее известны результаты этой проверки, — ограничение *обязательно* будет нарушено. В этом случае необходимо сразу приступать к компенсированию возникшего нарушения.

Понятие ограничения целостности базы данных достаточно сложно формализовать. Даже способов классификации подобных ограничений существует несколько, и сложно указать, какой из них лучше. Подробнее ознакомиться с этими способами классификации можно в *приложении 2*.

6.1.2. Изолированность транзакций

Современные СУБД являются многопользовательскими системами, т. е. допускают одновременную параллельную работу большого количества пользователей. При этом пользователи не должны мешать друг другу. Поскольку логической единицей работы для пользователя является транзакция, то работа СУБД должна быть организована так, чтобы у пользователя складывалось впечатление, что его транзакции выполняются независимо от транзакций других пользователей. Простейший способ обеспечить такую иллюзию у пользователя состоит в том, чтобы все поступающие транзакции выстраивались в единую очередь и выполнялись строго по очереди. Такой способ не годится по очевидным причинам — теряется преимущество параллельной

работы. Таким образом, транзакции необходимо выполнять одновременно, но так, чтобы результат был таким же, как если бы транзакции выполнялись по очереди. Трудность состоит в том, что если не предпринимать никаких специальных мер, то данные, измененные транзакцией одного пользователя, могут быть, например, изменены транзакцией другого пользователя раньше, чем закончится транзакция первого пользователя. В результате в конце транзакции первый пользователь увидит не результаты своей работы, а неизвестно что.

Транзакции и параллелизм

Транзакция рассматривается как последовательность элементарных атомарных операций. Атомарность отдельной элементарной операции состоит в том, что СУБД гарантирует, что, с точки зрения пользователя, будут выполнены два условия: атомарность операции (все или ничего) и то, что во время этой операции не выполняются никакие другие операции других транзакций.

Элементарные операции различных транзакций могут выполняться в произвольной очередности (конечно, внутри каждой транзакции последовательность элементарных операций этой транзакции является строго определенной).

Набор из нескольких транзакций, элементарные операции которых чередуются друг с другом, называется *смесью транзакций*. Последовательность, в которой выполняются элементарные операции заданного набора транзакций, называется *графиком запуска* набора транзакций.

График запуска набора транзакций называется *последовательным*, если транзакции выполняются строго по очереди, т. е. элементарные операции транзакций не чередуются друг с другом. Если же график запуска набора транзакций содержит чередующиеся элементарные операции транзакций, то такой график называется *чередующимся*. При выполнении последовательного графика гарантируется, что транзакции выполняются правильно, т. е. при последовательном графике транзакции не "чувствуют" присутствия друг друга.

Два графика называются *эквивалентными*, если при их выполнении будет получен один и тот же результат *независимо от начального состояния базы данных*.

График запуска транзакции называется *сериализуемым,* если он эквивалентен какому-либо последовательному графику.

При выполнении двух различных последовательных (а следовательно, верных) графиков, содержащих один и тот же набор транзакций, могут быть получены *различные* результаты. Действительно, пусть элементарная операция транзакции `TRAN1` заключается в действии "Сложить X с 1", а транзакции `TRAN2` — "Удвоить X". Тогда последовательный график {`TRAN1`, `TRAN2`} даст

результат $2(X + 1)$, а последовательный график {TRAN2, TRAN1} — 2X + 1. Таким образом, может существовать несколько верных графиков запусков транзакций, приводящих к разным результатам, при одном и том же начальном состоянии базы данных.

Очевидно, что для заданного набора транзакций существует достаточно много различных графиков запуска. Обеспечение изолированности пользователей сводится к выбору подходящего сериального графика их запуска. Вместе с этим выбранный график запуска должен быть в некотором смысле оптимальным, т. е. он должен давать минимум некоторой стоимостной функции. В качестве стоимостной функции можно взять, например, суммарное время выполнения всех транзакций в наборе. Это время складывается из следующих компонентов:

- время ожидания начала транзакции, которое проходит с момента, когда транзакция инициирована, до момента, когда начала реально выполняться ее первая элементарная операция;
- суммарное время выполнения элементарных операций транзакции;
- суммарное время всех элементарных операций других транзакций, вклинившихся между элементарными операциями транзакции.

В реальной ситуации не только не известно, какие транзакции будут поступать в какие моменты времени, но не известна и длительность периода времени, охватывающего набор транзакций. Необходимо, следовательно, чтобы к любому моменту времени набор выполненных и выполняющихся в этот момент транзакций был бы правильным и не слишком далеким от оптимального.

Конфликты между транзакциями

Анализ проблем параллелизма показывает, что если не предпринимать специальных мер, то при работе в смеси нарушается свойство изолированности транзакций, и они реально не позволяют друг другу получать правильные результаты. Однако не всякие транзакции мешают друг другу. Очевидно, что транзакции не мешают друг другу, если они обращаются к *разным данным* или выполняются в *разное время*.

Транзакции называются *конкурирующими*, если они пересекаются по времени и обращаются к одним и тем же данным. В результате конкуренции за данные между транзакциями возникают *конфликты доступа* к данным следующих видов:

- *W-W* (Write–Write, Запись–Запись). Первая транзакция изменила объект и не закончилась. Вторая транзакция пытается изменить этот объект. Результат — *потеря результатов обновления*;

- *R-W* (Read–Write, Чтение–Запись). Первая транзакция прочитала объект и не закончилась. Вторая транзакция пытается изменить этот объект. Результат — *неповторяемое считывание*;
- *W-R* (Write–Read, Запись–Чтение). Первая транзакция изменила объект и не закончилась. Вторая транзакция пытается прочитать этот объект. Результат — *чтение "грязных" данных*.

Существуют и более сложные конфликты доступа, к которым относятся так называемые *фантомы* и *несовместимый анализ*. Их принципиальное отличие состоит в том, что они не могут возникать при работе с одним объектом. Для возникновения этих проблем требуется, чтобы транзакции работали с целыми *наборами* данных.

Далее приведены примеры, которые более подробно иллюстрируют типичные ситуации конфликтов между транзакциями.

Пример потери результатов обновления

Предположим, что две транзакции по очереди записывают некоторые данные в одну и ту же строку S и фиксируют изменения, что отображено в табл. 6.1.

Таблица 6.1. *Иллюстрация потери результатов обновления*

Транзакция `TRAN1`	Время	Транзакция `TRAN2`
Чтение $S = S_0$	`t1`	---
---	`t2`	Чтение $S = S_0$
Запись $S_1 \rightarrow S$	`t3`	---
---	`t4`	Запись $S_2 \rightarrow S$
---	`t5`	Фиксация транзакции
Чтение $S = S_2$	`t6`	---
Фиксация транзакции		---

В результате описанных действий после окончания обеих транзакций строка S содержит значение S_2, занесенное более поздней транзакцией `TRAN2`. Так как транзакция `TRAN1` ничего не знает о существовании транзакции `TRAN2`, то она, естественно, ожидает, что в строке S содержится значение S_1. Таким образом, транзакция `TRAN1` потеряла результаты своей работы.

Пример чтения "грязных" данных

Транзакция `TRAN2` изменяет данные в строке. После этого транзакция `TRAN1` читает измененные данные и работает с ними. Транзакция `TRAN2` откатывается и восстанавливает старые данные, как это показано в табл. 6.2.

Таблица 6.2. *Чтение "грязных" данных*

Транзакция TRAN1	**Время**	**Транзакция TRAN2**
---	t1	Чтение $S = S_0$
---	t2	Запись $S_1 \rightarrow S$
Чтение $S = S_1$	t3	---
Работа с прочитанными данными S_1	t4	---
---	t5	Откат транзакции, запись $S_0 \rightarrow S$
Фиксация транзакции	t6	---

В результате непонятно, с чем же работала транзакция TRAN1. Действительно, после отката транзакции TRAN2 должна восстановиться такая ситуация, как если бы транзакция TRAN2 вообще *никогда не выполнялась*. Получилось, что TRAN1 использовала данные, которых нет и *не было* в базе данных!

Пример неповторяемого считывания

Транзакция TRAN1 дважды читает одну и ту же строку. Между этими чтениями вклинивается транзакция TRAN2, которая изменяет значения в строке, как это показано в табл. 6.3.

Таблица 6.3. *Неповторяемое считывание*

Транзакция TRAN1	**Время**	**Транзакция TRAN2**
Чтение $S = S_0$	t1	---
---	t2	Чтение $S = S_0$
---	t3	Запись $S_1 \rightarrow S$
---	t4	Фиксация транзакции
Повторное чтение $S = S_1$	t5	---
Фиксация транзакции	t6	---

Так как транзакция TRAN1 ничего не знает о существовании транзакции TRAN2 и сама TRAN1 не меняет значения в строке S, то TRAN1 ожидает, что после повторного чтения строки S значение в ней будет тем же самым. Таким образом, транзакция TRAN1 работает с данными, которые, с ее точки зрения, самопроизвольно изменяются.

Пример появления фиктивных элементов (фантомов)

Эффект фиктивных элементов несколько сложнее предыдущих, потому что здесь за один шаг выполняется достаточно много операций, а именно происходит одновременное чтение нескольких строк, удовлетворяющих заданному условию. Итак, транзакция `TRAN1` дважды выполняет выборку строк по заданному условию. Между этими выборками вклинивается транзакция `TRAN2`, которая добавляет новую строку, удовлетворяющую условию отбора, как это показано в табл. 6.4.

Таблица 6.4. *Возникновение фантомов*

Транзакция `TRAN1`	**Время**	**Транзакция `TRAN2`**
Выборка *n* строк, удовлетворяющих условию	`t1`	---
---	`t2`	Вставка новой строки, удовлетворяющей условию
---	`t3`	Фиксация транзакции
Выборка *n* + 1 строк, удовлетворяющих условию	`t4`	---
Фиксация транзакции	`t5`	---

Транзакция `TRAN1` ничего не знает о существовании транзакции `TRAN2`, и т. к. сама она ничего не меняет в базе данных, то ожидает, что после повторной выборки будут получены те же самые строки. Но это не так — во второй выборке, выполненной `TRAN1`, появилась лишняя строка — фантом, вставленный транзакцией `TRAN2`.

Пример несовместимого анализа

Конфликт транзакций, называемый *несовместимым анализом*, имеет место, когда в смеси присутствуют одна длинная транзакция и одна короткая. Длинная транзакция выполняет некоторый анализ по всей таблице, например, подсчитывает общую сумму единиц товара заданного вида (яблок) на всех складах предприятия. Пусть для простоты на каждом складе находится одинаковое количество яблок (1 000 кг). Короткая транзакция в этот момент выполняет перенос 100 кг яблок с одного склада на другой так, что общее количество яблок по всем складам не изменяется. Описанная последовательность действий отображена в табл. 6.5.

Таблица 6.5. *Несовместимый анализ*

Транзакция TRAN1	Время	Транзакция TRAN2
Чтение данных с первого склада K_1 = 1 000 и суммирование, *SUM* = 1 000	t1	---
---	t2	Списание яблок со склада S_3, K_3: 1 000 → 900
---	t3	Оприходование яблок на складе S_1, K_1: 1 000 → 1 100
---	t4	Фиксация транзакции
Чтение данных со второго склада K_2 = 1 000 и суммирование, *SUM* = 2 000	t5	---
Чтение данных с третьего склада K_3 = 900 и суммирование, *SUM* = 2 900	t6	---
Фиксация транзакции	t7	---

Хотя обе транзакции все сделали корректно (и яблоки перевезены со склада на склад без потерь, и суммирование выполняется, как нужно), в результате оказалось, что сумма товара, равная 2 900 кг по всем складам, неправильная — должно быть 3 000 кг. То есть транзакция TRAN1 подсчитала общую сумму неверно. Более того, если транзакции по перевозке товаров выполнять непрерывно, то в данной ситуации следует ожидать, что вообще невозможно узнать, сколько же товара имеется на базе.

Очевидно, что ни один из описанных конфликтов между транзакциями не может возникнуть, если транзакции не мешают друг другу. Для этого они либо должны обращаться к разным данным, либо выполняться в разное время. Следовательно, имеется два потенциальных способа разрешения конкуренции между поступающими в произвольные моменты транзакциями.

- Можно "притормаживать" некоторые из поступающих транзакций настолько, насколько это необходимо для обеспечения правильности смеси транзакций в каждый момент времени, что означает выполнение конкурирующих транзакций *в разное время*. Этот метод реализуется путем использования *блокировок* различных видов или метода *временных меток*.
- Предоставлять конкурирующим транзакциям "разные" экземпляры данных, т. е. обеспечить работу конкурирующих транзакций *с разными вер-*

сиями данных путем использования данных из журнала транзакций на основе *механизма выделения версий*.

Из приведенных методов основным в настоящее время является использование блокировок. Далее механизм блокировок рассматривается более подробно.

Блокировки и разрешение тупиковых ситуаций

Основная идея блокировок заключается в том, что если для выполнения некоторой транзакции необходимо, чтобы какой-то объект не изменялся без ведома этой транзакции, то этот объект должен быть заблокирован. Последнее означает, что доступ к этому объекту со стороны других транзакций ограничивается на время выполнения транзакции, вызвавшей блокировку.

Различают два основных типа блокировок:

- *монопольные*, или *исключительные*, *блокировки* (X-locks, eXclusive locks, X-блокировки) — блокировки без взаимного доступа (блокировка записи). Если транзакция `TRAN1` блокирует объект при помощи X-блокировки, то всякий доступ к этому объекту со стороны других транзакций отвергается;
- *разделяемые блокировки* (S-locks, Shared locks, *S*-блокировки) — блокировки с взаимным доступом (блокировка чтения). Если транзакция `TRAN1` блокирует объект при помощи S-блокировки, то запросы со стороны других транзакций на X-блокировку этого объекта будут отвергнуты, а запросы со стороны других транзакций на S-блокировку этого объекта будут приняты.

Для предотвращения конфликтов доступ к объектам базы данных должен осуществляться в соответствии со следующим *протоколом доступа к данным*:

- прежде чем прочитать объект, транзакция должна наложить на этот объект S-блокировку;
- прежде чем обновить объект, транзакция должна наложить на этот объект X-блокировку. Если транзакция уже заблокировала объект S-блокировкой (для чтения), то перед обновлением объекта S-блокировка должна быть заменена X-блокировкой;
- если блокировка объекта транзакцией `TRAN2` отвергается оттого, что объект уже заблокирован транзакцией `TRAN1`, то транзакция `TRAN2` переходит в *состояние ожидания*. Транзакция `TRAN2` будет находиться в состоянии ожидания до тех пор, пока транзакция `TRAN1` не снимет блокировку объекта;
- X-блокировки, наложенные транзакцией `TRAN1`, сохраняются до ее завершения.

В реальных клиент-серверных платформах могут быть реализованы некоторые дополнительные типы блокировок, кроме классических X- и S-блоки-

ровок. В табл. 6.6 перечислены и кратко описаны основные и некоторые дополнительные типы блокировок, которые может реализовать SQL-сервер.

Таблица 6.6. *Основные типы блокировок SQL-сервера*

Тип блокировки и термины	Что означает и где применяется
Shared (S-блокировка) или разделяемая блокировка чтения	Для операций чтения, например в операторе `SELECT`
Exclusive (X-блокировка) или исключительная блокировка изменения	Для операций модификации данных, например в операторах `INSERT`, `UPDATE` или `DELETE`
Update (U-блокировка)	Накладывается на изменяемые ресурсы. Предотвращает типичные мертвые блокировки, когда много соединений читают, блокируют и могут изменить ресурсы позже
Intent (I-блокировка) или блокировка намерения	Используется для установления иерархии блокировок: INTent shared (IS), INTent (IX) и shared с INTent (SIX)
Schema	Используется для операций по изменению схемы данных: (Sch-M) или (Sch-S)
Bulk Update (BU)	Используется для команды `BULK INSERT` или с хинтом `TABLOCK`

Замечание

В табл. 6.6 приведены только некоторые из возможных типов блокировок SQL-сервера, которые напрямую соответствуют общим стандартам серверов реляционных баз данных. Реально на сервере осуществляется гораздо больше типов блокировок, представляющих собой чаще всего некую комбинацию базовых типов. Полный перечень этих комбинаций можно узнать из информации о содержимом столбцов системной таблицы `syslockinfo`, в частности, все они описаны в столбце этой таблицы под названием `req_mode`.

Важно понимать не только, каким способом блокируется тот или иной объект, но и объем заблокированных данных. Современные СУБД, как правило, поддерживают минимальный объем блокировки на уровне строк базовых таблиц или страниц данных. Отличие этих двух уровней блокировки состоит в том, что при постраничной блокировке фактически блокируется не только отдельная строка, но и примыкающие к ней строки всей физической страницы данных. Реализация только страничной блокировки считается большим недостатком СУБД. Этим недостатком обладают, например, MS Access и MS SQL-сервер до версии 7.0. MS SQL-сервер 2000 и Oracle поддерживают построчную блокировку.

При блокировке на уровне строк в случае, если все транзакции в смеси подчиняются протоколу доступа к данным, решаются почти все проблемы параллелизма, кроме проблемы фантомов. В целом можно рассматривать блокировки объектов базы данных на следующих уровнях:

- блокировка самой базы данных;
- блокировка файлов базы данных;
- блокировка таблиц базы данных;
- блокировка страниц;
- блокировка отдельных строк таблиц;
- блокировка отдельных полей (для этого нужно использовать индексы).

Кроме того, можно блокировать индексы, заголовки таблиц или другие объекты.

Иногда для обозначения уровня блокируемых данных используется термин *гранулярность* (от англ. *granularity — крупность разбиения*). Более высокий уровень гранулярности означает, что блокируется более крупный объект. Для обозначения этого процесса иногда применяется такой термин, как *эскалация блокировок* (от англ. *escalation — увеличение масштабов*).

Переход от блокировки более низкого уровня к блокировке более высокого уровня сервер выполняет автоматически. Этот переход к блокировке более высокого уровня SQL-сервер считает оправданным в случаях, когда затраты на обслуживание большого количества блокировок начинают превышать преимущества от того, что заблокировано меньшее количество данных. Например, проверка графа ожидания транзакций, размеры которого напрямую зависят от количества наложенных блокировок, представляет собой достаточно накладную процедуру.

SQL-сервер может повысить уровень блокировки прямо в ходе выполнения транзакции до уровня таблицы в том случае, если перед этим блокировались отдельные записи. Необходимость повышения уровня блокировки объекта SQL-сервер определяет исходя из ситуации, когда превышается максимально допустимое количество блокировок, наложенных либо одной транзакцией, либо наложенных на один индекс или одну таблицу. Также возможно повышение уровня блокировки, если для обслуживания блокировок используется большее количество памяти, чем это допустимо. В подобных случаях сервер выбирает наиболее подходящую таблицу и пытается повысить уровень ее блокировки до табличного уровня. Если это невозможно из-за несовместимости блокировок частей таблицы, то работа продолжается без повышения уровня блокировки объекта, исключая, таким образом, возможность взаимоблокировки.

Вручную изменить уровень блокировки объекта можно либо с помощью подсказок оптимизатору (`SELECT` уровня таблицы) на время выполнения одной транзакции, либо для всего соединения установкой флажка трассировки `DBCC TRACEOFF(1211)`. Как обычно, использовать подобные средства не рекомендуется.

При использовании блокировок объектов разной величины возникает проблема обнаружения уже наложенных блокировок. Если транзакция `TRAN1` пытается заблокировать таблицу, то необходимо иметь информацию, не наложены ли уже блокировки на уровне строк этой таблицы, несовместимые с блокировкой таблицы. Для решения этой проблемы используется протокол преднамеренных блокировок, являющийся расширением протокола доступа к данным. Суть этого протокола в том, что перед тем как наложить блокировку на объект, например на строку таблицы, необходимо наложить специальную *блокировку намерения* (I-блокировку) на объекты, в состав которых входит блокируемый объект, т. е. на таблицу, содержащую эту строку, на файл, содержащий эту таблицу, и на базу данных, содержащую этот файл. Тогда установка преднамеренной блокировки всей таблицы будет свидетельствовать о наличии блокировки строк таблицы, и для другой транзакции, пытающейся блокировать целую таблицу, не нужно проверять наличие блокировок отдельных строк.

Другим способом блокирования является блокировка не объектов базы данных, а условий, которым могут удовлетворять объекты. Такие блокировки называются *предикатными блокировками*.

Идея предикатных блокировок состоит в том, что любая операция над реляционной базой данных может быть задана некоторым условием, т. е. в операции указывается не конкретный набор объектов базы данных, над которыми нужно выполнить эту операцию, а условие, которому должны удовлетворять нужные объекты. При подобном подходе удобным способом было бы указывать S- или X-блокировку именно этого условия. Однако при попытке использовать этот метод в реальной СУБД возникает трудность определения совместимости различных условий.

В некоторых случаях предикатные блокировки могут быть весьма эффективным средством. Так, проблема фантомов легко решается с использованием предикатных блокировок, потому что вторая транзакция не может вставить новые строки, удовлетворяющие уже заблокированному условию.

Заметим, что блокировка всей таблицы в каком-либо режиме фактически есть предикатная блокировка, поскольку каждая таблица имеет предикат, определяющий, какие строки содержатся в таблице, и блокировка таблицы есть блокировка предиката этой таблицы.

Общим правилом, связанным с блокировками, можно считать следующее — чем крупнее объект блокировки, тем меньше остается возможностей для па-

раллельной работы после наложения блокировки. С другой стороны, достоинством блокировок крупных объектов является уменьшение накладных расходов системы и устранение проблем, не решаемых с использованием блокировок менее крупных объектов. Например, очевидно, что использование монопольной блокировки на уровне таблицы решает проблему фантомов.

