Основы информационных технологий

С. Д. Кузнецов

ОСНОВЫ БАЗ ДАННЫХ

Учебное пособие

2-е издание, исправленное

Интернет-Университет
Информационных Технологий
www.intuit.ru

БИНОМ.
Лаборатория знаний
www.lbz.ru

Москва
2007

УДК 004.655.3(075.8)
ББК 32.973.26-018.2
К89

Кузнецов С. Д.

К89 Основы баз данных : учебное пособие / С.Д. Кузнецов – 2-е изд., испр. –М.: Интернет-Университет Информационных Технологий; БИНОМ. Лаборатория знаний, 2007. – 484 с. : ил. – (Серия «Основы информационных технологий»).

ISBN 978-5-94774-736-2 (БИНОМ. ЛЗ)

На современном уровне определяется реляционная модель данных, включая реляционную алгебру и реляционное исчисление. Обсуждается классический подход к проектированию реляционных баз данных на основе принципов нормализации. Описываются основные черты походов к семантическому моделированию баз данных. Рассматривается модель данных SQL.

Книга рассчитана на студентов, специализирующихся в области технологий баз данных, специалистов, которые интересуются проблемами проектирования и моделирования баз данных, а также на широкий круг читателей, которые хотели бы с этим разобраться.

УДК 004.655.3(075.8)
ББК 32.973.26-018.2

Издание осуществлено при финансовой и технической поддержке издательства «Открытые Системы», «РМ Телеком» и Kraftway Computers.

По вопросам приобретения обращаться:
«БИНОМ. Лаборатория знаний»
Телефон (499) 157-1902, (499) 157-5272,
e-mail: Lbz@aha.ru, http://www.Lbz.ru

ISBN 978-5-94774-736-2 (БИНОМ. ЛЗ)

О проекте

Интернет-Университет Информационных Технологий – это первое в России высшее учебное заведение, которое предоставляет возможность получить дополнительное образование во Всемирной сети. Web-сайт университета находится по адресу www.intuit.ru.

Мы рады, что вы решили расширить свои знания в области компьютерных технологий. Современный мир – это мир компьютеров и информации. Компьютерная индустрия – самый быстрорастущий сектор экономики, и ее рост будет продолжаться еще долгое время. Во времена жесткой конкуренции от уровня развития информационных технологий, достижений научной мысли и перспективных инженерных решений зависит успех не только отдельных людей и компаний, но и целых стран. Вы выбрали самое подходящее время для изучения компьютерных дисциплин. Профессионалы в области информационных технологий сейчас востребованы везде: в науке, экономике, образовании, медицине и других областях, в государственных и частных компаниях, в России и за рубежом. Анализ данных, прогнозы, организация связи, создание программного обеспечения, построение моделей процессов – вот далеко не полный список областей применения знаний для компьютерных специалистов.

Обучение в университете ведется по собственным учебным планам, разработанным ведущими российскими специалистами на основе международных образовательных стандартов Computer Curricula 2001 Computer Science. Изучать учебные курсы можно самостоятельно по учебникам или на сайте Интернет-Университета, задания выполняются только на сайте. Для обучения необходимо зарегистрироваться на сайте университета. Удостоверение об окончании учебного курса или специальности выдается при условии выполнения всех заданий к лекциям и успешной сдачи итогового экзамена.

Книга, которую вы держите в руках, – очередная в многотомной серии «Основы информационных технологий», выпускаемой Интернет-Университетом Информационных Технологий. В этой серии будут выпущены учебники по всем базовым областям знаний, связанным с компьютерными дисциплинами.

Добро пожаловать
в Интернет-Университет Информационных Технологий!

Анатолий Шкред
anatoli@shkred.ru

Предисловие

Этот курс является одним из результатов более чем десятилетнего чтения курса по тематике баз данных на факультете вычислительной математики и кибернетики МГУ им. М.В. Ломоносова. Будучи профессиональным программистом, десять лет назад я старался по возможности сокращать изложение теоретических аспектов баз данных, уделяя большее внимание алгоритмическим вопросам организации систем управления базами данных. С годами мне стала понятна дефектность такого подхода, присущая, кстати, большинству «универсальных» учебных книг, посвященных базам данных. Стремление покрыть материалом одного курса или одной книги модельно-теоретические аспекты организации баз данных, алгоритмы и структуры данных, используемые в СУБД, а также методы разработки приложений приводит к тому, что ни одну из этих тем не удается изложить последовательно и целостно.

Данный курс посвящен модельному компоненту баз данных. Я считаю, что изучение технологии баз данных в целом лучше всего начинать именно с этой темы, формируя некоторую платформу для освоения более технических вопросов. Мы ограничиваемся двумя наиболее распространенными, тесно связанными моделями: реляционной моделью данных и моделью данных языка SQL. Изложение ведется на современном уровне — реляционная модель данных описывается под влиянием ее интерпретации Кристофером Дейтом и Хью Дарвеном в их последних книгах, а модель данных SQL основывается на спецификациях стандартов SQL:1999 и SQL:2003.

Особняком в курсе стоят лекции 9-10 и 19. В лекциях 9-10 мы отклоняемся в область семантического моделирования баз данных. Конечно, в двух лекциях можно привести только беглый набросок этой области. Но я счел уместным хотя бы поверхностно познакомить читателей с особенностями использования диаграмм «сущность-связь» и диаграмм классов языка UML для проектирования SQL-ориентированных баз данных. В лекции 19 говорится