В целом блокировки подходящих типов объектов нужного уровня способны решать большинство проблем параллельного доступа к данным. Однако при использовании механизма блокировок появляются так называемые *тупики* (deadlocks). Состояние тупика характеризуется тем, что две или более транзакции пытаются заблокировать одни и те же объекты и бесконечно долго ожидают друг друга. Общий вид тупика описан в табл. 6.7.

Таблица 6.7. *Элементарные операции, иллюстрирующие состояние тупика*

Транзакция `TRAN1`	**Время**	**Транзакция `TRAN2`**
Блокировка объекта `O1` успешна	`t1`	---
---	`t2`	Блокировка объекта `O2` успешна
Блокировка объекта `O2` конфликтует с блокировкой, наложенной транзакцией `TRAN2`	`t3`	---
Ожидание...	`t4`	Блокировка объекта `O1` конфликтует с блокировкой, наложенной транзакцией `TRAN1`
Ожидание...	`t5`	Ожидание...

Как видно из табл. 6.7, ситуация тупика может возникать при наличии не менее двух транзакций, каждая из которых выполняет не менее двух операций. На самом деле в тупике может участвовать много транзакций, ожидающих друг друга.

Поскольку нормального выхода из тупиковой ситуации не существует, то ее необходимо распознавать и устранять. Методом разрешения тупиковой ситуации является откат одной из транзакций, называемой *транзакцией-жертвой*, таким образом, чтобы другие транзакции продолжили свою работу. После разрешения тупика транзакцию, выбранную в качестве жертвы, можно повторить заново.

Имеются два принципиально различных подхода к обнаружению тупиковой ситуации и выбору транзакции-жертвы.

1. СУБД не следит за возникновением тупиков. Транзакции сами принимают решение, быть ли им жертвой. Этот метод является наиболее простым и не

требует дополнительных ресурсов. Для транзакций задается время ожидания (или число попыток), в течение которого транзакция пытается установить нужную блокировку. Если за указанное время (или после указанного числа попыток) блокировка не завершается успешно, то транзакция откатывается (или генерируется ошибочная ситуация). За простоту этого метода приходится платить тем, что транзакции-жертвы выбираются, вообще говоря, случайным образом. В результате из-за одной простой транзакции может откатиться очень дорогая транзакция, на выполнение которой уже потрачено много времени и ресурсов.

2. Противоположной является такая стратегия, когда за возникновением тупиковой ситуации следит сама СУБД, и она же принимает решение, какой транзакцией пожертвовать. Второй способ характерен для промышленных СУБД, например для Oracle или MS SQL-сервер. В этом случае строится *граф ожидания транзакций*, представляющий собой ориентированный двудольный граф, в котором существует два типа вершин — вершины, соответствующие транзакциям, и вершины, соответствующие объектам захвата. Ситуация тупика возникает, если в графе ожидания транзакций имеется хотя бы один цикл. В этом случае одну из транзакций, попавших в цикл, необходимо откатить, причем СУБД сама может выбрать эту транзакцию в соответствии с некоторыми стоимостными соображениями, например, самую короткую или обладающую минимальным приоритетом.

6.1.3. Долговечность транзакций и восстановление данных

Требование *долговечности транзакций* состоит в том, что данные, зафиксированные транзакцией, должны сохраняться в базе данных, даже если в следующий момент произойдет ее сбой. Казалось бы, самый простой способ обеспечения такой гарантии — это во время каждой операции сразу записывать все изменения на дисковые носители. Однако такой способ не является удовлетворительным, т. к. имеется существенное различие в скорости работы с оперативной и дисковой памятью. Единственный способ достичь приемлемой скорости работы состоит в буферизации или кэшировании страниц данных в оперативной памяти. На практике это означает, что данные попадают во внешнюю долговременную память, т. е. на диск, не сразу после внесения изменений, а через некоторое (достаточно большое) время. Тем не менее что-то во внешней памяти должно оставаться, т. к. иначе неоткуда получить информацию для восстановления. Выход из этого противоречия достигается путем ведения так называемого *журнала транзакций*, который представляет собой параллельную запись в отдельный файл сведений о любой из транзакций, модифицирующей базу данных. В SQL-сервере реализован журнал транзакций с опережающей записью, основным принципом которого по согласо-

ванию выталкивания из буфера страниц журнала транзакций и страниц базы данных является протокол журнализации *Write Ahead Log* (WAL) — "*пиши сначала в журнал*".

Если предельно кратко описать механизм работы журнала транзакций, то суть его состоит в следующем. Сначала команда, инициирующая транзакцию, например `BEGIN TRAN`, записывается в журнал, находящийся в кэше, при этом запоминается уникальный идентификатор транзакции и ее номер в последовательности транзакций. Далее в кэше модифицируются страницы журнала транзакций. По окончании транзакции команда `COMMIT TRAN` записывается в журнал в кэше, после чего кэшированный журнал сбрасывается на диск.

Данные заносятся из кэша в базу данных отдельно от информации из журнала транзакций. Страницы базы данных, содержимое которых в буфере отличается от содержимого на диске, называются *грязными* (dirty) *страницами*. Запись "грязных" страниц из буфера на диск называется *выталкиванием страниц во внешнюю память*. Выталкивание страниц данных из кэша на диск называется *процессом контрольной точки* (checkpoint). Напомним, что страницы данных, которые были изменены только в буферном кэше, но еще не записаны на диск, носят название грязных страниц. Итак, сначала на диск записываются страницы журнала, потом страницы данных, после чего на диск записывается маркер контрольной точки, являющийся необходимым механизмом процедуры восстановления базы данных. В момент завершения процесса контрольной точки база данных и журнал транзакций на диске полностью синхронизированы. Процесс контрольной точки выполняется в следующих ситуациях:

- превышен интервал восстановления сервера;
- в кэше не осталось свободного места, и нужно выгрузить старые страницы данных для того, чтобы загрузить новые;
- выполнена команда `CHECKPOINT`;
- выполняется резервное копирование журнала транзакций.

Обычно свойство долговечности не является абсолютным, т. к. допускаются вложенные транзакции. Так, если транзакция `TRAN2` стартовала внутри транзакции `TRAN1` и для транзакции `TRAN2` подана команда ее завершения, то фиксация данных транзакции `TRAN2` является условной, поскольку внешняя транзакция `TRAN1` может откатиться. Результаты работы внутренней транзакции `TRAN2` будут окончательно зафиксированы, только если будет зафиксирована внешняя транзакция `TRAN1`.

Как уже говорилось, свойство долговечности транзакций достигается путем внесения в систему избыточности хранения данных, которая позволяет вос-

становить систему после сбоя. Основную роль при этом играет журнал транзакций.

Восстановление базы данных может производиться в случаях, когда происходит:

- ❒ индивидуальный откат транзакции;
- ❒ мягкий сбой системы (аварийный отказ программного обеспечения);
- ❒ жесткий сбой системы (аварийный отказ аппаратуры).

Минимальным требованием, гарантирующим возможность восстановления последнего согласованного состояния базы данных, является выталкивание при фиксации транзакции во внешнюю память журнала всех записей об изменении базы данных этой транзакцией.

Индивидуальный откат транзакции выполняется при помощи журнала транзакций.

Восстановление системы после мягкого сбоя осуществляется как *часть процедуры перезагрузки* системы. При перезагрузке системы транзакции проходят процедуру идентификации для выявления завершившихся и прерванных в результате сбоя транзакций. Транзакции, успешно завершившиеся до наступления сбоя, данные которых отсутствуют в базе данных, повторяются заново. Транзакции, не успевшие завершиться к моменту сбоя, данные которых имеются в базе данных, откатываются.

При жестком сбое база данных на диске нарушается физически. Основой восстановления в этом случае является журнал транзакций и *архивная копия базы данных*. Архивная копия базы данных должна создаваться периодически, а именно с учетом скорости наполнения журнала транзакций.

Восстановление начинается с обратного копирования базы данных из архивной копии. Затем выполняется просмотр журнала транзакций для выявления всех транзакций, которые закончились *успешно* до наступления сбоя. Транзакции, закончившиеся откатом до наступления сбоя, можно не рассматривать. После этого по журналу транзакций в прямом направлении повторяются все успешно законченные транзакции. При этом нет необходимости отката транзакций, прерванных в результате сбоя, т. к. изменения, внесенные этими транзакциями, отсутствуют после восстановления базы данных из резервной копии.

Наиболее плохим случаем является ситуация, когда физически разрушены и база данных, и журнал транзакций. В этом случае единственное, что можно сделать, — это восстановить состояние базы данных на момент последнего резервного копирования. Для того чтобы не допустить возникновения такой ситуации, базу данных и журнал транзакций обычно располагают на *физически* разных дисках, управляемых физически разными контроллерами.

6.2. Представление о способах оценивания производительности транзакций

В компьютерной индустрии термин *транзакция* может означать почти любой вид взаимодействия или обмена информацией. Однако в мире бизнеса транзакция имеет вполне определенный смысл: коммерческий обмен товарами, услугами или деньгами. В настоящее время практически все бизнес-транзакции выполняются с помощью компьютеров. Наиболее характерными примерами систем обработки транзакций являются системы управления учетом, системы резервирования авиабилетов и банковские системы. Таким образом, необходимость стандартов и тестовых пакетов для оценки таких систем все больше усиливается.

Если в процессе разработки конкретного приложения проведение тестирования быстродействия отдельных запросов и поиск альтернативных вариантов обычно может быть выполнен самим разработчиком, то протестировать интегрированную производительность выполнения смеси транзакций крайне затруднительно. Процесс проведения теста представляет собой обычно сравнение чего-то с чем-то. В данном случае можно попытаться сравнить по сопоставимым параметрам свое программное решение с аналогичными разработками, основываясь на том факте, что наиболее характерными примерами систем обработки транзакций являются системы управления учетом, системы резервирования авиабилетов и банковские системы. Поэтому не лишним будет представление о современных подходах к способам подобного тестирования.

Чтобы решать такие проблемы, была создана организация TPC (Transaction Processing Performance Council, Совет по оценке производительности обработки транзакций), являющаяся некоммерческой бесприбыльной организацией. Любая компания или организация может стать членом TPC после уплаты соответствующего взноса. На сегодня членами TPC являются практически все крупнейшие производители аппаратных платформ и программного обеспечения для автоматизации коммерческой деятельности.

Основной задачей TPC является создание тестовых пакетов для оценки производительности систем обработки транзакций и баз данных, а также для распространения объективных, проверяемых данных в промышленности. Наличие подобных тестов позволяет получать и распространять объективную информацию о быстродействии баз данных. TPC предъявляет крайне жесткие требования к тестированию надежности баз данных и их устойчивости ко всевозможным сбоям. Кроме того, предусматривается, что полученные при тестировании результаты должны пройти независимую проверку.

Возможно, наиболее важным аспектом тестов TPC является требование полного раскрытия всех деталей проведения испытаний. Информация, содержа-

щаяся в отчете о проведении испытаний, должна обеспечивать возможность полного воспроизведения результатов. TPC также следит за тем, чтобы до появления ее официального отчета никакая информация не публиковалась. Каждый отчет проверяется советом технических экспертов, который состоит из ведущих специалистов по базам данных.

TPC публикует отчеты о спецификациях тестовых пакетов, которые регулируют вопросы, связанные с работой тестов, и гарантируют, что покупатели имеют объективные значения показателей для сравнения производительности различных вычислительных систем. В частности, эти отчеты включают конфигурацию системы, методику калькуляции цены, диаграммы значений производительности и документацию, показывающую, что тест соответствует требованиям АСИД.

Обычно при описании конфигурации системы приводятся блок-схемы подключения каналов и устройств ввода/вывода, детальный список аппаратных средств и программного обеспечения, включающий номера составных частей, их описание и номер версии. В цену системы включены стоимость аппаратных средств и программного обеспечения, а также стоимость запасных частей, необходимых для эксплуатации системы в течение 5 лет; объем внешней памяти системы должен обеспечивать хранение информации о транзакциях за период 30, 90 или 180 дней.

К настоящему времени TPC создала несколько тестовых пакетов для обеспечения объективного сравнения различных систем обработки транзакций и планирует создать новые оценочные тесты.

К подобным тестам относится, например, тест, называемый TPC-C, который, являясь стандартным отраслевым тестом, позволяет оценивать эффективность функционирования систем с оперативной обработкой транзакций. При проведении теста TPC-C проверяется широкий спектр функциональных возможностей баз данных, включая быстродействие и масштабируемость приложений, выполнение запросов, обновление транзакций и обработку минигрупп транзакций с постановкой в очередь. Многие специалисты в области информационных технологий считают результаты теста TPC-C объективной и надежной характеристикой быстродействия реально существующих OLTP-систем.

В тесте TPC-C измеряется производительность (количество бизнес-транзакций в единицу времени), достигаемая в среде, моделирующей поступление и распределение потоков заказов. В частности, в ходе теста оценивается, сколько транзакций, связанных с обработкой новых заказов, генерируется системой в минуту наряду с выполнением транзакций четырех других типов, относящихся, например, к прохождению платежей, обновлению статуса заказа, поставкам материалов и изменению уровня запасов. Независимые аудито-

ры подтверждают достоверность результатов тестирования, предоставляя соответствующий полный отчет в TPC. С полными отчетами о результатах проведенных тестов можно ознакомиться на Web-сайте TPC.

Создание тестов для оценки производительности систем обработки транзакций является достаточно сложным и неоднозначным процессом, потому что невозможно охарактеризовать нагрузку, генерируемую приложением базы данных, без детальной информации о том, что же в действительности приложение делает. Даже имея подобные сведения, приходится допускать множество эвристических предположений о том, как сама СУБД будет обращаться к данным, насколько эффективным может оказаться дисковый кэш СУБД при выполнении определенных транзакций, или даже о том, какова может быть смесь транзакций. Все вместе данные сведения формируют понятие *рабочей нагрузки*, которое является одним из ключевых в данной сфере.

На сегодня в промышленности приняты следующие типы характеристик нагрузки, генерируемой приложением базы данных: "легкая", "средняя", "тяжелая" и "очень тяжелая". Например, нагрузками "средней" тяжести считаются транзакции, определенные стандартом теста TPC-C.

Основным классом приложений, которые попадают в категорию "очень тяжелой" нагрузки, являются системы поддержки принятия решений. Из-за очень больших различий в природе запросов к системе поддержки принятия решений администраторы баз данных или самой СУБД сталкиваются с очень большими проблемами по обеспечению широкомасштабной, полезной оптимизации. Запросы к системе поддержки принятия решений часто приводят к формированию существенно большего числа запросов к нижележащей системе из-за необходимости выполнения многонаправленных соединений, агрегирования или сортировки данных, поэтому здесь также необходимы свои специфические тесты.

Считается, что SQL-сервер обеспечивает для широкого круга заказчиков доступную по цене, высокопроизводительную и масштабируемую технологию баз данных, вполне удовлетворяя современным требованиям, предъявляемым к обработке транзакций, обладая к тому же существенным резервом для реализации быстро развивающихся приложений. На текущий момент времени SQL-сервер устойчиво демонстрирует самые лучшие результаты в категории цена-быстродействие теста TPC-C, занимая в ней 40 верхних позиций.

Кроме TPC-C теста, можно упомянуть также о тесте TPC-H, который проводится для систем поддержки принятия решений. В тесте TPC-H моделируется система поддержки принятия решений, обеспечивающая обработку большого объема информации, синхронизированная с сетевыми производственными базами данных. В нем используются сложные специальные запросы, разработанные для получения ответов на возникающие в ходе бизнес-процессов ре-

альные вопросы, связанные с ценообразованием и продвижением товара, снабжением и спросом, доходом и прибылью, а также расширением производства и выходом на новые рынки. Скорость генерации документов стандартной отчетности демонстрируется при помощи другого теста, называемого TPC-R.

Следует отметить, что, как и любой другой тест, ни один тест TPC не может измерить производительность системы, которая применима для любой возможной среды обработки транзакций, но эти тесты действительно могут помочь пользователю справедливо сравнивать похожие системы. Однако когда пользователь планирует выбор конкретной клиент-серверной платформы с целью ее покупки, нужно ясно понимать, что никакой тест не может быть приспособлен именно под его конкретную прикладную задачу.

При создании любого приложения необходимо хотя бы приблизительно оценить минимально необходимую конфигурацию системы на основе анализа основных транзакций. При выборе конфигурации системы приходится рассматривать комбинацию достаточно сложных компонентов, к которым относятся аппаратные средства, операционная система, сеть, сама СУБД и конкретное приложение. Из-за сложности анализа комбинации этих объектов обычно невозможно определить, будет ли данная система способна поддерживать требуемую нагрузку на приемлемом уровне быстродействия. В процессе получения подобной оценки рассматриваются известные ограничения, которым удовлетворяют отдельные разнородные компоненты системы, и затем производится их сравнение с минимальными потребностями в соответствии с поставленной задачей.

Тем не менее обычно возможно сделать некоторые предположения и затем грубо проанализировать основные транзакции, чтобы определить, при каких конфигурациях система не будет способной обработать эти основные транзакции. Хотя такой подход оправдан только для относительно простых приложений с одной или двумя основными транзакциями, он дает неплохое начальное приближение и для больших и более сложных приложений. Чтобы дать минимальное представление о способах его применения, приведем два соответствующих примера, взятых из информационно-аналитических материалов Центра Информационных Технологий [23].

6.2.1. Пример 1. Генерация базы данных

В качестве примера рассматривается приложение, которое в течение четырех часов каждую ночь должно заново создавать базу данных, состоящую из двух таблиц, одна из которых содержит 10 млн записей по 2 Кбайта каждая, а другая — 40 000 записей размером примерно по 1 Мбайт каждая. Таким образом, общий объем первой таблицы составляет около 20 Гбайт, а второй —

40 Гбайт. Предполагается, что индексация выполняется с помощью отдельной операции. Тогда для создания первой таблицы требуется выполнить 10 млн дисковых операций по 2 Кбайта за 4 часа, или 695 операций в секунду. Для теста выбран дисковый накопитель, который может обеспечить выполнение примерно 60 операций по 2 Кбайта произвольного доступа и примерно 400 операций в секунду в режиме последовательного доступа. Отсюда ясно, что необходимо иметь, по крайней мере, два параллельно работающих диска, если создание таблицы выполняется последовательными операциями, или 12 параллельно работающих дисков, если выполняются операции произвольного доступа. Обычно сама таблица создается последовательными операциями, в то время как индексы часто создаются операциями произвольного доступа.

Создание второй таблицы требует размещения на диске 40 000 записей по 1 Мбайт каждая (всего 40 Гбайт). Хотя конкретные СУБД отличаются обработкой и размещением BLOB-объектов (Binary Large Object, двоичные объекты большого объема), все они почти наверняка будут использовать для записи каждого элемента данных только один адрес, и в результате такие записи будут выполняться последовательными операциями. Таким образом, поскольку в нашем случае каждая запись будет состоять примерно из 512 физических записей на диск, в этой работе будут доминировать операции последовательного доступа к диску (даже если каждая запись хранится отдельно, потребуется не более одного позиционирования головок на все 512 обращений к диску). Для создания этой таблицы требуется выполнить 20 480 000 операций записи на диск со скоростью примерно 1 450 операций в секунду. Для достижения этой скорости по крайней мере 4 диска должны осуществлять запись одновременно. Поскольку для размещения 40 Гбайт данных потребуется не менее 14 дисковых накопителей емкостью 2,9 Гбайт, то для достижения требуемой производительности не возникает проблем с обеспечением достаточного количества дисков, но требование параллельной записи на 4 диска диктует, чтобы использовался некоторый механизм, позволяющий расщепить доступ по дискам. Поэтому минимально требуемая конфигурация должна обеспечивать четырехкратное расщепление.

Необходимо помнить, что при достижении таких скоростей дискового ввода/вывода система должна затрачивать также примерно 1,7 мс процессорного времени на каждую операцию с диском. Создание первой таблицы требует выполнения 695 дисковых операций в секунду, или 1 181 мс на каждую секунду процессорного времени. Для создания второй таблицы требуется выполнить 1 425 дисковых операций в секунду, или примерно 2 425 мс на каждую секунду процессорного времени. В сумме это составляет примерно 3 606 мс на каждую секунду процессорного времени. Таким образом, можно сделать вывод о том, что четыре процессора SuperSPARC частотой 50 МГц должны заниматься исключительно обслуживанием дисков.

Кроме того, следует принять во внимание тот факт, что все данные для формирования таблиц базы данных откуда-то должны поступить, а обработка приходящих данных также требует некоторого процессорного времени. Предположим, что данные поступают по сети FDDI (Fiber Distributed Data Interface, распределенный интерфейс передачи данных по волоконно-оптическим каналам) с использованием подходящих протоколов, и мы ожидаем получить всего 60 Гбайт данных за 4 часа, т. е. скорость поступления данных составляет примерно 4,2 Мбайт/с. Если взять максимальный размер пересылки по FDDI равным 4 500 байт, это составит примерно 949 пакетов в секунду. На обработку каждого пакета процессор должен затратить 0,4 мс, поэтому обработка всего потока обойдется нам в 380 мс процессорного времени на каждую секунду. Эти накладные расходы могут еще увеличиться, если данные не занимают полностью 4 500-байтового пакета.

Для обеспечения дискового и сетевого ввода/вывода достаточно четырех процессоров. Однако в конфигурацию системы необходимо включить, по крайней мере, еще один дополнительный процессор, чтобы выполнять любую другую работу. Следует отметить, что для работы самой СУБД, а также для любой другой обработки, которая должна выполняться во время ночных перезагрузок базы данных, требуется некоторое процессорное время. Более того, необходимо предусмотреть небольшой поправочный коэффициент, поскольку должен быть некоторый дополнительный расход системных ресурсов, хотя бы на обслуживание собственных потребностей, что неизбежно приведет к тому, что средняя производительность окажется меньше, чем было вычислено.

Таким образом, в состав конечной конфигурации системы следует включить 6 процессоров. Поскольку минимальный объем дискового пространства составляет 40 Гбайт, только для хранения данных требуется четырнадцать дисков емкостью 2,9 Гбайт, так что в конфигурацию системы необходимо включить, по крайней мере, 16 накопителей, при этом отдельный диск требуется для размещения системы и журнала СУБД. Поскольку в обращениях к дисковой подсистеме доминируют последовательные операции, при конфигурировании дисков СУБД необходимо подключать не более 4 дисков к одному главному адаптеру SCSI (Small Computer Systems Interface, интерфейс малых компьютерных систем). Необходимость четырехкратного расщепления увеличивает общее количество дисков до восемнадцати.

Сам по себе процесс загрузки СУБД не предъявляет каких-либо особых требований к объему оперативной памяти. Базовые 64 Мбайт оперативной памяти на каждый процессор дают общий ее объем, равный 384 Мбайт, что представляет собой вполне умеренную оценку. Этот объем близок к рекомендованному — 1 % от размера базы данных составляет 400 Мбайт.

6.2.2. Пример 2. Обслуживание приложения

Рассмотрим приложение, обеспечивающее информацию о состоянии заказов для телефонного бюро обслуживания, в котором имеется одна ключевая транзакция: запрос о состоянии счета/заказа. Исходная информация, необходимая для расчетов, такова:

- *требуемая производительность транзакции.* В течение дня система должна поддерживать 120 пользователей, выполняющих запросы счет/заказ с максимальным временем ответа 2 с. Обычно операторы обрабатывают примерно один телефонный звонок за две минуты;
- *анализ обстановки, связанной с конкуренцией транзакций.* Оформление новых заказов выполняет отдельная подсистема таким образом, что новые заказы поступают в систему запросов по ночам после резервного копирования, следовательно, транзакция оформления нового заказа не конкурирует с транзакцией запроса о состоянии счета/заказа;
- *размер страницы данных СУБД.* В нашем случае этот размер принимается равным 2 Кбайта;
- *расчет объема данных, вовлеченных в транзакцию.* В среднем имеется 7 000 активных счетов и примерно 8 500 неоплаченных заказов. Всего в текущий момент времени имеется 260 000 счетов и 65 000 последних заказов. Заказы сроком более трех месяцев архивируются и изымаются из таблицы неуплаченных счетов. Каждая строка в таблице счетов имеет размер примерно 3 Кбайта. Таблица СЧЕТА индексируется номером заказа и фамилией заказчика. Строки в таблице ЗАКАЗЫ имеют размер в 90 байтов. Каждый заказ ссылается в среднем на 6 строк в таблице ТОВАРЫ, каждая строка которой содержит 30 байтов. Таблица ЗАКАЗЫ индексируется номером заказа. Наконец, каждая строка таблицы ТОВАРЫ ссылается на таблицу ОПИСАНИЕ_ТОВАРА, в которой в текущий момент имеется 1 050 записей размером 200 байтов. Вся эта информация сведена воедино в табл. 6.8.

Таблица 6.8. *Сводные данные об исходных параметрах транзакции*

Таблица	Количество строк	Размер строки	Чистый размер данных
СЧЕТА	260 000 (7 000 активных)	3 Кбайта	780 Мбайт = 260 Кбайт × 3 Кбайт
ЗАКАЗЫ	65 000 (8 500 активных)	90 байтов	5,9 Мбайт = 65 Кбайтов × 0,09 Кбайтов
ТОВАРЫ	390 000	30 байтов	11,7 Мбайт
ОПИСАНИЕ_ТОВАРА	1 050	220 байтов	231 Кбайт

Итак, для хранения активных данных требуется примерно 800 Мбайт = 780 Мбайт + 5,9 Мбайт дискового пространства. После этот объем, возможно, возрастет до 2 Гбайт дискового пространства.

В нормальных условиях работы оператор отвечает на телефонный звонок от покупателя, который сообщает либо номер заказа, либо имя заказчика. Примерно в 5 % случаев заказчик не может быть найден прямо по имени, и оператор должен запрашивать таблицу заказов по zip-коду. Когда заказ идентифицирован, информация о нем отображается на экране. Большая часть запросов заказчиков (более 99 %) связана с запросом состояния одного заказа.

В терминах обращения к диску основная транзакция включает чтение по ключу для выборки индекса счета, предшествующее чтению по ключу из таблицы счетов, которое читает 3 Кбайта данных. В записи счета хранится индексный ключ для поиска индекса таблицы заказов. Поскольку имеется 8 500 неоплаченных заказов и только 7 000 активных счетов, каждое обращение к индексу заказа приводит примерно к 8 500 / 7 000 = 1,21 обращениям к диску, куда входят как обращения к таблице `ЗАКАЗЫ`, так и к ее индексу. Наконец, в каждом заказе имеется ссылка на 6 строк таблицы `ТОВАРЫ`. Поскольку эти строки очень малы по размеру и записаны в базу данных в одно и то же время, уместно предположить, что их поиск будет генерировать обращение к индексу и затем только одно обращение к данным. Последнее утверждение следует из того обстоятельства, что 6 строк по 30 байтов на одну строку — это намного меньше, чем размер страницы данных. Последнее обращение происходит к таблице `ОПИСАНИЕ_ТОВАРА`. Поскольку объем этой таблицы очень мал (вся таблица составляет всего 230 Кбайтов) и она очень часто используется, можно предположить, что вся таблица кэширована и обращений к диску при чтении данных из таблицы `ОПИСАНИЕ_ТОВАРА` не происходит.

Проанализировав приведенные данные, можно прийти к заключению, что обработка каждого звонка заказчика будет вызывать следующую последовательность обращений к диску:

- одно обращение к диску по индексу таблицы `СЧЕТА`;
- 1,5 обращения к самой таблице `СЧЕТА` (строка составляет 3 Кбайта, но большинство блоков данных имеют размер 2 Кбайта);
- примерно 1,2 обращений к таблице `ЗАКАЗЫ`;
- 1,2 обращений к индексу таблицы `ЗАКАЗЫ`, каждое из которых в свою очередь генерирует обращение к таблице `ТОВАРЫ` и ее индексу.

То есть для обработки одного звонка всего потребуется примерно 6 операций чтения диска. Причем почти все эти операции выполняются в режиме произвольного доступа.

Итак, ожидается, что каждый запрос будет приводить к 6 операциям чтения диска и что 120 операторов будут генерировать 120 запросов каждые 2 мину-

ты. Следовательно, среднее число запросов будет составлять один запрос в секунду. Поскольку для обработки каждого запроса требуется 6 обращений к диску, ясно, что даже одной дисковой системы будет вполне достаточно для обработки запросов в установившемся режиме работы. Так как дисковый накопитель способен выполнить примерно 60 полностью произвольных операций ввода/вывода в секунду, отдельная дисковая система, очевидно, может обработать десять запросов, возникших почти в один и тот же момент времени, так что если остальная часть системы имеет достаточно ресурсов, дисковая подсистема может обработать почти все, что запросят пользователи.

Кроме того, для приведенного расчета следует учесть поправку, которая касается 5 % выполнения анализируемой транзакции, когда записи счетов должны отыскиваться с помощью zip-кода. Поскольку таблица счетов не индексируется по этому полю, поиск должен осуществляться последовательно. Это означает, что для поиска нужной записи счета система должна в среднем прочитать в течение двух секунд 260 000 / 2 = 130 000 строк. Поскольку размер строки составляет 3 Кбайта, для каждой строки требуется 1,5 операции ввода/вывода, что составляет в целом около 195 000 операций ввода/вывода строк по 2 Кбайта. Предположим, что диск емкостью 535 Мбайт способен выполнить около 450 последовательных операций ввода/вывода в секунду, а четыре таких диска общей емкостью 2,1 Гбайт способны выполнить примерно 1 800 операций в секунду. Даже при использовании 16 дисков, что в 4 раза превышает требуемую емкость памяти, можно обеспечить обработку только 7 200 операций в секунду. Отсюда ясно, что экономически невыгодно конфигурировать дисковую подсистему, которая будет обеспечивать необходимую пропускную способность. Единственный альтернативный вариант заключается в том, чтобы разрешить этой транзакции выполняться гораздо дольше, чем обычно, или проиндексировать таблицу счетов по полю zip-кода.

Поскольку тип СУБД не был определен, а также не было никакой информации о количестве времени, связанного с обработкой самого приложения, невозможно точно оценить количество процессоров в системе. Каждый пользователь выполняет одну транзакцию каждые две минуты, а каждая транзакция требует тривиального количества обработки: 6 операций с диском отнимают по 1,7 мс процессорного времени каждая или примерно 1 % от каждой минуты работы процессора. Для обработки запросов всех операторов потребуется примерно 10 мс процессорного времени на каждый запрос (120 пользователей / 120 секунд = 10 мс процессорного времени в секунду). Ясно, что любой процессор легко справится с такой нагрузкой.

Что же касается объема оперативной памяти, то ясно, что необходим минимальный объем в 128 Мбайт, поскольку с ней будут работать 120 пользователей. Если выделить под каждого пользователя 1 Мбайт памяти, 16 Мбайт под базовую операционную систему и 8 Мбайт для кэша, согласно общепринятой рекомендации, по которой размер кэша составляет 1 % от раз-

мера базы данных, то минимальный объем основной памяти составит 144 Мбайт, а 160 Мбайт, возможно, представляет собой более безопасную оценку.

Предупреждение

Анализ подобного рода связан с целым рядом неточностей. В частности, он пренебрегает ростом эффективности, получаемым за счет кэширования различных частей данных, не принимает во внимание накладные расходы, связанные с входом в систему и архивацией, и предполагает, что B-деревья и другие внутренние механизмы доступа к базе данных ничего не стоят. Также в расчет не принимается объем обработки, выполняющейся приложением, который часто бывает существенным. И самое главное, анализируется ситуация, в которой априорно отсутствует возможность конкуренции за ресурсы между транзакциями, т. к. все составные части рассматриваемой транзакции только читают данные, следовательно, в процессе ее выполнения можно ограничиться разделяемыми блокировками.

Учесть все эти нюансы теоретически обычно не представляется возможным.

Поэтому приведенный расчет лишь иллюстрирует общий подход к оценке минимальной конфигурации системы, с точки зрения производительности транзакций, и предоставляет возможность спрогнозировать лишь начальную конфигурацию системы, позволяя скорее определить, что данная конфигурация недостаточна для удовлетворения сформулированных требований. Однако этот метод не может решить с какой-либо определенностью, что предлагаемая конфигурация способна удовлетворить всем требованиям.

6.3. Средства управления транзакциями на SQL-сервере

Проектирование транзакций, управление ими, а также оптимизация их выполнения на любом сервере баз данных относятся к наиболее сложным аспектам программирования приложения базы данных в целом. Во-первых, крайне сложны подходы к тестированию этого аспекта функционирования приложения. Во-вторых, при работе транзакций неизбежно взаимодействуют как клиентская, так и серверная части приложения, что также вносит дополнительные трудности. Например, следует отчетливо понимать, что транзакция может стартовать как в результате выполнения операторов языка T-SQL, так и операторов, которые являются средствами управления транзакциями в рамках той среды программирования, которая выбрана непосредственно для создания клиентского приложения. Обычно в последнем случае средства управления транзакциями реализованы в контексте того способа соединения с сервером, который характерен для конкретной среды разработки. Например, при выборе ADO-соединения (ActiveX Data Objects, активные объекты

данных) независимо от среды разработки операторы BeginTrans, CommitTrans и RollbackTrans являются методами объекта Connection этого способа соединения. При использовании ODBC-соединения (Open DataBase Connectivity, открытый интерфейс доступа к базам данных) можно управлять транзакциями с помощью его средств, например, таких как оператор SQLEndTran.

Главное правило, которого следует придерживаться на этом уровне, состоит в том, что каждая транзакция от начала до конца должна управляться одним и тем же механизмом. То есть недопустимо, чтобы транзакция стартовала с помощью оператора BeginTrans в контексте ADO-соединения, а потом каким-то образом был бы выполнен оператор COMMIT для этой транзакции на языке T-SQL. Подобный подход приведет к очевидной путанице и непредсказуемым последствиям.

Все дальнейшее изложение ведется для способов управления транзакциями языка T-SQL. Это означает, что команды управления транзакциями являются операторами, расположенными либо внутри хранимой процедуры, либо внутри пакета команд на языке T-SQL. Эти хранимая процедура или пакет запускаются на выполнение любым доступным способом. При этом каждая транзакция должна быть начата и полностью завершена в рамках одной и той же хранимой процедуры или пакета.

6.3.1. Явный и неявный режимы работы транзакций

С точки зрения выполнения транзакций в контексте конкретного соединения, SQL-сервер баз данных обычно может работать в одном из двух режимов их старта: явном и неявном. Переключение между этими режимами устанавливается с помощью оператора:

```
SET IMSLICIT_TRANSACTIONS {ON | OFF}
```

При этом режим OFF считается заданным по умолчанию. Просмотр режима работы транзакций можно выполнить любым способом просмотра опций соединения сервера, например с помощью команды DBCC USEROPTIONS. Говорят, что опция OFF задает неявный режим работы транзакций, а опция ON включает явный режим их работы.

В любом из режимов работы с транзакциями допустимо использование явных операторов управления транзакциями. Эти операторы служат, в первую очередь, для предоставления возможности объединения в одну транзакцию нескольких операторов T-SQL. К подобным операторам относятся следующие:

- BEGIN TRANSACTION — начинает транзакцию;
- COMMIT TRANSACTION — фиксирует в базе данных изменения всех объектов, затронутых транзакцией;

- `ROLLBACK TRANSACTION` — выполняет откат транзакции, который заключается в том, что ни одно из изменений, выполненных в ходе транзакции, не будет зафиксировано в базе данных.

Команда `COMMIT` завершает текущую транзакцию и автоматически начинает новую. При этом гарантируется, что результаты работы завершенной транзакции фиксируются, т. е. сохраняются в базе данных.

Команда `ROLLBACK` приводит к тому, что все изменения, сделанные текущей транзакцией, откатываются, т. е. отменяются, как будто их вообще не было. При этом автоматически начинается новая транзакция.

Замечание

Перечисленные ранее операторы кажутся аналогичными операторным скобкам, при этом `BEGIN TRAN` представляется аналогичным открывающей скобке, а `COMMIT TRAN` и `ROLLBACK TRAN` — закрывающей. Однако аналогия здесь весьма условная, и следование ей часто ведет к ошибкам и недоразумениям. Так, например, для одного `BEGIN TRAN` допустимы несколько `COMMIT TRAN` или `ROLLBACK TRAN`, которые обычно расположены внутри логических операторов. Кроме того, синтаксически никак не проверяется наличие хотя бы одного оператора завершения транзакции в том случае, когда явно задано ее начало. Еще сильнее отличие указанных операторов от операторных скобок проявляется при записи последовательности вложенных транзакций.

Транзакции могут быть вложенными друг в друга. Существует глобальная переменная `@@TRANCOUNT`, значение которой возрастает на единицу всякий раз, когда выполняется оператор `BEGIN TRANSACTION`, и уменьшается на единицу при выполнении команд `COMMIT` или `ROLLBACK`.

Наиболее простым и привычным является неявный режим работы транзакций. В этом режиме, если не использовать специальные команды управления транзакциями, каждый оператор считается отдельной транзакцией, и изменения, которые он оказал на базу данных, автоматически фиксируются в ней, если оператор завершился без ошибки.

Работа в явном режиме обозначает, что нужно посылать серверу специальные операторы, задающие начало и, особенно, окончание транзакции. Транзакция обычно начинается автоматически с момента присоединения пользователя к СУБД и продолжается до тех пор, пока не произойдет одно из следующих событий:

- подана команда `COMMIT` или `ROLLBACK`;
- произошло отсоединение пользователя от СУБД и автоматическая фиксация всех его транзакций;
- произошел сбой системы, при котором запускаются сложные процессы, связанные с выполнением процедуры восстановления базы данных.

Таблица 6.9. *Операторы, начинающие транзакцию автоматически*

Основные операторы DDL	Основные операторы DML	Операторы задания и отзыва прав доступа	Операторы работы с курсорами
ALTER TABLE	UPDATE	REVOKE	FETCH
CREATE	SELECT		
TRUNCATE TABLE	DELETE	GRANT	OPEN
DROP	INSERT		

В режиме SET IMSLICIT_TRANSACTIONS ON транзакцию автоматически начинают операторы, приведенные в табл. 6.9.

В явном режиме работы транзакций после каждого из операторов, приведенных в табл. 6.9, для фиксации транзакции нужно выполнять оператор COMMIT. Если закрыть соединение без его выполнения, то изменения не сохранятся и блокировки не снимутся.

Таблица 6.10. *Сравнение режимов явной и неявной работы транзакций*

Общая часть — создание таблицы	
IF exists (SELECT * FROM dbo.sysobjects WHERE id = object_id(N'[dbo].[t1]') and OBJECTPROPERTY(id, N'IsUserTable') = 1) DROP TABLE [dbo].[t1] CREATE TABLE t1 (a INT) INSERT INTO t1 VALUES (1) GO	
Явный режим работы транзакций	**Неявный режим работы транзакций**
SET IMSLICIT_TRANSACTIONS ON GO BEGIN TRAN -- оператор не -- обязателен, но возможен INSERT INTO t1 VALUES (4) SELECT 'Уровень вложенности'= @@TRANCOUNT -- выведется значение 1 COMMIT TRAN -- нужно обязательно SELECT 'Уровень вложенности'= @@TRANCOUNT -- выведется значение 0	SET IMSLICIT_TRANSACTIONS OFF GO -- написать BEGIN TRAN нельзя INSERT INTO t1 VALUES (4) SELECT 'Уровень вложенности'= @@TRANCOUNT -- выведется значение 0 написать COMMIT TRAN нельзя SELECT 'Уровень вложенности'= @@TRANCOUNT -- выведется значение 0

Следует отметить, что операторы из табл. 6.9 автоматически начинают транзакцию только на самом верхнем уровне вложенности. То есть в случае, когда внешняя транзакция уже инициирована, внутренняя транзакция автоматически не стартует при выполнении операторов из табл. 6.9, и значение переменной `@@TRANCOUNT` не возрастает. Сравнение режимов явной и неявной работы транзакций описано в табл. 6.10.

Возникает вопрос, зачем же вообще нужен режим `IMSLICIT_TRANSACTIONS ON`? Ответ на этот вопрос состоит в том, что указанный режим позволяет удерживать блокировку до тех пор, пока она не снимется явным образом при завершении транзакции.

6.3.2. Управление уровнем изоляции транзакций

Еще одним средством, позволяющим управлять работой транзакций, является возможность задать опцию, устанавливающую *уровень изоляции транзакций*. Для этого служит оператор:

```
SET TRANSACTION ISOLATION LEVEL
 {READ UNCOMMITTED
 | READ COMMITTED
 | RESEATABLE READ
 | SERIALIZABLE}
```

Как видно из этого оператора, существуют четыре уровня изоляции транзакций, а именно:

- `READ UNCOMMITTED` — это низший уровень изоляции, равный нулю, при котором гарантируется только физическая целостность данных. Возможны как грязное чтение, так и неповторяемое чтение, а также фантомы;
- `READ COMMITTED` — на этом уровне гарантируется отсутствие грязного чтения, но проблема неповторяемого чтения и фантомов остается. Используются коллективные блокировки. Этот уровень изоляции транзакций установлен на SQL-сервере по умолчанию;
- `REPEATABLE READ` — на этом уровне блокировки накладываются на все данные запроса, предотвращая изменение данных другими пользователями. Решается проблема неповторяемого чтения;
- `SERIALIZABLE` — на этом уровне гарантируется сериализуемость, т. е. максимальный уровень изоляции транзакций. Проблема фантомов решается на уровне блокировки ключа.

Увидеть, какой уровень изоляции транзакций установлен для конкретного соединения, можно выполнив команду `DBCC USEROPTIONS`. Следует помнить,

что эта команда выводит информацию только о тех опциях соединения, значения которых отличны от значений по умолчанию.

Общее правило, относящееся к установлению уровня изоляции транзакций, таково: чем выше уровень изоляции транзакций, тем ниже уровень параллелизма работы пользователей и, следовательно, тем ниже общая производительность системы. По умолчанию установлен уровень изоляции, гарантирующий отсутствие грязного чтения. Его можно снизить только в ситуации, если известно, что соединение будет работать с данными, которые не могут быть изменены из других соединений.

Кроме изменения уровня изоляции транзакций для всего соединения в целом, можно задать конкретный уровень для отдельного оператора, который составляет транзакцию. Этот параметр задается как подсказка оптимизатору запросов уровня таблицы (так называемые TABLE HINTs) оператора `SELECT` или оператора создания представления `CREATE VIEW`. Полный перечень подсказок блокировки можно найти в BOL по критерию поиска "Locking HINTs" (хинты блокировки). Возможность использования подсказок оптимизатору считается достаточно экзотической, т. к. порождает дополнительные сложности без гарантии явных преимуществ *(см. разд. 5.6)*.

Тем не менее можно привести следующий сравнительно бесспорный совет по использованию хинтов блокировки. Если предполагается стандартная стратегия поддержания безопасности, одним из положений которой является предоставление прав доступа к данным через представления, и при этом можно очертить тот круг пользователей, которым разрешен только просмотр данных, то тогда представления, создаваемые для этой цели, можно создавать с хинтом `NOLOCK`. В последнем случае указанная подсказка добавляется как подсказка уровня таблицы в операторе `SELECT`, использованном при создании представления. Тогда при просмотре данных их уровень изоляции будет снижен до `READ UNCOMMITTED`. Но в данном случае это не вызовет никаких коллизий, т. к. вышеупомянутым пользователям в соответствии с их правами доступа отказано в возможности выполнения операторов, требующих более высокого уровня изоляции данных.

Рассмотрим несколько простейших примеров, иллюстрирующих разницу между уровнями изоляции транзакций на практике. Работа будет осуществляться на одном компьютере, следовательно, нужно организовать два клиентских соединения, используя для простоты утилиту SQL Query Analyzer. Итак, работа происходит из двух окон: в одном окне (окне A) выполняется оператор `SELECT`, устанавливающий блокировку чтения (S-блокировку), в то время как в другом окне (окне B) выполняются операторы, устанавливающие блокировку изменения. Во втором окне уровень изоляции транзакций будет принудительно изменяться начиная с самого низкого до самого высокого. Мы

увидим, как каждый последующий уровень изоляции транзакций устраняет проблему, возникающую на предыдущем уровне.

Для всех последующих примеров использована таблица `T1`, созданная оператором из листинга 6.1.

Листинг 6.1. Создание тестовой таблицы

```
IF exists (SELECT * FROM dbo.sysobjects
WHERE id = object_id(N'[dbo].[ T1]')
and OBJECTPROPERTY(id, N'IsUserTable') = 1)
DROP TABLE [dbo].[T1]
CREATE TABLE T1 (p1 INT, p2 INT)
GO
```

Для всех примеров исходной ситуацией является наличие в таблице `T1` единственной строки, содержащей данные (5, 555).

Пример 1. Чтение данных с эффектом "грязного чтения"

Транзакция в окне B изменяет данные в строке. После этого транзакция в окне A читает измененные данные и работает с ними. Затем транзакция из окна B откатывается и восстанавливает старые данные. В окне B установлена опция `SET IMSLICIT_TRANSACTIONS ON` для того, чтобы можно было организовать задержку фиксации транзакции из этого окна, перейти в окно A и работать из него (табл. 6.11).

Таблица 6.11. *Практическая организация эффекта "грязного чтения"*

Вставка данных (окно B)	Время	Чтение данных (окно A)	Результат чтения
SET IMSLICIT_TRANSACTIONS ON	t1	SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED	
INSERT INTO T1 VALUES (5,555)	t2		
------- разрыв выполнения			
	t3	SELECT * FROM T1	5 555
ROLLBACK TRAN	t4		
SELECT * FROM T1	t5		нет строк
Результат: в процессе выполнения транзакции из окна A были прочитаны данные, которые не фиксировались ни на мгновение в базе данных			

Пример 2. Эффект "неповторяемого чтения"

Транзакция в окне А дважды читает одну и ту же строку, например, по заданному логическому условию или, как в нашем примере, в таблице `T1` имеется всего одна строка. Между этими чтениями вклинивается транзакция из окна В, которая изменяет значения в прочитанной строке (табл. 6.12).

Таблица 6.12. *Практическая организация эффекта "неповторяемого чтения"*

Вставка данных (окно В)	Время	Чтение данных (окно А)	Результат чтения

`SET IMSLICIT_TRANSACTIONS OFF`
`SET TRANSACTION ISOLATION LEVEL READ COMMITTED`
`-- Для обоих окон явно установлен режим работы транзакций,`
`-- принятый по умолчанию`			

	t1	BEGIN TRAN	
	t2	SELECT * FROM T1	S1 S2
	t3	------ разрыв выполнения	5 555
	t4		
UPDATE T1 SET S1 = 6	t5	SELECT * FROM T1	S1 S2
	t6	COMMIT TRAN	6 555
Результат: в процессе выполнения транзакции из окна А были прочитаны данные, которые, с точки зрения этой транзакции, самопроизвольно изменились			

Приведем комментарий к табл. 6.12. Последняя ситуация полностью соответствует режиму работы транзакций, принятому по умолчанию, который обычно никто не меняет. В учебном примере транзакция из окна А была задержана искусственно. На практике неповторяемое чтение возникает, если транзакция из окна А выполняется настолько долго, что между двумя ее операторами `SELECT` успевает выполниться и зафиксироваться оператор `UPDATE` из транзакции, выполняющейся в окне В.

В результате, чтобы понизить вероятность неповторяемого чтения, транзакция из окна А должна быть как можно короче. Кроме того, очевидно, что следует избегать написания программного кода таким образом, чтобы операторы `SELECT` принудительно объединялись в одну транзакцию.

Интересно, что если в последнем примере убрать в окне А операторы `BEGIN TRAN` и `COMMIT TRAN`, то результаты выполнения пары операторов `SELECT` все равно будут разными, но это уже две разные транзакции, и речи о неповторяемом считывании быть не может.

Замечание

Если при режиме работы транзакций из второго примера попытаться выполнять программу из примера 1, то оператор `SELECT * FROM T1` из окна А будет невозможно выполнить вообще до тех пор, пока не зафиксируется транзакция из окна В.

Пример 3. Чтение данных с эффектом "фантомов"

Транзакция из окна А дважды выполняет выборку строк по одному и тому же условию. Между выборками вклинивается транзакция из окна В, которая добавляет новую строку, удовлетворяющую условию отбора (табл. 6.13).

Таблица 6.13. *Практическая организация эффекта "фантомов"*

Вставка данных (окно В)	Время	Чтение данных (окно А)	Результат чтения

`SET IMSLICIT_TRANSACTIONS OFF`
`SET TRANSACTION ISOLATION LEVEL RESEATABLE READ`
`-- Для обоих окон явно установлен режим работы транзакций,`
`-- принятый по умолчанию`			

	t1	BEGIN TRAN	
	t2	SELECT * FROM T1	S1 S2
		--- разрыв выполнения	5 555
INSERT INTO T1 VALUES (7,777)	t3		
	t4	SELECT * FROM T1	S1 S2
	t5	COMMIT TRAN	5 555
			7 777
Результат аналогичен предыдущему: в процессе выполнения транзакции из окна А были прочитаны данные, которые, с точки зрения этой транзакции, самопроизвольно изменились — добавилась новая строка			

Разница между третьим и вторым примерами состоит в повышении уровня изоляции транзакций и в замене оператора `UPDATE` оператором `INSERT`. Действия, выполненные во втором примере, при установленном в третьем примере уровне изоляции транзакций выполнить не удастся — оператор `UPDATE` транзакции из окна В теперь не выполнится, пока не завершится транзакция из окна А.

Мы видим, что в транзакции из окна А из третьего примера появился фантом — строка с данными (7, 777). Устранить этот фантом можно только на самом высоком уровне изоляции транзакций, который задается опцией `SERIALIZABLE`. На этом уровне изоляции невозможно будет выполнить опера-

тор INSERT транзакции из окна B до тех пор, пока транзакция из окна A не будет зафиксирована.

В заключение приведем сводную таблицу, которая описывает, какие эффекты имеют место на определенном уровне изоляции транзакций.

Таблица 6.14. *Уровни изоляции и эффекты, связанные с конфликтами транзакций*

Уровень изоляции	Грязное чтение	Неповторяемое чтение	Фантомы
READ UNCOMMITTED	Да	Да	Да
READ COMMITTED	Нет	Да	Да
REPEATABLE READ	Нет	Нет	Да
SERIALIZABLE	Нет	Нет	Нет

6.3.3. Управление блокировками на SQL-сервере

Как уже говорилось, блокировки используются для достижения изолированности транзакций на серверах баз данных. Когда транзакция считывает или записывает строку в таблице, сервер блокирует строку для предотвращения изменения данных или получения неверных данных. Этот процесс выполняется автоматически, и в распоряжении разработчика имеется небольшое количество средств для управления этим процессом. К этим средствам, кроме задания уровня изоляции транзакций, относятся следующие возможности: управление временем ожидания разблокирования необходимых ресурсов и управление приоритетом транзакций, с точки зрения возможности их выбора в качестве жертвы тупика.

Параметры настройки соединения, связанные с работой транзакций

Как известно, для того чтобы выполниться, транзакция должна иметь возможность накладывать необходимые ей блокировки на определенные объекты. Если эти объекты заблокированы другой транзакцией таким способом, что наложить новые блокировки невозможно, транзакция будет ожидать разблокирования нужных ресурсов до тех пор, пока не будет превышен установленный в системе предел ожидания. Значение этого предела задается в миллисекундах как опция настройки соединения и сохраняется в системной переменной @@LOCK_TIMEOUT. По умолчанию это значение равно 1, что означает бесконечный период ожидания. Изменить его можно с помощью команды SET LOCK_TIMEOUT.

В том случае, когда транзакция превысила период ожидания, генерируется сообщение об ошибке с номером 1222 и текстом

```
Lock request time-out period exceeded
```

Нужно понимать, что превышение времени ожидания не ведет к автоматическому откату транзакции. Поэтому если имеется вероятность генерации такого сообщения, то клиентское приложение должно обязательно отслеживать эту ситуацию, и в нем должны быть предусмотрены действия по реакции на это сообщение. Такими действиями может быть, например, откат всей транзакции, для этого должен быть явный выход на оператор `ROLLBACK`, после чего транзакцию можно повторить сначала.

Если период ожидания разблокирования установлен, как и принято по умолчанию, бесконечно большим, то любая транзакция в конце концов выполнится, если не будет иметь место ситуация тупика. В случае тупика одна из транзакций будет, как известно, снята, и по умолчанию сервер сам выбирает транзакцию-жертву. При этом тому соединению, в контексте которого предпринималась попытка выполнить транзакцию-жертву, будет послано сообщение об ошибке с кодом 1205 и текстом

```
Transaction (Process ID %d) was deadlocked on {%Z}
resources with another process and has been chosen
as the deadlock victim. Rerun the transaction.
```

Перевод этого сообщения звучит примерно так: "Транзакция процесса с идентификатором `ID` вступила в конфликт тупика с другим процессом и была выбрана в качестве жертвы. Повторите транзакцию".

Если необходимо, чтобы в каком-то из соединений транзакции не становились жертвами тупика, то можно изменить значение соответствующего приоритета. В контексте конкретного соединения с помощью команды `SET DEADLOCK_PRIORITY {LOW |NORMAL}` можно понизить приоритет транзакций со значения `NORMAL`, принятого по умолчанию, до значения `LOW`. В последнем случае вероятность того, что транзакция будет выбираться в качестве жертвы, возрастает, особенно если в конфликт тупика вступили транзакции с приоритетом `LOW` и `NORMAL`. Таким образом, если известно, что какая-то из транзакций особенно важна, то нужно понизить `DEADLOCK_PRIORITY` для всех конкурирующих с ней транзакций.

Манипуляция со значениями параметров `LOCK_TIMEOUT` и `DEADLOCK_PRIORITY` сама по себе редко может радикально улучшить ситуацию, когда конкуренция за ресурсы ведет к заметному снижению производительности системы.

Так, уменьшение `LOCK_TIMEOUT` никак не позволит ускорить выполнение конкретной транзакции, а только даст возможность получать соответствующие

сообщения и, возможно, реагировать на них, что для пользователя, может, и веселее, чем просто ожидать своей очереди.

Изменение `DEADLOCK_PRIORITY` также бесполезно в достаточно типичном случае, когда из многих клиентских соединений выполняется по смыслу бизнес-логики одна и та же транзакция. В этом случае конкуренция за ресурсы и возможность возникновения тупика максимальны, т. к. все обращаются к одним и тем же данным, и повлиять на эту ситуацию изменением приоритета в принципе невозможно.

Поэтому стратегически возможность увеличения общей производительности приложения в ситуации острой конкуренции за ресурсы при выполнении транзакций может быть достигнута двумя способами, которые должны эффективно сочетаться:

- *Грамотная административная политика*, суть которой должна состоять в том, чтобы по возможности распределить нагрузку на сервер наиболее равномерно.
- *Правильное проектирование самих транзакций*, особенно тех из них, которые выполняются в моменты наибольших нагрузок на сервер.

Для выполнения последней рекомендации, кроме использования некоторых достаточно абстрактных общих правил, нужно уметь выявлять и устанавливать причины ситуаций, когда конкуренция за ресурсы происходит наиболее остро и ведет к проблемам производительности.

Средства мониторинга блокировок на SQL-сервере

Для решения сформулированной выше задачи на SQL-сервере используются специальные средства мониторинга процесса выполнения транзакций и, следовательно, накладываемых ими блокировок. К этим средствам относятся следующие:

- системные хранимые процедуры, такие как `sp_who`, `sp_lock`;
- системная таблица `syslockinfo`;
- системная переменная `@@TRANCOUNT`;
- средство `DBCC OPENTRAN`;
- окна утилиты Enterprise Manager под названием ...**Management\Current Activity\Locks/Process ID** и ...**Management\Current Activity\Locks/Object**;
- утилита SQL Profiler, а именно такие ее классы событий, как **Locks** и **Transaction**.

Нет возможности подробно останавливаться на всех этих средствах мониторинга транзакций и блокировок. Рассмотрим на примере использования сис-

темной хранимой процедуры sp_lock, какая же информация при этом представляется полезной и как ее трактовать.

Запущенная без параметров процедура sp_lock выдает информацию, приведенную в табл. 6.15.

Таблица 6.15. *Выходные данные процедуры sp_lock*

Имя столбца	Тип данных столбца	Описание
spid	smallINT	Системный идентификатор процесса SQL-сервера, идентифицирующий, в частности, конкретное соединение (глобальная переменная @@SPID)
dbid	smallINT	Системный идентификатор базы данных, которая ходатайствует о наложении блокировки
ObjId	INT	Системный идентификатор объекта, который ходатайствует о наложении блокировки
IndId	smallINT	Системный идентификатор индекса
Type	nchar(4)	Что заблокировано: DB — база данных FIL — файл IDX — индекс SG — страница KEY — сервисный ключ TAB — таблица EXT — экстент RID — строка и ее идентификатор
Resource	nchar(16)	Какие конкретно ресурсы заблокированы для уровня, например строк или страниц
Mode	nvarchar(8)	Тип блокировки, например исключительная или разделяемая
Status	INT	Статус блокировки, о которой ходатайствует процесс: GRANT — предоставлена WAIT — состояние ожидания CNVRT — состояние преобразования

Рассмотрим пример работы системной процедуры sp_lock для следующей ситуации. Пусть с сервером созданы три соединения, имеющие системные идентификаторы 52, 54 и 55 соответственно. В соединении 54 стартовала и не

зафиксировалась транзакция по вставке строки в таблицу, т. е. это соединение наложило X-блокировку. После этого из соединения 55 нужно считать все строки упомянутой таблицы. Уровень изоляции транзакций установлен таким, что из соединения 55 наложить необходимую блокировку чтения невозможно, поэтому транзакция из этого соединения находится в состоянии ожидания. Из соединения 52 тем временем запускается процедура `sp_lock`. Далее приведена табл. 6.16, отображающая результаты работы процедуры `sp_lock`, из которой удалены строки, связанные с выполнением процессов и блокировками объектов, которые нас не интересуют в данном случае.

Таблица 6.16. *Результаты работы процедуры* `sp_lock`

spid	dbid	ObjId	IndId	Type	Resource	Mode	Status
54	5	306100131	0	TAB		IX	GRANT
54	5	306100131	0	RID	0.12	X	GRANT
55	5	306100131	0	RID	0.12	S	WAIT
55	5	306100131	0	TAB		IS	GRANT

Итак, соединение 55 выглядит как бы "зависшим", потому что оно находится в состоянии ожидания. Подобное состояние является сигналом к выяснению того, почему же оно возникло. Запускаем из нового соединения процедуру `sp_lock`. По результатам ее работы видно, что "зависшее" соединение ожидает (`WAIT` в столбце `Status`) возможности наложить S-блокировку на объект с идентификатором `ObjId` = 306100131. Также видно, почему это невозможно — соединение 54 уже наложило и держит X-блокировку на этот объект.

Другой характерный пример целесообразности использования средств мониторинга блокировок может быть примерно следующим. Предположим, что имеется база данных, которая поддерживает генерацию внутренних производственных отчетов компании. Обычно имеется среднее число пользователей, одновременно работающих с этой базой данных (около пяти), а среднее время выполнения запроса меньше, чем одна секунда. Однако в пятницу после обеда и в понедельник утром, когда большинство служащих вводит данные своего недельного рабочего расписания, количество одновременно работающих пользователей увеличивается примерно в десять раз, а время выполнения запроса увеличивается в еще большей пропорции. Что следует предпринять в данной ситуации?

Стандартной рекомендацией для описанной ситуации является использование системных процедур `sp_lock` или `sp_who` с целью выявления заблокированных ресурсов. Информация, полученная подобным способом, как раз и

является исходной посылкой для принятия решений о перепроектировании транзакций с целью уменьшения конфликтов доступа, что обычно ведет к общему росту производительности системы.

6.3.4. Общие рекомендации по программированию транзакций

Итак, сформулируем основную стратегию действий, которая может быть применена для повышения быстродействия приложения, с точки зрения работы транзакций: нужно уметь определять конкретную причину проблемы, используя адекватные средства диагностики и мониторинга, и устранять эту проблему путем перепроектирования проблемных транзакций. Самым сложным в данном случае является ответ на вопрос, в чем же может такое перепроектирование заключаться.

Например, для рассмотренного примера про составление рабочего расписания можно предположить, что приложение спроектировано таким образом, что в момент ввода данных все пользователи запускают одну и ту же транзакцию и в рамках этой транзакции вводят свои данные. Ясно, что конкуренция за ресурсы в этой ситуации существенно возрастает, возрастает также и вероятность возникновения тупика. В данном случае стоит подумать о том, нельзя ли вынести процесс заполнения формы рабочего расписания за рамки операторов `BEGIN TRAN` и `COMMIT TRAN`.

Повысить общее быстродействие выполнения смеси транзакций в ситуации, когда не предполагается изменять уровень их изоляции без использования явных средств управления блокировками, можно за счет выполнения следующих принципов:

- транзакции должны быть максимально короткими;
- транзакции лучше всего размещать внутри хранимых процедур;
- следует по возможности избегать вложенных транзакций, т. к. в целом они удлиняют время выполнения отдельной транзакции и сильно запутывают программный код.

Сформулированные ранее принципы считаются универсальными и общеизвестными. Следующие рекомендации приводятся менее часто. К ним относятся такие советы:

- правильный порядок выполнения отдельных операторов внутри транзакций может влиять на общую производительность;
- использовать возможности уменьшения уровня изолированности транзакций, если это не влияет отрицательно на функциональность приложения;

- нужно стараться добиваться оптимального сочетания действий, выполняемых на сервере и на клиенте.

Приведенные рекомендации некоторым образом поясняются и по возможности обосновываются далее.

Основные приемы, используемые при программировании транзакций

Итак, основной рекомендацией является требование проектировать транзакции таким образом, чтобы они были максимально короткими. В первую очередь, речь здесь идет о транзакциях, которые накладывают X-блокировки, т. е. о транзакциях изменения данных. При проектировании таких транзакций нужно попытаться разбить весь процесс перевода базы данных из одного целостного состояния в другое на последовательность максимально коротких изменений, каждое из которых может быть зафиксировано (выполнен оператор `COMMIT`) по отдельности.

Когда речь идет о проектировании сложных транзакций, включающих несколько операторов модификации данных и сложную логику проверок, нужно отчетливо понимать основной механизм, на основании которого происходят эти проверки. Чаще всего используется проверка глобальной переменной `@@ERROR`, которая содержит код ошибки, после каждого выполненного оператора. Значение этой переменной равно нулю, если ошибки при выполнении оператора не было. Для пояснения механизма использования этой переменной при выполнении проверок рассмотрим простейший пример, суть которого заключается в том, что нужно объединить в одну транзакцию вставку двух разных значений в поле первичного ключа. При этом проверяется выполнение ограничения, и если хоть одно из значений не удовлетворяет ограничению, то оба значения не будут вставлены. Если операторы `INSERT` не объединять в одну транзакцию, то невозможно одновременно, с точки зрения транзакции, вставить два значения. Приведенный листинг 6.2 является простейшим примером, который демонстрирует работу механизма явных транзакций.

Листинг 6.2. Пример работы явной транзакции

```
IF exists (SELECT * FROM dbo.sysobjects WHERE id =
object_id(N'[dbo].[TESTTRAN]') and
OBJECTPROPERTY(id, N'IsUserTable') = 1)
DROP TABLE [dbo].[TESTTRAN]
GO
-- Создается тестовая таблица
CREATE TABLE TESTTRAN (Cola INT PRIMARY KEY, Colb CHAR(3))
GO
```

```
-- Заполнение таблицы первоначальными данными
INSERT INTO TESTTRAN(Cola) VALUES(1)
INSERT INTO TESTTRAN(Cola) VALUES(2)
INSERT INTO TESTTRAN(Cola) VALUES(3)
SELECT Cola FROM TESTTRAN
-- Объявляем переменную для хранения кода ошибки
DECLARE @err INT
-- Начинаем транзакцию
BEGIN TRAN
-- Пытаемся вставить дубликатный ключ, искусственно вызывая ошибку
INSERT INTO TESTTRAN(Cola) VALUES(3)
-- Заносим код ошибки в переменную @err для ее последующей проверки
SET @err = @@ERROR
SELECT @err
INSERT INTO TESTTRAN (Cola) VALUES(7) -- Новый ключ
-- Проверяем, было ли нарушение ограничения первичного ключа
IF @err<>0
ROLLBACK TRAN -- ошибка была, транзакция откатывается
ELSE COMMIT -- ошибки не было, транзакция фиксируется
-- Проверяем содержимое таблицы — в ней только три записи
SELECT Cola FROM TESTTRAN
```

При работе программы из листинга 6.2 будет сгенерировано сообщение о том, что произошло нарушение ограничения первичного ключа (код ошибки 2627) для одного из операторов `INSERT`. Отметим, что ошибка, ставшая причиной нарушения ограничения первичного ключа, вызвана искусственно, поэтому оператор `SET @err = @@ERROR` выполнен только один раз, т. к. место ошибки заранее известно.

В общем случае предлагается проверять значение переменной `@@ERROR` после каждого оператора, и если оно отлично от нуля, откатывать транзакцию — в этом и состоит суть рекомендуемого механизма проверки отсутствия ошибок на сервере. Однако заметим, что подобный способ действий не является оптимальным, с точки зрения общей производительности клиент-серверного приложения. Очевидно, что постоянное выполнение логического оператора `IF @@ERROR <> 0` ощутимо замедляет работу, и, кроме того, при выполнении данной рекомендации размер программы существенно возрастает. Зачастую более эффективным решением является перехватывание сообщений об ошибках по мере их возникновения на клиенте, тем более что многие среды разработки предоставляют и рекомендуют использование подобной возможности.

Вложенные транзакции

Как уже говорилось, SQL-сервер допускает *вложенные транзакции*. Механизм их использования является достаточно сложным, и чтобы его применять, нужно четко представлять, как он работает. Приведенный листинг 6.3 является учебным примером, позволяющим в общих чертах разобраться в этом.

Листинг 6.3. Работа вложенных транзакций

```
IF exists (SELECT * FROM dbo.sysobjects WHERE id =
object_id(N'[dbo].[TESTTRAN]') and
OBJECTPROPERTY(id, N'IsUserTable') = 1)
DROP TABLE [dbo].[TESTTRAN]
GO
-- Создается тестовая таблица
CREATE TABLE TESTTRAN (Cola INT PRIMARY KEY, Colb CHAR(3))
GO
BEGIN TRANSACTION OuterTran -- @@TRANCOUNT равно 1
GO
INSERT INTO TESTTRAN VALUES (1, 'aaa')
GO
BEGIN TRANSACTION Inner1 -- @@TRANCOUNT равно 2
GO
INSERT INTO TESTTRAN VALUES (2, 'bbb')
GO
BEGIN TRANSACTION Inner2 -- @@TRANCOUNT равно 3
GO
INSERT INTO TESTTRAN VALUES (3, 'ccc')
GO
COMMIT TRANSACTION Inner2 -- Уменьшает @@TRANCOUNT до 2
-- Ничего не фиксируется
GO
COMMIT TRANSACTION Inner1 -- Уменьшает @@TRANCOUNT до 1
-- Ничего не фиксируется
GO
COMMIT TRANSACTION OuterTran -- Уменьшает @@TRANCOUNT до 0
-- Фиксируется транзакция OuterTran.
```

Листинг 6.3 позволяет отслеживать уровень вложенности транзакций, поэтому в комментариях описано, как изменяется значение переменной `@@TRANCOUNT` там, где это происходит. Кроме того, в комментариях явно отслеживается тот момент, когда же происходит фиксация изменений в базе данных. Видно, что блокировки снимаются только после того, как выполнен оператор `COMMIT` для

самой внешней транзакции. Это утверждение очень легко проверить — если из другого соединения попытаться открыть таблицу `TESTTRAN`, например, с помощью Enterprise Manager, то пока не будет выполнен третий оператор `COMMIT` листинга 6.1, эта таблица будет заблокирована и никаких вставленных строк увидеть не удастся. При этом не имеет значения то обстоятельство, что в данной программе используются именованные транзакции. Порядок согласования `BEGIN TRAN` и `COMMIT` определяется только порядком их выполнения. Имена транзакций, таким образом, носят не более чем информационный смысл для разработчика. То есть может иметь место ситуация, когда для поименованного оператора `BEGIN TRAN` выполнится фиксация транзакции с помощью оператора `COMMIT`, в котором указано другое имя транзакции. Полезным может оказаться следующее правило управления транзакциями: перед инициированием транзакции проверять наличие уже открытой транзакции и открывать новую транзакцию, выполняя оператор `BEGIN TRANSACTION` только в случае отсутствия незавершенной транзакции.

Если вложенные транзакции тем не менее используются, то нужно иметь в виду, что откат (срабатывание оператора `ROLLBACK`) в этом случае можно организовать только к началу самой внешней именованной транзакции, а попытка любого другого отката приводит к ошибке. Это обстоятельство является еще одним аргументом в пользу того, что явные вложенные транзакции использовать нежелательно. Если все же необходимо достижение эффекта поэтапного отката транзакции, то следует использовать *промежуточные точки сохранения* (savepoints).

Промежуточные точки сохранения позволяют запоминать промежуточные состояния базы данных во время выполнения транзакции. Напомним, что для того, чтобы задать промежуточную точку, используется оператор `SAVEPOINT` с указанием имени промежуточной точки сохранения. После задания такой промежуточной точки возможен откат с помощью оператора `ROLLBACK TO` с указанием ее имени. В этом случае все промежуточные изменения будут отменены.

Используя именованные промежуточные точки в одной транзакции, можно создавать несколько промежуточных точек. Изменения от промежуточной точки могут быть отменены откатом до предыдущей промежуточной точки или откатом всей транзакции.

Следует отметить, что откат до промежуточной точки не ведет к откату всей транзакции; транзакция продолжает находиться в активном, неподтвержденном состоянии. Отсюда следует, что в случае неуспеха всей транзакции (например ошибки) все промежуточные точки и все, что с ними было связано, пропадет. Точно так же пропадут все изменения в базе данных, связанные с промежуточными точками, и после осуществления `COMMIT` для всей транзакции.

С точки зрения производительности приложения, считается, что промежуточные точки сохранения полезны в той ситуации, когда возникновение ошибок маловероятно. В этом случае может быть более эффективным однократное выполнение проверки. Модификации данных и откаты являются дорогостоящими операциями, так что промежуточные точки сохранения эффективны, только если вероятность ошибки, ведущей к откату, низкая, а стоимость предварительной проверки согласованности данных перед такой модификацией относительно высока.

Для пояснения сказанного приведем пример. Этот пример иллюстрирует использование точки сохранения в системе материального учета с низкой вероятностью исчерпания запаса некоторого товара. В соответствующей базовой таблице имеется ограничение целостности данных `CHECK (ОСТАТОК >= 0)`, связанное с контролем того обстоятельства, что не может быть продано товара больше, чем имеется в наличии.

Предположим, что разрабатывается транзакция, которая предназначена для занесения в базу данных сведений о некотором сложном заказе. На клиенте заполняется форма, содержащая сведения об этом заказе, после чего стартует транзакция. Если весь заказ целиком не может быть выполнен в связи с отсутствием какого-то из товаров, то необходимо откатив изменение таблицы, содержащей сведения об остатках товаров, сохранить, тем не менее, сведения о самом заказе.

Этот пример предполагает, что по некоторым причинам, к которым могут относиться соединение по медленному модему или глобальной сети, предварительная проверка и подтверждение достаточности запаса каждого товара являются относительно дорогостоящими действиями. В этом случае быстрая проверка значения `@@ERROR` после попытки обновления может оказаться намного более эффективной, чем предварительная проверка.

Пусть рассматриваемая транзакция имеет имя `OUTER_TRAN`. В процессе ее выполнения будет вызываться хранимая процедура с именем `ТОВАРЫ_UPDATE`, внутри которой происходит выполнение оператора изменения интересующей нас таблицы с именем `ТОВАРЫ`. Нарушение ограничения проверки этой таблицы генерирует ошибку с кодом 547. Внутри хранимой процедуры создана точка сохранения вызывающей ее транзакции. Если ошибка 547 происходит, то откатывается только оператор `UPDATE`, а реальное количество остатка соответствующего товара возвращается вызывающему процессу. Тогда в вызывающем процессе имеется возможность размещения кода, который откатит всю транзакцию, либо сохранит сведения о заказе с какими-то пометками, либо уменьшит количество заказанного недостающего товара на реально имеющееся в наличии его количество (листинги 6.4 и 6.5).

Листинг 6.4. Хранимая процедура для отката до точки сохранения

```
-- Напомним структуру таблицы ТОВАРЫ
CREATE TABLE [dbo].[ТОВАРЫ] (
       [Товар_Ид] [int] IDENTITY (1, 1) NOT NULL,
       [Товар] [varchar] (15) COLLATE Cyrillic_General_CI_AS NOT NULL,
       [Цена_учетн] [money] NOT NULL,
       [Остаток] [int] NULL
-- Ограничение добавлено только для данного примера
    CONSTRAINT Остаток_Check CHECK (Остаток > 0))
    ON [PRIMARY]
GO
CREATE PROCEDURE ТОВАРЫ_UPDATE
@Товар_Ид INT, -- параметр для идентификатора товара
@Кол_расх INT -- параметр для отпускаемого количества товара
AS
DECLARE @ERRORVar INT -- переменная для кода ошибки
    -- Создаем точку сохранения для транзакции,
    -- в рамках которой вызвана хранимая процедура
SAVE TRANSACTION OUTER_TRAN
UPDATE ТОВАРЫ SET ОСТАТОК = ОСТАТОК - @Кол_расх
       FROM ТОВАРЫ WHERE Товар_Ид = @Товар_Ид
SELECT @ERRORVar = @@ERROR
IF (@ERRORVar = 547) -- Нарушено ограничение Остаток_Check
       BEGIN
       ROLLBACK TRANSACTION OUTER_TRAN
       RETURN (SELECT ОСТАТОК FROM ТОВАРЫ WHERE Товар_Ид = @Товар_Ид)
       END
              ELSE
              RETURN 0
GO
```

Листинг 6.5. Вызывающий процесс для процедуры из листинга 6.4

```
-- Переменная для приема кода возврата вызванной хранимой процедуры
DECLARE @RETURN INT
-- Инициируем транзакцию
BEGIN TRAN OUTER_TRAN
/*********************************************************************
** Здесь можно размещать операторы, которые вставляют
** в разнообразные базовые таблицы сведения о заказе, например,
** INSERT INTO ЗАКАЗ VALUES .. ..
*********************************************************************/
```

```
-- Вызывается хранимая процедура
EXEC @RETURN = ТОВАРЫ_UPDATE 25, 60
-- Если нужно, выводим значение кода возврата вызова процедуры
SELECT @RETURN
-- Анализ ситуации
 IF @RETURN <> 0
       BEGIN
/*********************************************************************
** UPDATE таблицы ТОВАРЫ откачен. Можно откатить всю транзакцию **/
       ROLLBACK TRAN OUTER_TRAN
       END
/*********************************************************************
** Транзакция фиксируется целиком, таблица ТОВАРЫ изменена **/
              ELSE
              COMMIT TRAN
```

В листинге 6.4 код программы написан таким образом, что при недостатке товара откатывается вся транзакция `OUTER_TRAN` целиком, т. е. изменения для таблицы `ЗАКАЗ` также оказываются незафиксированными в базе данных. Если же в программе из этого листинга заменить оператор `ROLLBACK TRAN OUTER_TRAN` оператором `COMMIT TRAN OUTER_TRAN`, тогда изменения для таблицы `ЗАКАЗ` будут зафиксированы.

Последний пример в некотором смысле перекликается с примером из листинга 5.39, в котором рассмотрен триггер, срабатывающий в рамках транзакции, связанной со вставкой товарной позиции в расходную накладную. Внутри триггера проверялось условие наличия достаточного количества товара по отдельной товарной позиции. Если это условие оказывалось нарушенным, то вся транзакция откатывалась, т. е. соответствующая товарная позиция не была вставлена в расходную накладную. В последнем же примере описанная проверка осуществляется не в триггере, а с помощью ограничения базы данных уровня таблицы.

Все описанные соображения по поводу первых трех рекомендаций, связанных с проектированием транзакций, можно обобщить, предложив следующий примерный шаблон написания хранимой процедуры, содержащей код сложной транзакции от момента ее старта до завершения (листинг 6.6).

Листинг 6.6. Шаблон хранимой процедуры, содержащей сложную транзакцию

```
IF EXISTS (SELECT * FROM sysobjects WHERE type = 'S'
AND name = '<Stored_Srocedure_Name>')
 BEGIN
 PRINT ' Удаляется Stored_Procedure_Name'
```

```
 DROP PROCEDURE <Stored_Procedure_Name>
 END
GO
/*********************************************************************
** Хранимая процедура с именем  Stored_Procedure_Name
*********************************************************************/
CREATE PROCEDURE <Stored_Procedure_Name>
AS
-- Переменная для хранения уровня вложенности транзакций
DECLARE @trncount INT
-- Начинаем транзакцию, если она еще не начата
SELECT @trncount = @@trancount
IF @trncount = 0
-- Если нет незафиксированной транзакции, инициируем новую транзакцию
 BEGIN TRANSACTION <transaction_name>
 ELSE
-- Если имеется незафиксированная транзакция,
-- организуем для нее точку сохранения
 SAVE TRANSACTION <transaction_name>
IF @@ERROR <> 0 GOTO errorTrn
/*********************************************************************
** Операторы, формирующие тело процедуры
*********************************************************************/
IF @trncount = 0
 BEGIN
 COMMIT TRANSACTION <transaction_name>
 IF @@ERROR <> 0 GOTO error
 END
-- Выход из процедуры в случае успешной фиксации транзакции
RETURN(0)
-- Выход из процедуры при наличии ошибки, ведущей к откату транзакции
errorTrn:
       ROLLBACK TRANSACTION <transaction_name>
error:
RETURN(1)
GO
GRANT EXECUTE ON <Stored_Procedure_Name> TO PUBLIC
```

Использование вложенных транзакций, тем не менее, оправдано, когда для операторов изменения данных, находящихся внутри транзакции, определены триггеры *(см. главу 5)*. Триггеры, как известно, всегда запускаются из транзакции и служат, в первую очередь, для выполнения сложных проверок согласованности базы данных. При этом неважно, явная эта транзакция или нет,

включает ли она в себя только тот оператор, к которому привязан триггер, или еще и другие операторы. Основное правило согласования взаимодействия триггера и транзакции, которая была причиной его срабатывания, состоит в том, что выполнение команды ROLLBACK в триггере приводит к откату всех операций, начиная с самой внешней BEGIN TRAN, завершает вычисления в триггере и прерывает текущий пакет. Этим отличается срабатывание оператора ROLLBACK в триггере от его срабатывания в процедуре, при котором пакет команд не прерывается. Кроме того, при использовании триггеров в транзакциях необходимо помнить о следующих нюансах:

- команды, следующие в триггере за ROLLBACK, будут выполнены. Откат относится к обработке транзакций, а не к потоку команд. Поэтому если при откате нужно прервать выполнение команд, образующих тело триггера, то следует использовать команду RETURN;
- есть данные о том, что использование в триггере команды ROLLBACK без соответствующей команды RETURN ведет к необъяснимым проблемам для некоторых клиентских приложений;
- в триггере не следует использовать команду BEGIN TRAN, т. к. команды, образующие триггер, уже находятся внутри транзакции;
- команда SAVEPOINT в триггере допустима, и работает она по обычным правилам. При использовании промежуточных точек сохранения в триггере считается полезным вставлять команду генерации сообщения RAISERROR вместе с RETURN с тем, чтобы вызывающий процесс мог проанализировать код ошибки и предпринять необходимые действия — откат транзакции, откат до точки сохранения или фиксацию изменений;
- в триггерах нельзя выполнять откат именованной транзакции, это приведет к ошибке времени выполнения, откату всей работы и отмене пакета.

Другие рекомендации для сервера

Рассмотрим еще одну из интересных и неочевидных рекомендаций. Предположим, что предвидится конкуренция между двумя транзакциями, которые изменяют данные в одних и тех же таблицах, каждая по-своему. Чтобы снизить вероятность тупиков в этой ситуации, рекомендуется по возможности располагать операторы изменения данных в одних и тех же таблицах в одинаковом порядке, что иллюстрируется в табл. 6.17.

Предположим, что нужно выполнять множество вставок данных в один и тот же столбец, и при этом в 90 % случаев вставляется одно и то же значение. В подобной ситуации трафик по сети уменьшится путем создания значения по умолчанию для указанного столбца.

Таблица 6.17. *Порядок выполнения частей транзакций*

Хорошо		Плохо	
Транзакция 1	**Транзакция 2**	**Транзакция 1**	**Транзакция 2**
BEGIN TRAN UPDATE TABLE1 .... UPDATE TABLE2 .... UPDATE TABLE3 END TRAN	BEGIN TRAN UPDATE TABLE1 .... UPDATE TABLE2 .... UPDATE TABLE3 END TRAN	BEGIN TRAN UPDATE TABLE1 .... UPDATE TABLE2 .... UPDATE TABLE3 END TRAN	BEGIN TRAN UPDATE TABLE3 .... UPDATE TABLE2 .... UPDATE TABLE1 END TRAN

При выполнении оператора UPDATE используются дополнительные ресурсы SQL-сервера. Поэтому в этом случае полезно придерживаться следующих рекомендаций (хотя особых альтернатив действиям, неправильным, с точки зрения этих рекомендаций, обычно найти не удается, но все-таки):

- если нужно изменить столбец таблицы, по которому создан уникальный индекс, то лучше изменять строки по одной, а не целым диапазоном;
- лучше вообще не изменять значения в столбце первичного ключа;
- при изменении данных в столбце типа VARCHAR нужно постараться, чтобы новые значения имели ту же самую длину, что и исходные;
- нужно стараться минимизировать количество выполнений оператора UPDATE для таблиц, обладающих триггером для этого оператора;
- нужно стремиться избегать частых изменений данных в тех столбцах, которые подвергаются репликации;
- нужно по возможности избегать частых изменений данных в тех столбцах, которые используются при создании громоздких композитных индексов;
- нужно стараться избегать изменений данных в тех столбцах, которые используются в опции WHERE оператора UPDATE, т. е. являются потенциально плохими, с точки зрения быстродействия этого оператора. В этом случае один и тот же столбец используется как в опции WHERE, так и в опции SET.

При использовании оператора SELECT INTO следует иметь в виду, что этот оператор блокирует системные таблицы, поэтому лучше выполнять эти команды тогда, когда сервер наименее загружен.

Если программируется сложная транзакция, состоящая из нескольких частей, одна из которых с высокой степенью вероятности может породить откат всей транзакции, то эту часть нужно по возможности выполнять в начале всей транзакции.

Использование возможностей клиента

Основной парадигмой клиент-серверных технологий, позволяющей добиться оптимального функционирования приложения, является правильное распределение функциональности между клиентом и сервером. Особенно это относится к проектированию транзакций, т. к. по своей сути ни одна транзакция не может быть выполнена только на сервере. Ведь данные запроса на генерацию отчета пользователь должен увидеть в адекватно отформатированном виде, и пользователь же на клиенте заполняет формы, без данных из которых невозможно выполнение транзакций по модификации базы данных.

В OLTP-системах к наиболее часто выполняемым транзакциям относятся такие, в процессе которых пользователь модифицирует информацию на сервере, например, вводит новые данные. Разумеется, выполнение таких транзакций невозможно без взаимодействия с клиентом хотя бы потому, что для их выполнения нужен пользовательский интерфейс, содержащий формы ввода данных. Объективно, пользователь взаимодействует с интерфейсом очень медленно. Поэтому по возможности нужно избегать проектирования приложения таким образом, чтобы взаимодействие с пользователем выполнялось посреди транзакции. Очевидно, что при нарушении этого требования блокировка будет удерживаться до тех пор, пока пользователь не заполнит все поля формы и не нажмет все необходимые кнопки. При этом совершенно неважно, каким образом транзакция инициирована: средствами управления транзакциями той среды программирования, с помощью которой разрабатывалось клиентское приложение, или командами сервера.

Также недопустима вставка внутрь транзакции оператора вывода сообщения, на которое пользователь должен хоть как-то отреагировать — пусть даже простым нажатием клавиши или кнопки подтверждения. В этом случае SQL-сервер сохраняет все установленные блокировки до тех пор, пока пользователь, который мог уйти на обеденный перерыв или вообще покинуть офис на целый день, не вернется и не нажмет кнопку **OK**.

Предположим, что имеется бизнес-правило по поводу согласования между собой данных, вводимых в какую-то из форм, которые в свою очередь служат исходной информацией для некоторой транзакции. Если при этом не нужна проверка того обстоятельства, что эти данные как-то согласованы с данными, уже хранящимися на сервере, то такое бизнес-правило нужно реализовывать на клиенте, проверяя его перед инициацией транзакции.

Взаимодействие с пользователем в процессе выполнения транзакции косвенно может произойти следующим образом. Предположим, что пользователь заполнил данными какую-то сложную форму и нажал кнопку, сигнализирующую об этом обстоятельстве. По нажатию этой кнопки инициируется транзакция. Если во введенные данные вкралась какая-нибудь ошибка, кото-

рая привела к нарушению одного из ограничений целостности базы данных, то вся транзакция будет откачена, пользователю же нужно будет среагировать на сгенерированное сообщение, исправив ошибку и снова инициировав транзакцию. Ясно, что такой способ действий малоэффективен. Поэтому нужно стараться проектировать приложение так, чтобы ограничения целостности базы данных не строились на данных, которые невозможно проверить вне рамок транзакций, особенно если эти транзакции являются продолжительными или часто выполняемыми. Помочь здесь могут со стороны клиента всевозможные приемы контроля вводимых данных с использованием средств форматирования, например, маски и шаблоны ввода, списки подстановки, правильный выбор типов данных на клиенте. На сервере нужно стараться, чтобы основные ограничения целостности базы данных, в первую очередь, такие как ограничения ключей, строились не на данных, непосредственно вводимых пользователем, а на данных, полученных в процессе автогенерации, т. е. на суррогатных ключах.

Рассуждения по поводу последней рекомендации звучат красиво, но как это может быть реализовано на практике, когда пользователю нужно взаимодействовать с сервером в процессе выполнения транзакции?

Во-первых, как быть с проверками, результаты которых пользователь должен видеть и на их основании принимать решения о том, какие данные куда вводить?

Во-вторых, отсутствие взаимодействия с сервером во время заполнения формы приводит к тому, что заносить данные, вводимые в форму, можно только в свободные поля этой формы, для которых не указан источник данных. Как в этом случае организовать, например, ввод товарных позиций в расходную накладную, если заранее неизвестно даже их количество? Если все же это необходимо сделать, то полезным может оказаться следующий прием. Из вводимых в локальные элементы управления клиентской формы данных можно сформировать строку, которая передается на сервер в результате выполнения короткой транзакции. Дальше на сервере эта строка разбирается и записывается, например, в столбец соответствующей таблицы. Разумеется, контроль целостности данных остается на совести того, кто эти данные вводил. В листинге 6.7 приведена пользовательская функция, которая принимает своим аргументом строку и разбирает ее на слова, используя пробелы в качестве разделителя, после чего слова помещаются в столбец таблицы.

Листинг 6.7. Функция, разбивающая строку на последовательность значений столбца

```
CREATE FUNCTION MY_FUN (@String nvarchar(300))
RETURNS @table TABLE
```

```
(Number int Identity (1,1) not null, Value Nvarchar(30))
 AS
BEGIN
DECLARE @Str1 nvarchar(500), @pos int
Set @Str1=@String
While 1>0
       BEGIN
       SET @POS=CHARINDEX(' ',@STR1)
       IF @POS >0
              BEGIN
              INSERT INTO @table VALUES(SUBSTRING(@STR1,1,@POS))
              SET @STR1=SUBSTRING(@STR1,@POS+1,500)
              END
       ELSE
              BEGIN
              INSERT INTO @table VALUES(@STR1)
              BREAK
              END
       END
RETURN
END
```

В заключение можно, например, попытаться представить, как должна выглядеть транзакция и соответствующая ей форма по вводу данных о расходной накладной для базы данных МАТ_УЧЕТ. Напомним, что в этой транзакции одновременно заполняются две таблицы, связанные ограничением внешнего ключа. При этом количество записей, которые будут вставлены в таблицу РАСХОДЫ_ТОВАРЫ, неизвестно заранее, т. к. товарных позиций в расходной накладной может быть произвольное число. Кроме того, пользователю неплохо бы было видеть значения остатков каждого товара — это вовлекает в транзакцию еще одну таблицу ТОВАРЫ. В форме удобно иметь список подстановки по полю Товар, который можно также взять из таблицы ТОВАРЫ. Все вместе это может выглядеть примерно так, как показано на рис. 6.1.

Простейший способ реализовать данную транзакцию в рамках ADP-проекта (Access Data Project, проект Microsoft Access) может быть следующим. Открывается главная форма транзакции. Эта форма может быть создана простейшими методами визуального программирования с использованием элемента управления **Подчиненная форма**. В этом случае таблица РАСХОДЫ является источником данных главной формы, а таблица РАСХОДЫ_ТОВАРЫ выступает источником данных подчиненной формы. При открытии главной формы на сервер поступает команда о начале транзакции. Эта команда привязана к событию загрузки главной формы. По наступлению события закрытия главной

формы на сервер посылается команда фиксации транзакции. Соответствующие процедуры обработки событий приведены в листинге 6.8 без комментариев, т. к. изучение методов программирования на клиенте выходит за рамки данной книги.

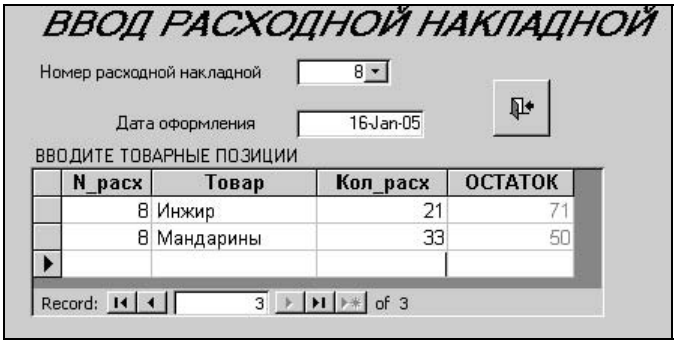


Рис. 6.1. Форма, связанная со сложной транзакцией

Листинг 6.8. Инициация транзакции с клиента средствами сервера

```
' Процедура обработки события загрузки формы
Srivate Sub Form_Load()
Dim cnn1 As ADODB.Connection
Dim cmd As New ADODB.Command
 Set cnn1 = CurrentSroject.Connection
 cmd.ActiveConnection = cnn1
 cmd.CommandText = "BEGIN TRAN"
 cmd.CommandType = adCmdText
       cmd.Execute
End Sub
' Процедура обработки события закрытия формы
Srivate Sub Form_Close()
Dim cnn1 As ADODB.Connection
Dim cmd As New ADODB.Command
 Set cnn1 = CurrentSroject.Connection
 cmd.ActiveConnection = cnn1
 cmd.CommandText = "COMMIT TRAN"
 cmd.CommandType = adCmdText
       cmd.Execute
End Sub
```

Разумеется, это не весь необходимый для реализации транзакции клиентский код. Опущена работа с таблицей `ТОВАРЫ` и не уточнен механизм проверки со-

гласования отпущенного количества товара и остатков. Но, во всяком случае, использованный прием позволил объединить операторы вставки записей в подчиненной таблице РАСХОДЫ_ТОВАРЫ в одну транзакцию, чего не было, когда в *разд. 5.11* рассматривалось применение триггеров. Дальше с целью проверки согласования отпущенного количества товара и остатков для оператора вставки записи в таблицу РАСХОДЫ_ТОВАРЫ можно создать необходимый триггер по аналогии с листингом 5.40. Можно создать проверяемое ограничение по примеру листинга 6.4 или запрограммировать эту проверку на клиенте.

Заметим, что если заполнение формы не представить как целостную транзакцию, то, открыв другое соединение с сервером легко можно увидеть, что таблицы ТОВАРЫ, РАСХОДЫ и РАСХОДЫ_ТОВАРЫ доступны для изменений. То есть в этом случае пока один сотрудник, открыв форму, уговаривает клиента купить какой-то товар, этот товар может оказаться проданным из другого соединения.

Практический выход из этих противоречивых тенденций заключается в том, что обычно, если реальных проблем производительности не возникает и не предвидится, то можно на первый план ставить простоту программирования как клиентской, так и серверной части приложения и удобство работы с интерфейсом. Если же ситуация, когда состояние транзакции удерживается на все время заполнения формы, становится неприемлемой по быстродействию из-за возросшего соперничества за ресурсы, то нужно решать, как в каждом конкретном случае можно преодолеть конкретную проблему.

В *главе 5* звучали предостережения по поводу использования серверных курсоров. Но клиентские курсоры — совсем другое дело. Построчная обработка клиентской программой набора результатов, которые были сгенерированы SQL-сервером с использованием операции, ориентированной на наборы данных, в определенных случаях является приемлемым решением. С точки зрения механизма транзакций, клиентские курсоры могут разгрузить систему особенно сильно, если они используются для формирования сложных отчетов, в которых можно пренебречь несущественными последними изменениями, сделанными в базе данных в то время, пока отчет генерируется. В этом случае клиентский курсор заполняется в результате выполнения короткой транзакции с минимальным уровнем изоляции, после чего к серверу в процессе генерации отчета можно вообще не обращаться.

Доведение этой рекомендации до логического завершения может быть описано следующим образом. Пусть, например, на сервере имеется базовая таблица, в которой хранятся данные, описывающие какую-нибудь иерархическую или сетевую структуру. По ходу функционирования приложения для этих данных необходимо решать некую оптимизационную задачу. Как из-

вестно, алгоритмы решения задач подобного рода выполняются достаточно долго, обычно требуя более сложного доступа к данным таблицы, чем простой последовательный перебор, т. е. потенциально эти алгоритмы можно реализовывать только на медленно работающих курсорах с произвольным доступом. Зачастую вообще реализовать подобные алгоритмы средствами T-SQL крайне затруднительно, т. к. чаще всего они разрабатываются для такой структуры данных, как массивы, а этой структуры данных в T-SQL нет вообще. Следовательно, подобные задачи эффективнее решать на клиенте, поступая, например, следующим образом. Сначала нужно установить соединение с сервером, получив необходимые данные в контексте этого соединения, заполнив ими клиентский курсор (объект **Recordset**). После этого может быть разумным создать обычный массив на клиенте, заполнить его данными из курсора, структурировав их нужным образом. Затем соединение с сервером можно закрыть. Оптимизационная задача будет спокойно решаться на клиенте без блокировки таблицы на сервере, используя стандартный алгоритм, написанный и отлаженный для такой привычной структуры данных, как массив.

Разумеется, необходимо изучать документацию по программному интерфейсу конкретной среды разработки, чтобы применять разнообразные клиентские курсоры максимально эффективно. Например, ADO-соединение с сервером, которое характерно для многих сред разработки клиентских приложений, поддерживает четыре типа курсоров: курсор типа "только вперед", используемый только для чтения данных, статический, ключевой и динамический. Конечно, курсор типа "только вперед", называемый также шлангом, наиболее эффективен и связан с наименьшими накладными расходами. Однако зачастую возникает искушение воспользоваться более удобными, но и более ресурсоемкими курсорами, такими как ключевой или динамический. Поэтому в том случае, когда, например, стоит задача обеспечить лишь базовые возможности просмотра, следует подумать — а стоит ли прибегать к таким мощным средствам?

Оправданность использования объектов типа **Recordset** или клиентских курсоров для всех добавлений или модификаций данных менее бесспорна. Безусловно, применять объекты **Recordset** для обновления базы данных легко и приятно. Но этот метод связан с открытием гораздо более ресурсоемкого курсора, чем курсор, предназначенный только для чтения, а это приведет к дополнительным затратам ресурсов.

Чаще более эффективным окажется обновление базы данных с использованием непосредственно операторов SQL, особенно при множественных операциях вставки, обновления и удаления. Упомянутые операторы SQL для снижения трафика сети и использования возможностей кэширования планов выполнения рекомендуется размещать внутри хранимых процедур. Эти хра-

нимые процедуры вызываются с клиента, снабжаясь необходимыми параметрами. Однако хочется отметить, что прямолинейное следование рекомендации, согласно которой общение клиента с сервером происходит только путем вызова хранимых процедур, при несомненных преимуществах может привести и к некоторым проблемам. Предположим, что приложение активно формирует большое количество разнообразных коротких, с точки зрения их кода, и простых, с точки зрения выполнения, операторов `INSERT`, `DELETE` и `UPDATE` с параметрами и без, каждый из которых является самостоятельной транзакцией. Разумеется, все эти операторы могут быть выполнены только на сервере. Если каждый из них оформлять как отдельную хранимую процедуру, то окажется, что достаточно простой по сути код приложения в серверной его части будет размазан по большому количеству коротких хранимых процедур. Чем больше этих процедур, тем сложнее ими управлять, а само приложение становится все менее обозримым, т. к. в нем может быть сгенерировано больше служебного кода для синтаксического оформления хранимых процедур, чем собственно функциональности, которую реализуют операторы модификации данных. Разумной альтернативой в данном случае может служить кодирование этих операторов непосредственно на клиенте с использованием для этих целей той технологии встроенного (embedded) SQL, которую поддерживает соответствующая среда разработки.

ПРИЛОЖЕНИЕ 1

Средства мониторинга производительности SQL-сервера

В состав SQL-сервера входит ряд средств, которые вместе с определенными средствами операционной системы можно использовать для анализа различных аспектов его производительности. Можно выделить два уровня этого анализа, а именно мониторинг производительности сервера на *уровне операционной системы* и *на уровне самого сервера*.

Мониторинг на уровне операционной системы — это мониторинг на наиболее высоком уровне, в задачи которого входит установление адекватности между наличными программными средствами и тем набором задач, которые предстоит выполнять. Мониторинг этого уровня осуществляется непосредственно с помощью средств операционной системы, т. е. используя утилиты, которые являются стандартными административными средствами той операционной системы, поверх которой функционирует SQL-сервер. При этом полезными могут оказаться следующие утилиты, о которых будет приведено только краткое упоминание.

- Microsoft Event Viewer.
- SQL Server Performance Monitor.

Утилита Microsoft Event Viewer отображает сообщения об ошибках, предупреждения и информационные сообщения. Кроме того, она позволяет просматривать такие журналы событий операционной системы, как:

- Windows NT Application Log (журнал приложений) — в этот журнал заносятся сведения о событиях, к которым относятся и события, происходящие на SQL-сервере;
- Windows NT Security Log (журнал безопасности) — в этот журнал записывается информация о загрузке системных компонентов;
- Windows NT System Log (системный журнал) — в этот журнал записываются данные о событиях, связанных с работой системных компонентов.

Утилита SQL Server Performance Monitor представляет собой вариант настройки утилиты Windows Performance Monitor, при котором отображается заранее определенный набор счетчиков, относящихся непосредственно к функционированию SQL-сервера как приложения. Эти счетчики сохраняются в файле с расширением pmc. Во время установки SQL-сервера программа SQL Server Setup автоматически создает дополнительные объекты утилиты Windows Performance Monitor, предназначенные для измерения производительности SQL-сервера. Мониторинг производительности, проводимый подобным методом, заключается в задании набора счетчиков, значения которых отображают определенную информацию, порождаемую событиями, возникающими при функционировании SQL. Эти события связаны, в первую очередь, с использованием ресурсов операционной системы SQL-сервером. Используя утилиту SQL Server Performance Monitor, можно подсчитать количество событий определенного типа, которые наиболее тесно связаны с вопросами производительности. Эти данные могут сохраняться и отображаться либо в графическом, либо в текстовом виде, т. е. таким образом, который максимально удобен для их последующего анализа.

Набор счетчиков утилиты SQL Server Performance Monitor, из которого можно выбирать, довольно велик. Полный перечень этих счетчиков приводится в BOL, для поиска можно использовать критерий "Using SQL Server Objects".

Следующий уровень мониторинга производительности SQL-сервера реализуется с помощью специальных утилит для отслеживания специфических моментов работы сервера, которые наиболее важны для его оптимального функционирования. К этим моментам относятся, в первую очередь, показатели соперничества за ресурсы, связанные, например, с блокировками данных или использованием базы данных `TEMPDB`. Для мониторинга на уровне сервера используются следующие программные средства:

- утилита SQL Profiler;
- окно **Current Activity** приложения SQL Server Enterprise Manager;
- системные хранимые процедуры;
- операторы языка T-SQL.

Использование утилиты SQL Profiler

Утилита SQL Profiler представляет собой наиболее универсальное и мощное средство, с помощью которого можно выполнять комплексный анализ практически всех аспектов, связанных с оптимизацией производительности функционирования SQL-сервера. Например, эта утилита позволяет отслеживать конкретные операции сервера и базы данных, в том числе операции подключения, действия пользователя, а также операции клиентских приложений сер-

вера. При этом можно задать сбор сведений в базовую таблицу SQL-сервера, в файл или создать сценарий проведения мониторинга.

Собирать данные можно о самых разных событиях, затем эти данные фильтруются и группируются по смыслу, сохраняя лишь самую существенную информацию. В качестве примеров отслеживаемых событий приводятся следующие типы событий:

- медленно выполняющиеся запросы;
- запросы, требующие сканирования таблиц;
- действия отдельных пользователей и приложений;
- случаи взаимоблокировки;
- попытки подключения к серверу и их результаты;
- логические операции чтения и записи на диск;
- режимы использования процессора на уровне операторов;
- процесс кэширования данных;
- ошибки определенных уровней;
- время ожидания для событий конкретных типов.

Для активизации утилиты SQL Profiler можно воспользоваться опцией **Tools** основного меню утилиты SQL Server Enterprise Manager. Кроме того, SQL Profiler является самостоятельным приложением, зарегистрированным в операционной системе, следовательно, данную утилиту можно вызвать из главного меню Windows. Следует только позаботиться о том, чтобы при установке клиентских компонентов SQL-сервера не был упущен компонент под названием SQL Profiler.

Не нужно забывать, что утилита SQL Profiler представляет собой всего лишь графический интерфейс для вызова специальных системных хранимых процедур, предназначенных для работы со следами трассировки. К таким процедурам относятся процедуры, названия которых начинаются с префикса `sp_trace...` Иногда использование хранимых процедур предпочтительнее, чем непосредственный вызов SQL Profiler. Во-первых, процедуры работают быстрее, а во-вторых, более гибко. Самое очевидное преимущество непосредственного использования хранимых процедур состоит в возможности настройки задания утилиты SQL Server Agent на периодический кратковременный запуск и остановку процесса трассировки в автоматическом режиме. Обычно именно таким способом создаются результаты трассировки, которые предполагается использовать как входные данные для Мастера Настройки Индексов.

Утилита SQL Profiler имеет средство, позволяющее автоматически создать T-SQL-сценарий, который описывает параметры и настройки созданного с ее

помощью следа трассировки на языке системных хранимых процедур. Чтобы это сделать, необходимо в главном меню SQL Profiler выбрать опции **File/Script Trace...**, указать версию SQL-сервера и задать имя текстового файла, в котором будет сохранен сгенерированный программный код.

После запуска утилиты SQL Profiler появляется ее главное меню, работу с которым можно начинать с создания нового следа трассировки.

След трассировки представляет собой таблицу специального формата, в которой находит свое отражение информация о событиях заданного типа, происходивших на сервере во время выполнения трассировки. Подробнее таблицы, содержащие след трассировки, будут рассмотрены далее.

Для создания следа трассировки нужно активизировать опции **File/New** основного меню утилиты, после чего выбрать параметр **Trace...**. След трассировки может быть создан на основе так называемого шаблона. Имеется готовый набор шаблонов, которые будут доступны для выбора при задании свойств следа трассировки. Можно также создать и пользовательский шаблон трассировки. Для этого нужно после выбора опции **File/New** основного меню указать параметр **Trace Template...**, после чего указать имя файла, в котором шаблон будет сохранен (расширение этого файла tdf), и настроить свойства шаблона по аналогии с настройкой свойств следа трассировки в окне **Trace Properties**. После этих действий пользовательский шаблон появится в списке шаблонов окна **Trace Properties** под именем, которое совпадает с именем файла шаблона.

После того как выбрано действие по созданию нового следа трассировки, первым делом происходит соединение с сервером, и выполняется одна из стандартных процедур аутентификации подключения, о чем сигнализирует соответствующее окно. Если аутентификация произошла успешно, то появится следующее окно под названием **Trace Properties**, в котором можно задать свойства создаваемого следа трассировки. Указанное окно имеет четыре вкладки, называемые **General**, **Events**, **Data Columns** и **Filters** соответственно. Кратко опишем назначение этих вкладок.

Вкладка **General** служит для задания общих свойств следа трассировки, к которым относятся:

- **Trace Name** — имя создаваемого следа;
- **Trace SQL Server** — имя экземпляра SQL-сервера;
- **Template Name** — имя шаблона, с помощью которого будет создан след трассировки. Под шаблоном в данном случае понимается заранее определенный набор параметров следа трассировки. При необходимости эти параметры можно изменить, работая с остальными тремя вкладками окна **Trace Properties**;

- **Template File Name** — имя файла, в котором хранится информация о параметрах соответствующего шаблона. Пользователь может сам создавать шаблоны, сохранять их в файлах и использовать по мере надобности;
- **Save to File** — показывает, что результаты трассировки будут сохранены в файле;
- **Save to Table** — показывает, что результаты трассировки будут сохранены в базовой таблице SQL-сервера;
- **Enable Trace Stop Time** — служит для задания времени прекращения трассировки.

Остановимся подробнее на способах сохранения следа трассировки. Если выбирается сохранение следа трассировки в файл, то, разумеется, нужно указать имя файла. Стандартное зарегистрированное в операционной системе расширение для файлов этого типа — trc. Файлы, содержащие след трассировки, можно просматривать с помощью утилиты SQL Profiler. Кроме того, файлы со следом трассировки могут являться входными данными для Мастера Настройки Индексов утилиты SQL Query Analyzer. Интересно, что даже если предполагается оптимизация индексов на основании следа трассировки, то текстовое окно утилиты SQL Query Analyzer нельзя оставить совершенно пустым — для активизации опции меню **Index Tuning Wizard** необходимо ввести в это окно хоть какой-нибудь текст.

При выборе сохранения следа трассировки в файл становятся доступными еще несколько опций (флажков), связанных с ограничением размера файла следа. Можно задать непосредственно максимально допустимый размер этого файла (опция **Set Maximum file size (MB)**). Задание опции **Enable file rollover** приводит к созданию новых файлов следа трассировки по мере достижения текущим файлом максимально допустимого размера. Задание опции **Server processes SQL Server trace data** предотвращает возможную потерю событий, которая может произойти при интенсивном заполнении файла следа трассировки в том случае, когда SQL Profiler не успевает перехватывать и заносить в выходной поток все события.

Если местом сохранения следа трассировки выбрана базовая таблица SQL-сервера, то после соединения с сервером нужно указать как имя базы данных, так и имя таблицы (**Destination Table**). Разумеется, для того чтобы создавать таблицы со следом трассировки, нужно обладать соответствующими правами в контексте сервера или базы данных.

Просмотр следа трассировки, сохраненного в базовой таблице, также можно выполнять с помощью утилиты SQL Profiler. Кроме того, эта таблица обрабатывается любыми средствами, которые вообще применимы к базовым таблицам. Так, содержимое таблицы со следом трассировки можно просматривать

с помощью утилиты Enterprise Manager. К ней также применяется оператор выборки `SELECT` с использованием, например, утилиты SQL Query Analyzer. То есть при сохранении следа в базовую таблицу появляются мощные дополнительные средства фильтрации данных с применением стандартных средств языка T-SQL, что очень удобно и позволяет легко находить информацию о необходимых событиях.

При этом приходится иметь в виду, что существуют небольшие отличия в формате данных при просмотре таблицы следа трассировки утилитой Enterprise Manager по сравнению с ее оригинальным форматом, который имеет место при использовании утилиты SQL Profiler. В частности, первое подобное различие, которое сразу бросается в глаза, состоит в том, что если SQL Profiler отображает информацию о классе события (**EventClass**) названием, то Enterprise Manager выводит числовое значение соответствующего кода, что достаточно неудобно.

Обойти данную проблему не очень сложно. В BOL имеется таблица, содержащая коды классов событий и их расшифровку. Найти эту таблицу можно, например, на странице, посвященной системной хранимой процедуре `sp_trace_setevent`. Далее следует скопировать эту таблицу на рабочий лист MS Excel, после чего с помощью мастера DTS SQL-сервера эти данные можно легко трансформировать в базовую таблицу сервера и использовать ее для написания запросов с участием таблиц, содержащих следы трассировки.

Если имеется необходимость преобразовать информацию о следе трассировки в формат другого типа, то проще всего это сделать через буфер обмена. Например, если открыть в SQL Profiler таблицу или файл следа трассировки и занести данные в буфер обмена, то после вставки его содержимого в рабочий лист MS Excel все данные отобразятся достаточно прозрачно в виде таблицы с сохранением всех столбцов и строк. После этого можно воспользоваться обширными стандартными возможностями для любого преобразования форматов данных.

При выборе способа сохранения следа трассировки следует принимать во внимание, что заполнение файла следа происходит быстрее, чем базовой таблицы. С другой стороны, для работы с файлом следа трассировки годится только утилита SQL Profiler, а ее может не быть на каждом клиентском компьютере, в то время как те или иные средства работы с базовыми таблицами обычно имеются везде.

Следующие три вкладки окна **Trace Properties** служат для задания параметров следа трассировки.

Вкладка **Events** позволяет выбрать те типы событий, происходящих на сервере, данные о которых предполагается изучать. Для включения события в след

трассировки нужно развернуть соответствующую категорию в левом списке окна и с помощью кнопки **Add** перенести событие в правый список. Кнопка **Remove** служит для удаления названия события из следа трассировки. Используя эти кнопки, можно включить в след трассировки выбранную категорию событий целиком. События сгруппированы по категориям, в табл. П1.1 перечислены названия категорий событий и кратко поясняется их смысл.

Таблица П1.1. *Основные события утилиты SQL Profiler*

Категория событий (Event category)	Описание
Cursors	События, связанные с работой курсоров
Database	События, связанные с автоматическим изменением размеров файлов базы данных и журналов транзакций
Errors and Warnings	События, связанные с возникновением всевозможных ошибок и предупреждений
Locks	События, связанные с блокировками ресурсов
Objects	События, связанные с автоматическим обновлением статистики, а также открытием, закрытием и удалением объектов базы данных
Performance	События, связанные с анализом параллелизма и планов выполнения запросов
Scans	События, связанные с выполнением операций полного сканирования таблиц
Security Audit	События, связанные с вопросами безопасности сервера
Sessions	События, связанные с созданием и разрывом соединений клиентов с сервером
Server	События, связанные с изменением объема памяти, используемой сервером
Stored Procedures	События, связанные с выполнением хранимых процедур и кэшированием данных
Transactions	События, связанные с процессом выполнения транзакций
T-SQL	События, связанные с подготовкой пакетов операторов к выполнению и собственно с их выполнением
User Configurable	События трассировки, которые создает сам пользователь

С помощью следующей вкладки окна **Trace Properties**, которая называется **Data Columns**, можно выбирать столбцы таблицы следа трассировки, которые будут содержать интересующую нас информацию о событиях, выбран-

ных с помощью предыдущей вкладки. Полный перечень столбцов данных следа трассировки не приводится, для примера в табл. П1.2 указаны некоторые наиболее часто используемые столбцы данных следа трассировки.

Таблица П1.2. *Наиболее часто используемые столбцы следа трассировки*

Столбец данных	Описание
Application Name	Имя клиентского соединения, установившего связь с сервером
ClientProcessID	Идентификатор, присвоенный клиентскому соединению
CPU	Время в миллисекундах, затраченное центральным процессором на определенные действия
Database ID	Системный идентификатор базы данных, присвоенный ей при выполнении оператора `USE` в контексте текущего соединения, который можно узнать с помощью функции `DB_ID`
DatabaseName	Имя базы данных, в контексте которой выполняется оператор
DBUserName	Имя клиента SQL-сервера
Duration	Полное время в миллисекундах, затраченное на выполнение различных действий. Обычно взаимосвязано с такими столбцами данных, как **Start Time** и **End Time**
End Time	Системное время завершения определенного события
Error	Код ошибки, если таковая имела место. Часто этот код хранится в системной таблице `sysmessages`
EventClass	Тип события
Host Name	Имя компьютера, на котором выполняется клиентское приложение
Index ID	Системный идентификатор индекса, затронутого при наступлении какого-то события. Этот идентификатор можно определить с помощью столбца `indid` системной таблицы `sysindexes`
LoginName	Имя логина, использованного при подключении к серверу
LoginSid	Системный идентификатор безопасности подключения SID (Security Identifier) из системной таблицы `sysxlogins`
NestLevel	Значение глобальной переменной `@@NESTLEVEL` при ее изменении
NT Domain Name	Имя домена операционной системы Microsoft Windows NT® 4.0 или Windows 2000, к которому принадлежит пользователь, подключившийся к серверу
NT User Name	Имя пользователя операционной системы Windows NT 4.0 или Windows 2000

Таблица П1.2 *(окончание)*

Столбец данных	Описание
Object ID	Системный идентификатор объекта, затронутого при наступлении какого-то события. Этот идентификатор можно определить с помощью системной таблицы `sysobjects`
ObjectName	Имя объекта (*см. системную таблицу* `sysobjects`)
ObjectType	Тип объекта (*см. системную таблицу* `sysobjects`)
Owner Name	Системное имя владельца объекта
Reads	Количество элементарных операций чтения диска, выполненных в течение события
Server Name	Имя экземпляра SQL-сервера, в контексте которого произошло событие
SPID	Системный идентификатор процесса, присвоенный сервером клиентскому процессу
Start Time	Системное время начала события
Success	Выводит данные о том, было ли событие успешно завершено (1 — да /0 — нет)
TextData	Текстовое представление оператора, породившего событие
Transaction ID	Системный идентификатор, присвоенный транзакции
Writes	Количество элементарных операций записи на диск, выполненных в течение события

Последняя вкладка окна **Trace Properties**, называемая **Filters**, позволяет задавать параметры отбора событий, которые попадут в след трассировки. Без использования фильтра в след трассировки попадут все события заданного вида. Разумеется, при этом размеры таблицы, содержащей информацию о результатах трассировки, могут оказаться очень большими, и разобраться в ней будет крайне затруднительно. Задав фильтр, можно существенно сократить ее объем, т. к. зачастую большая часть событий не представляет интереса, с точки зрения оптимизации производительности.

Фильтр представляет собой набор условий, которым должны удовлетворять данные, находящиеся в столбцах **Data column**. Имеется флажок под названием **Exclude system Ids,** который позволяет сразу отключать занесение в след трассировки информации о системных событиях. Условия фильтрации событий задаются с помощью простейших логических предикатов `Equals` (Равно), `Not equal to` (Не равно), `Greater then or Equal` (Больше или равно) и `Less`

`then or Equal` (Меньше или равно). Для перечисленных предикатов можно задать по одному значению, используемому в операции отношения.

Для текстовой информации, например, для названия приложения (**ApplicationName**) допустимы предикаты `Like` и `Not like`, причем для обоих предикатов можно задать список значений. При создании этого списка для того, чтобы получить доступ к его следующему элементу, нужно несколько раз щелкнуть на значке раскрытия поддерева (плюс).

Конструирование логического условия для фильтра несколько непривычно, т. к. логические предикаты нужно соединять графическими средствами, за которыми трудно сразу угадать привычные логические функции. Разобраться в этом достаточно легко, если сравнить графически построенные фильтры с результатами автоматического создания T-SQL-скрипта этих фильтров, зная смысл соответствующих параметров системной хранимой процедуры `sp_trace_setfilter`. Не приводя полного описания этих параметров, покажем результаты конструирования фильтров на примере сравнения графически заданных фильтров и кода программы (рис. П1.1 и листинг П1.1).

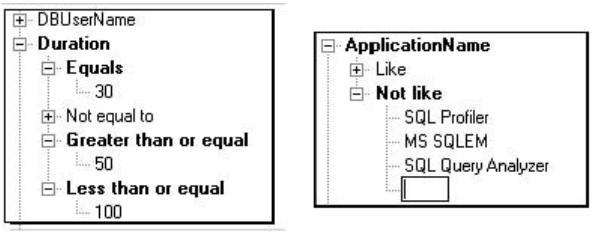


Рис. П1.1. Задание предикатов фильтрации SQL Profiler визуальными средствами

Листинг П1.1. Соответствие графических опций фильтра программным

```
/* Фильтр по Application Name (второй параметр равен 10).
Третий параметр, равный 0, соответствует логической
операции AND, четвертый параметр, равный 7, — это предикат Not Like */
exec sp_trace_setfilter @TraceID, 10, 0, 7, N'SQL Profiler'
exec sp_trace_setfilter @TraceID, 10, 0, 7, N'SQL Query Analyzer'
exec sp_trace_setfilter @TraceID, 10, 0, 7, N'MS SQLEM'
/* Задание фильтра по Duration (второй параметр равен 13).
Третий параметр, равный 0/1, соответствует логическим операциям AND/OR,
а четвертый параметр задает тип операции сравнения.
В результате получено логическое условие
Duration between 50 And 100 Or Duration = 30 */
set @bigintfilter = 50
exec sp_trace_setfilter @TraceID, 13, 0, 4, @bigintfilter
```

```
set @bigintfilter = 100
exec sp_trace_setfilter @TraceID, 13, 0, 5, @bigintfilter
set @bigintfilter = 30
exec sp_trace_setfilter @TraceID, 13, 1, 0, @bigintfilter
```

Применение фильтров практически всегда оправдано, т. к. никогда не следует собирать данные о тех событиях, анализ которых не нужен для поиска путей повышения производительности проблемных мест. В первую очередь, это относится к неизбежно происходящим системным событиям, которые генерируются такими компонентами системы, как Enterprise Manager, сам SQL Profiler, SQL Server Agent или SQL Query Analyzer. Чтобы исключить из следа трассировки события перечисленного типа, нужно воспользоваться опцией фильтра **ApplicationName** с параметром `Not like`, например, так, как это и показано на рис. П1.1.

При анализе общей производительности SQL-сервера полезно обратить внимание на следующие события таких классов, как:

- **Execution Warnings**. Этот класс событий имеет такие два события, как **Query Wait** и **Query Time-Out**. Первое из этих событий показывает, как часто наступает ситуация, когда запрос не может быть немедленно выполнен из-за отсутствия необходимых ресурсов и вынужден ждать их освобождения. Второе событие данного класса показывает, насколько часто возникает ситуация, связанная с превышением заданного в системе порога ожидания для запросов. Очевидно, что чем меньше возникает событий подобного типа, тем лучше;
- **Hash Warning**. Это событие показывает наступление ситуации, называемой **hash recursion** (событие 0) либо ситуации, называемой **hash bail** (событие 1). Событие **hash recursion** происходит, когда результаты запроса не помещаются целиком в оперативную память, а вынуждены обрабатываться отдельными частями. Наступление события **hash bail** влияет на производительность еще хуже, т. к. оно показывает, что операция хеширования достигает своей максимально допустимой глубины вложения, следовательно, запрос вообще не может быть выполнен по плану с использованием этой операции, и вся автоматическая оптимизация запроса оказывается сомнительной. Для запросов с событием **hash bail** в том случае, если они оказываются критическими, требуется тщательная ручная оптимизация с учетом всей статистической информации, настройкой индексов и, возможно, с использованием подсказок оптимизатору;
- **Missing Column Statistics**. Это событие показывает, что по каким-либо причинам статистика, полезная при оптимизации плана запроса, потеряна или не создана для столбца или индекса;

- **Missing Join Predicate**. Показывает, что у запроса потеряна опция `JOIN`, что не позволяет проводить оптимизацию запроса должным образом;
- **Sort Warnings**. Показывает, что сортировка не может быть выполнена целиком в оперативной памяти, поэтому ее нужно разделить на несколько этапов, выполняя при этом свопинг на диск. Понятно, что такая сортировка выполняется потенциально медленно, поэтому нужно либо попытаться от нее избавиться, либо добавить памяти для SQL-сервера.

Разумеется, практически не представляется возможным настраивать оптимальным образом все объекты базы данных. Однако полезно выявить те запросы, таблицы и индексы, которые используются максимально интенсивно, и уделить хоть какое-то внимание их оптимизации. Для этого удобно использовать класс событий **Object:Opened**, предварительно сгруппировав данные в столбце под названием **TextData**.

Для наиболее типичных случаев существуют готовые шаблоны, например:

- **TSQL.** Этот шаблон служит для сбора информации об операторах T-SQL в том порядке, в котором они выполняются.
- **TSQL (Grouped).** Этот шаблон аналогичен предыдущему с той лишь разницей, что данные в нем сгруппированы по **Application name**, **Windows NT user name**, **login name** и **client process ID**.
- **SP Count.** Собирает информацию о том, какие хранимые процедуры запускаются и как часто это происходит. Чем чаще вызывается хранимая процедура, тем более верным кандидатом на оптимизацию она является.
- **TSQL SP.** Отслеживает запуск хранимых процедур вместе с операторами T-SQL, встречающимися внутри них. Результаты отсортированы по времени наступления событий.
- **TSQL by Duration.** Отслеживает операторы T-SQL по мере их выполнения вместе с их продолжительностью. Этот шаблон полезен для выявления запросов, которые долго выполняются.
- **TSQL For Replay.** Отбирает события, необходимые для репликации следа трассировки между SQL-серверами.
- **Tuning.** Отслеживает события, связанные с хранимыми процедурами и операторами T-SQL, с высокой степенью детализации для последующего анализа.
- **Standard.** Стандартный шаблон, принятый по умолчанию.

Полный перечень событий утилиты SQL Profiler, связанных с трассировкой выполнения хранимых процедур, приведен в табл. П1.3.

Таблица П1.3. *События для трассировки выполнения хранимых процедур*

Класс события (EventClass)	Код события	Описание
RPC Output Parameter	100	Генерация выходного параметра для удаленного вызова процедуры
RPC:Completed	10	Завершение удаленного вызова процедуры
RPC:Starting	11	Начало удаленного вызова процедуры
SP:CacheHit	38	Хранимая процедура найдена в буферном кэше
SP:CacheInsert	35	Хранимая процедура вставлена в буферный кэш
SP:CacheMiss	34	Хранимая процедура не найдена в буферном кэше
SP:CacheRemove	36	Хранимая процедура удалена из буферного кэша
SP:Completed	43	Хранимая процедура завершилась
SP:ExecContextHit	39	Выполнимая версия хранимой процедуры для иного контекста найдена в буферном кэше
SP:Recompile	37	Хранимая процедура перекомпилирована
SP:Starting	42	Хранимая процедура стартовала
SP:StmtCompleted	45	Оператор из хранимой процедуры завершился
SP:StmtStarting	44	Оператор из хранимой процедуры стартовал

Не рекомендуется запускать утилиту SQL Profiler на том же компьютере, на котором находится сам SQL-сервер, подвергающийся мониторингу, т. к. это сильно снижает общую производительность. Лучше запускать SQL Profiler с рабочей станции или другого сервера и сохранять результаты трассировки там же. Особенно это важно, если след трассировки сохраняется в базовой таблице. В этом случае рекомендуется создать отдельный экземпляр SQL-сервера, который будет находиться на другом компьютере, и использовать его базы данных для сохранения результатов трассировки.

Не стоит генерировать слишком большие результаты трассировки. Во-первых, результат трассировки, который накоплен в течение нескольких часов, практически невозможно осмыслить из-за его огромного объема. Во-вторых, если след трассировки предполагается использовать как входные данные для Мастера Настройки Индексов, то все равно эта утилита не сможет обработать слишком большой объем информации и многие события будут потеряны для оптимизации индексов.

ПРИЛОЖЕНИЕ 2

Классификация ограничений целостности и средства их реализации на SQL-сервере

Ограничение целостности — это некоторое утверждение, которое может быть истинным или ложным в зависимости от состояния базы данных. Любое ограничение целостности является *семантическим* понятием, т. е. появляется как следствие определенных свойств объектов предметной области и/или их взаимосвязей.

Некоторые из ограничений целостности являются ограничениями реляционной модели данных (целостность сущностей, ссылочная целостность) *(см. главу 1)*. Другие ограничения являются в достаточной степени произвольными утверждениями.

Ограничения целостности можно классифицировать несколькими способами.

- Ограничения целостности по способам реализации.
  - Декларативная поддержка ограничений целостности.
  - Процедурная поддержка ограничений целостности.
- Ограничения целостности по времени проверки.
  - Немедленно проверяемые ограничения.
  - Ограничения с отложенной проверкой.
- Ограничения целостности по области действия.
  - Ограничения домена.
  - Ограничения атрибута.
  - Ограничения кортежа.
  - Ограничения отношения.
  - Ограничения базы данных.

- Ограничения целостности по классификации Microsoft для SQL-сервера (из BOL).
  - Целостность доменов.
  - Целостность сущностей.
  - Ссылочная целостность.
  - Целостность, определяемая пользователем.

Ограничения целостности по способам реализации

Декларативная поддержка ограничений целостности заключается в их определении средствами языка определения данных DDL. Обычно на этом уровне определяются ограничения на значения доменов и атрибутов, целостность сущностей и ссылочную целостность. Стандарт SQL позволяет задавать декларативные ограничения следующими способами:

- ограничения домена;
- ограничения, входящие в определение таблицы;
- ограничения, хранящиеся в базе данных в виде независимых утверждений (assertion).

Процедурная поддержка ограничений целостности заключается в использовании триггеров и хранимых процедур. Эти объекты реализуют некоторый элемент процедурности стандартного языка SQL. Основными здесь являются так называемые *действия, исполняемые по ссылке* (referential triggered actions), которые определяют, что будет происходить при изменении значения родительского ключа, на который ссылается некоторый внешний ключ. Эти действия можно задавать независимо для операций обновления или для операций удаления записей в родительском отношении. Стандартом SQL определяется четыре типа действий, исполняемых по ссылке.

- `CASCADE`. Изменения значения родительского ключа автоматически приводят к таким же изменениям связанного с ним значения внешнего ключа. Удаление кортежа в родительской таблице приводит к удалению связанных с ним кортежей в дочерней таблице.
- `SET NULL`. Все внешние ключи, которые ссылаются на обновленный или удаленный родительский ключ, получают значения `NULL`.
- `SET DEFAULT`. Все внешние ключи, которые ссылаются на обновленный или удаленный родительский ключ, получают значения, принятые по умолчанию для этих ключей.

- `NO ACTION`. Значения внешнего ключа не изменяются. Если операция приводит к нарушению ссылочной целостности, то такая операция не выполняется.

Действия `CASCADE`, `SET NULL` и `SET DEFAULT`, исполняемые по ссылке, фактически являются встроенными в СУБД триггерами, которые реализуют компенсирующие операции и вызываются при попытке нарушить ссылочную целостность.

По сути, наличие ограничения целостности как декларативного, так и процедурного характера *всегда* приводит к созданию некоторого программного кода, реализующего это ограничение. Разница заключается в том, где такой код хранится и как он создается. Если ограничение целостности реализовано в виде триггеров, то этот программный код является просто телом триггера. Если используется декларативное ограничение целостности, то возможны два подхода: или текст ограничения хранится в виде некоторого объекта СУБД (тогда этот код представляет собой внутренние функции ядра СУБД), или при декларировании ограничения СУБД автоматически генерирует триггеры, выполняющие необходимые действия по проверке ограничений.

Если же СУБД не имеет средств реализации ни декларативной, ни процедурной поддержки ссылочной целостности, то программный код, следящий за корректностью базы данных, приходится размещать в пользовательском приложении. Это сильно затрудняет разработку программ и не защищает от попыток пользователей напрямую внести некорректные данные в базу данных. Подобные платформы на сегодняшний день считаются безнадежно устаревшими.

Замечание

Стандарт SQL 92 *не предусматривает* процедурных ограничений целостности, реализуемых при помощи триггеров и хранимых процедур. В этом стандарте отсутствует понятие "триггер", хотя триггеры имеются в большинстве промышленных СУБД SQL-типа. Таким образом, реализация ограничений средствами конкретной СУБД обладает большей гибкостью, чем с использованием исключительно стандартных средств SQL.

Ограничения целостности по времени проверки

По времени проверки допускаются как ограничения, немедленно проверяемые, так и с отложенной проверкой.

Немедленно проверяемые ограничения проверяются непосредственно в момент выполнения операции, способной нарушить ограничение. Для их проверки обычно используются внутренние функции ядра СУБД. Так, например, проверка уникальности потенциального ключа осуществляется в момент вставки записи в таблицу. Если ограничение нарушается, то такая операция

отвергается. Транзакция, внутри которой произошло нарушение немедленно проверяемого утверждения целостности, обычно откатывается.

Ограничения с отложенной проверкой проверяются в момент фиксации транзакции оператором. Внутри транзакции ограничение может и не выполняться. Если в момент фиксации транзакции обнаруживается нарушение ограничения с отложенной проверкой, то транзакция откатывается. Обычно ограничения этого типа не могут быть проверены на уровне средств ядра СУБД, т. к. они чаще всего являются достаточно сложными и для их реализации нужен дополнительный программный код, который может быть размещен внутри явной транзакции. При написании этого кода следует помнить, что режим проверки отложенных ограничений можно в любой момент изменить так, чтобы ограничение проверялось:

- после исполнения *каждого оператора*, изменяющего содержимое таблицы, к которой относится данное ограничение;
- при завершении *каждой транзакции*, включающей операторы, изменяющие содержимое таблиц, к которым относится данное ограничение;
- *в любой промежуточный момент*, если пользователь инициирует проверку.

Ограничения целостности по области действия

Ограничения целостности домена представляют собой ограничения, накладываемые на допустимые значения домена, и на уровне логической модели базы данных обязаны являться частью определения домена. Это немедленно проверяемое ограничение. Наиболее распространенным средством реализации этого ограничения является возможность создания пользовательского типа данных.

Ограничения целостности атрибута представляют собой ограничения, накладываемые на допустимые значения этого атрибута. Ограничения атрибута в точности совпадают с ограничениями домена (типа данных), на котором этот атрибут основан. Очень часто логика предметной области такова, что на значения атрибута необходимо наложить помимо ограничений домена дополнительные ограничения. Ограничение атрибута является *немедленно проверяемым* ограничением, потому что оно не зависит ни от каких других объектов базы данных, кроме домена, на котором основан атрибут, следовательно, никакие изменения в других объектах не могут повлиять на истинность этого ограничения.

Замечание

Ограничения целостности домена и атрибута на физическом уровне реализуются понятием типа данных СУБД. Хотя любая современная СУБД предостав-

ляет достаточно широкий и гибкий перечень стандартных типов данных, все же рекомендуется сначала создать пользовательские типы данных, и уже на них строить описание столбцов базовых таблиц. Во-первых, такой подход увеличивает наглядность модели базы данных. Во-вторых, упрощается ее сопровождение. Например, пусть имеется множество столбцов в разных таблицах, основанных на одном и том же домене, и в их определении к стандартным ограничениям типа данных нужно добавить дополнительную проверку. Если эту проверку нужно изменить, то в ситуации, когда нет пользовательского домена, оператор `ALTER TABLE` придется выполнять для всех таблиц, в которых имеется интересующий нас столбец. Если же предварительно создан пользовательский тип данных, то достаточно произвести необходимые изменения только для него.

Ограничения целостности кортежа представляют собой ограничения, накладываемые на допустимые значения *отдельного* кортежа отношения и *не являющиеся* ограничением целостности атрибута. Это означает, что для проверки данного типа ограничения недостаточно данных только о конкретном значении одного столбца, но и *не требуется* никакой информации о других кортежах отношения. К моменту проверки ограничения кортежа должны быть проверены ограничения целостности атрибутов, входящих в этот кортеж. Ограничение кортежа является *немедленно проверяемым* ограничением, т. к. оно не зависит ни от каких других объектов базы данных, кроме атрибутов, входящих в состав кортежа.

Ограничения целостности отношения представляют ограничения, накладываемые только на допустимые значения *отдельного* отношения и *не являющиеся* ограничением целостности кортежа. Требование того, что ограничение относится к отдельному отношению, означает, что для его проверки не нужна информация о других отношениях (в том числе не требуется ссылок *по внешнему ключу* на кортежи *этого же* отношения). Ограничение отношения может быть как *немедленно проверяемым* ограничением, так и ограничением *с отложенной проверкой*. Ограничениями отношения, например, являются следующие немедленно проверяемые ограничения:

- ограничение целостности сущности, задаваемое потенциальным ключом отношения, для проверки которого необходимо иметь информацию обо всех кортежах отношения, или более точно, обо всех занятых в данный момент значениях потенциального ключа;
- ограничение целостности, определяемое наличием функциональных или многозначных зависимостей.

В качестве примера ограничения отношения с отложенной проверкой можно привести ограничение, основанное на наличии зависимостей соединения *(см. разд. 6.3)*. Это ограничение требует, чтобы кортежи вставлялись и удалялись *целыми группами,* что невозможно сделать, если выполнять проверку после каждой одиночной вставки или удаления кортежа.

Проверка ограничения отношения выполняется после проверки ограничения целостности кортежей этого отношения.

Ограничения целостности базы данных представляют ограничения, накладываемые на значения двух или более связанных между собой отношений, или в том случае, когда отношение связано само с собой. Наиболее часто здесь используется ограничение целостности ссылок, задаваемое внешним ключом отношения. К моменту проверки ограничения базы данных должны быть проверены ограничения целостности отношений. Ограничение базы данных может быть как *немедленно проверяемым* ограничением, так и ограничением *с отложенной проверкой*. Ограничение внешнего ключа может быть как немедленно проверяемым, так и отложенным ограничением.

Ограничения целостности по классификации Microsoft для SQL-сервера

Данная классификация отличается простотой, хотя и не сильно детализирована. Так, из всего многообразия ограничений в ней выделено всего четыре их типа, первые три из которых, такие как *целостность доменов*, *целостность сущностей* и *ссылочная целостность*, совпадают со стандартными ограничениями реляционной модели данных *(см. главу 1)*, а все остальные мыслимые типы ограничений считаются *ограничениями*, *задаваемыми пользователем*. Разумеется, первые три категории ограничений также задать может только пользователь, поэтому понятие "ограничения, задаваемые пользователем" — не более чем термин. Нельзя даже сказать, что последняя категория ограничений проверяется как-то иначе или по-другому программируется. Ограничения первых трех категорий реализуются на уровне ядра СУБД, а записываются либо с помощью DDL операторов, либо с помощью специальных системных хранимых процедур, необходимых для создания пользовательских типов данных. Ограничения четвертой категории чаще всего реализуются с помощью триггеров или в пользовательских хранимых процедурах. Но к последнему типу ограничений, например, относятся также *ограничения уровня кортежа*, которые задаются средствами DDL оператора `ALTER TABLE` и реализуются на уровне ядра СУБД.

Уточним средства реализации того или иного типа ограничений на SQL-сервере.

- Целостность доменов означает наличие ограничений на значения, которые допустимы для конкретного столбца; при этом широко используются пользовательские типы данных. К базовому средству реализации ограничения, основанному на том, что любой столбец имеет определенный тип данных, добавляются такие опции операторов DDL, как `CHECK`, `DEFAULT` и `NOT NULL`, накладываемые на отдельный столбец таблицы.
- Целостность сущностей требует наличия уникального идентификатора для каждой строки отношения, что реализуется либо с помощью индексов, об-

ладающих ограничениями `UNIQUE` или `PRIMARY KEY`, либо заданием для столбца свойства `IDENTITY`, которое не дает все же стопроцентной гарантии уникальности, или типа данных `uniqueidentifier`.

- Ссылочная целостность устанавливает ограничение внешнего ключа между парой таблиц и реализуется ограничением `FOREIGN KEY` операторов DDL. Ограничение ссылочной целостности SQL-сервера включает в себя два *действия*, *исполняемых по ссылке*, а именно *каскадное удаление* и *обновление*.
- Целостность, определяемая пользователем, чаще всего реализуется триггерами или пользовательскими хранимыми процедурами.

В заключение приведем текст программы, которая содержит операторы реализации большинства стандартных ограничений базы данных на примере двух таблиц, связанных ограничением внешнего ключа (листинг П2.1). Комментариями поясняются операторы, отличные от стандарта SQL.

Листинг П2.1. Задание ограничений целостности базы данных для SQL-сервера

```
-- Создание пользовательского значения по умолчанию
create default [sex] as 'муж'
GO
-- Создание пользовательского правила проверки по умолчанию
create rule [s] as @sex in ('муж','жен')
GO
-- Создание пользовательского типа данных
EXEC sp_addtype N'sex', N'char (3)', N'null'
GO
-- Связывание пользовательского правила с типом данных
EXEC sp_bindrule N'[dbo].[s]', N'[sex]'
GO
-- Связывание пользовательского значения по умолчанию и типа данных
EXEC sp_bindefault N'[dbo].[i]', N'[sex]'
GO
-- Создание базовых таблиц
CREATE TABLE [dbo].[Table1] (
-- Задание поля со свойством счетчика
       [Stud_Id] [int] IDENTITY (1, 1) NOT NULL,
       [FIO] [varchar] (50) COLLATE Cyrillic_General_CI_AS NULL,
       [Birthday] [smalldatetime] NULL,
-- Создание поля на основе пользовательского типа данных
       [Пол] [sex] NULL,
       [Group_Id] [int] NULL,
) ON [PRIMARY]
```

```
GO
CREATE TABLE [dbo].[Table3] (
       [Group_Id] [int] NOT NULL,
       [Kol_Stud] [int] NULL,
) ON [PRIMARY]
GO
ALTER TABLE [dbo].[Table1] WITH NOCHECK ADD
-- Задание ограничения первичного ключа
       CONSTRAINT [PK_Table1] PRIMARY KEY CLUSTERED
       ([Stud_Id]
) ON [PRIMARY]
GO
ALTER TABLE [dbo].[Table3] WITH NOCHECK ADD
       CONSTRAINT [PK_Table3] PRIMARY KEY CLUSTERED
       ( [Group_Id]
) ON [PRIMARY]
GO
-- Добавление ограничения внешнего ключа с опциями каскадирования
ALTER TABLE [dbo].[Table1] ADD
       CONSTRAINT [FK_Table1_Table3] FOREIGN KEY  ([Group_Id])
       REFERENCES [dbo].[Table3] ([Group_Id])
       ON DELETE CASCADE
       ON UPDATE CASCADE
GO
-- Добавление в таблицы ограничений целостности, основанных на
-- значениях по умолчанию
ALTER TABLE [dbo].[Table1] WITH NOCHECK ADD
CONSTRAINT [DF_Table1_Birthday] DEFAULT ('01-Jan-1980') FOR [Birthday]
GO
ALTER TABLE [dbo].[Table3] WITH NOCHECK ADD
CONSTRAINT [DF_Table3_Kol_Stud] DEFAULT (0) FOR [Kol_Stud],
CONSTRAINT [DF_Table3_Nag_group] DEFAULT (0) FOR [Nag_group],
-- Добавление проверяемого ограничения на столбец
CONSTRAINT [CK_Table3] CHECK ([Kol_Stud] > 10)
GO
-- Создание триггера
CREATE TRIGGER TRIG1 ON dbo.Table1
FOR INSERT
AS
update table3 set kol_stud = kol_stud
+(select count(*) from inserted where Group_Id=t3.Group_Id)
from table3 t3,inserted i where t3.Group_id=i.Group_id
```

Список литературы

1. Атре Ш. Структурный подход к организации баз данных. — М.: Финансы и статистика, 1983.
2. Бойко В. В., Савинков В. М. Проектирование баз данных информационных систем. — М.: Финансы и статистика, 1989.
3. Боуман Д., Эмерсон С., Дарновски М. Практическое руководство по SQL. — К.: Диалектика, 1997.
4. Васкевич Д. Стратегии клиент/сервер. — К.: Диалектика, 1997.
5. Гарбус Дж., Паскуази Д., Чанг Э. Database Design on SQL Server 7. Сертификационный экзамен — экстерном. — СПб.: Питер, 2000.
6. Гилуа М. М. Множественная модель данных в информационных системах. — М.: Наука, 1992.
7. Грабер М. Введение в SQL. — М.: Лори, 1996.
8. Грабер М. Справочное руководство по SQL. — М.: Лори, 1997.
9. Дейт К. Введение в системы баз данных. 6-е изд. — К.: Диалектика, 1998.
10. Джексон Г. Проектирование реляционных баз данных для использования с микроЭВМ. — М.: Мир, 1991.
11. Диго С. М. Проектирование и использование баз данных. — М.: Финансы и статистика, 1995.
12. Дунаев С. Б. Доступ к базам данных и техника работы в сети. — М.: Диалог МИФИ, 2000.
13. Конноли Т., Бегг К., Страчан А. Базы данных: проектирование, реализация и сопровождение. Теория и практика. — М.: Издательский дом "Вильямс", 2000.

14. Маклаков С. В. Bpwin и Erwin. CASE-средства разработки информационных систем. — М.: Диалог МИФИ, 2000.
15. Маклаков С. В. Моделирование бизнес-процессов с AllFusion. — М.: Диалог МИФИ, 2000.
16. Малыхина М. П. Базы данных: основы, проектирование, использование. — СПб.: БХВ-Петербург, 2004.
17. Мамаев Е., Шкарина Л. Microsoft SQL Server 2000 — М.: Питер, 2001.
18. Мамаев Е. Microsoft SQL Server 2000. Наиболее полное руководство. — СПб.: БХВ-Петербург, 2004.
19. Мейер М. Теория реляционных баз данных. — М.: Мир, 1987.
20. Нагао М., Катаяма Т., Уэмура С. Структуры и базы данных. — М.: Мир, 1986.
21. Озкарахан Э. Машины баз данных и управление базами данных. — М.: Мир, 1989.
22. Хансен Г., Хансен Дж. Базы данных, разработка и управление. — М.: Бином, 2000.
23. Шнитман В. З., Кузнецов С. Д. Информационно-аналитические материалы Центра Информационных Технологий. **http://citforum.univ.kiev.ua /database/skbd**.

Предметный указатель

B

B-дерево 181
◊ индексное 181

C

CASE-средства 27

N

NULL-значение 18

O

OLAP-система 117, 189, 199
OLTP-приложение 116
OLTP-система 189, 199

S

S-блокировка 322

X

X-блокировка 322

А

Автогенерация данных 270
Автомодификация статистики 221
Аномалии модификации данных 131, 137
Архитектура ANSI-SPARC 118
Ассоциативная связь 55
Ассоциативная сущность 54
Атрибут сущности 43
Атрибутивная сущность 45

Б

Базовый атрибут 124
Бизнес-правило 30
Бинарная связь 46
Блокировка 322
◊ монопольная 322
◊ намерения 325
◊ разделяемая 322
Буферный кэш 175, 209, 255, 311

В

Вертикальная фильтрация 209
Вложенные транзакции 356
Внешний ключ 19
Внутреннее представление 287, 294
Восстановление базы данных 329
Время выполнения запроса 216
Вставка картинки из файла 280
Выделение:
◊ версий 322
◊ экстентов 174

Выравнивание модели 90
Выталкивание страниц во внешнюю память 328
Вычислимый атрибут 124, 125

Г

Гиперкуб 117
Горизонтальная фильтрация 209
Граф ожидания транзакций 327
График запуска 316
Группа файлов 162
◊ PRIMARY 162
Группировка 236
Грязная страница 175, 328

Д

Декомпозиция 140
◊ без потерь 145
◊ запроса 209
◊ правило цепочки 142
Дерево запроса 212
Детерминант 132
Дефрагментация данных 199
Диаграмма:
◊ ER-экземпляров 38, 56
◊ бизнес-процессов 29
◊ вариантов использования 30
◊ потоков данных 29
◊ сущность-связь 28, 32
Динамический SQL 259, 261, 301
Дискриминатор 67, 89
Долговечность транзакций 327
Домен 14, 16, 96
Дочернее отношение 19
Дочерняя сущность 50

Е

Естественное соединение 130

Ж

Журнал транзакций 327

З

Зависимость соединения 140

И

Идентификатор строки 186
Идентифицирующая связь 39, 50
Иерархическая структура данных 179
Иерархия наследования 88
Избыточность 124, 125
Инверсионный вход 96
Индексированное представление 245, 295, 311
Индексный поиск 237
Инжиниринг 28
Интеграция внешнего ключа 40
Информационное проектирование 26
Исключающая связь 93

К

Кардинальность 219
Карта размещения индексов 171
Класс принадлежности:
◊ необязательный 56
◊ обязательный 52
Класс принадлежности (модальность) 35, 47
Кластерная таблица 169
Кластерный индекс 169, 183, 224
Клиентский курсор 368
Ключ:
◊ кластеризации 184, 186
◊ отношения 17
  ▫ альтернативный 17, 96
  ▫ атомарный 17
  ▫ возможный 17
  ▫ первичный 17
  ▫ потенциальный 17
  ▫ составной 17
◊ сущности 42
Компенсирующие действия 315
Конкурирующие транзакции 317
Конфликт доступа 317
Корень дерева 180
Коррелированный подзапрос 293
Курсор 273, 299
◊ динамический 299
◊ клиентский 299
◊ ключевой 299
◊ серверный 299
◊ статический 299
Кэширование данных 175, 256

Л

Лист дерева 180
Логическое чтение 175

М

Массовое копирование 272
Мастер 295
Мастер Настройки Индексов 241
Миграция ключевых атрибутов 39
Минимальное покрытие 135, 150
Многозначная зависимость, нетривиальная 138
Многомерная связь 62
Моделирование времени 71
Модель данных 31, 118
◊ диаграмма 119
◊ концептуальная 32, 118
◊ логическая 32, 118, 119
◊ физическая 118, 120
Мониторинг транзакций 350
Мощность связи (степень связи) 35, 46

Н

Недетерминированная функция 239, 269
Неидентифицирующая связь 39, 50
Некластерный индекс 184, 226
Неповторяемое считывание 319
Несовместимый анализ 321
Неявный режим работы транзакций 340
Нормализация 129
Нормальная форма 131
◊ Бойса — Кодда 134
◊ вторая 133
◊ первая 131
◊ пятая 140
◊ третья 134
◊ четвертая 136
Нотация модели:
◊ IDEF1X-нотация 34
◊ IE-нотация 34
◊ нотация Баркера 35
◊ нотация Чена 38

О

Объем базы данных 165
Оператор ПРОЕКЦИЯ 140
Операторы управления транзакциями 340
Оптимизатор запросов 210, 246
Оптимизация запросов:
◊ стоимостная 210
◊ эвристическая 210
Откат транзакции 341
Отношение 14
◊ много-ко-многим 47, 83, 88
◊ один-к-одному 20, 47, 83
◊ один-ко-многим 20, 47, 83
Отношение связи 55

П

Панель инструментов ERwin 75
Первичный ключ 69
Период ожидания 349
План запроса 212
◊ выполненный 214
◊ оценочный 214
Повторяющаяся группа атрибутов 131, 147
Подсказка оптимизатору (хинт) 248, 249, 344, 349
◊ запроса 253
◊ таблицы 251
Поиск диапазона 186
Покрывающий индекс 228
Потеря результатов обновления 318
Появление фиктивных элементов (фантомов) 320
Правило вывода ФЗ 135
Предварительные отношения 51
◊ правила генерации 51
Предикат отношения 15
Предикатная блокировка 325
Предметная область 29, 118
◊ модель 118
Представление 310
Проектное решение 33, 48
Промежуточная точка сохранения 357
Протокол доступа к данным 322
Процесс контрольной точки 328

Р

Размер:
◊ индекса 166
◊ таблицы 166
◊ таблицы кучи 172

Расщепление страниц данных 170, 193
Реакция системы на попытку нарушения целостности 315
Редактор:
◊ атрибутов Erwin 80
◊ связей Erwin 83
◊ сущности Erwin 79
Реинжиниринг 28
Рекурсивная связь 86, 94, 98
Реляционная модель данных 13, 16
◊ определение основных терминов 14
Родительская сущность 50
Родительское отношение 19
Ролевая связь 65
Ролевое имя атрибута 85
Ролевые имена 97

С

Свойства АСИД транзакций 313
Связь 45
◊ без права передачи 70
◊ супертип-подтип 67, 89
  ▫ неполная 67
  ▫ полная 67
◊ триплет 62
Селективность 226, 237
Семантика данных 16, 18, 28
Семантическое (инфологическое) моделирование 28, 34
Сериализуемость 316
Сжатие файлов базы данных 167
Системная база данных 165
Системная таблица 165
◊ syscomments 255
◊ sysindexes 171, 195, 220, 251
◊ syslockinfo 323, 350
◊ sysobjects 196, 269
Системная хранимая процедура:
◊ sp_configure 176
◊ sp_dboption 221
◊ sp_executesql 261
◊ sp_executesql 258
◊ sp_helpindex 197
◊ sp_lock 350
◊ sp_recompile 257
◊ sp_statistics 197
◊ sp_tableoption 176
◊ sp_updatestats 221
◊ sp_who 350
Системные ресурсы 161
Скалярный подзапрос 285
Сканирование таблицы 179, 190, 209
Словарь данных (каталог данных) 31
Смесь транзакций 316
Соединение таблиц 244
◊ HASH JOIN 247
◊ LOOP JOIN 246
◊ MERGE JOIN 246
◊ естественное соединение 244
◊ левое внешнее соединение 291
◊ тета-соединение 244
◊ эквисоединение 244
Создание:
◊ базы данных 164
◊ индексов 191
Составной индекс 228
Среда соединения 256
Ссылочная целостность 20
Ссылочные действия 87
Стоимость запроса 210
◊ оценочная 211
Страница данных 169
Суррогатный первичный ключ 70
Сущность 17, 30, 41
◊ сильная (независимая) 39
◊ слабая (зависимая) 39
Сущность-связь 28

Т

Таблица кучи 170
Технология:
◊ клиент-сервер 22
◊ файл-сервер 23
Транзакция 313
◊ Атомарность 313
◊ Долговечность 314
◊ Изолированность 313
◊ Согласованность 313, 314
Транзакция-жертва 326, 349
Триггер 123, 305, 361
Тупик 326

У

Уровень изоляции транзакций 343
Утилита:
◊ Enterprise Manager 281
◊ Performance Monitor 193

◊ Performance Monitor 177
◊ Query Analyzer 212, 217
◊ Query Analyzer 230
◊ SQL Profiler 218, 222, 224, 250, 262
◊ Sqlmaint.exe 199
◊ Мастер Настройки Индексов 230

Ф

Файл:
◊ базы данных 161
  ▫ вторичный 162
  ▫ первичный 162
◊ журнала транзакций 162
Фактор заполнения индекса 192, 245
Физическая модель данных 32
Физическое чтение 175
Фиксация транзакции 341
Фрагментация данных 198
Функциональная зависимость 131, 132
◊ диаграмма 149, 154
Функциональное имя атрибута 85

Х

Хранимая процедура 123, 254
◊ владелец 256
◊ кэширование плана выполнения 259
◊ план выполнения 259
Хеширование 237

Ц

Целевая СУБД 159
Целевая таблица 270
Целостность внешних ключей 20
Целостность сущностей 19

Ч

Чтение грязных данных 318

Э

Экземпляр сущности 41, 42
Экстент 169
Эскалация блокировок 324

Я

Явный режим работы транзакций 340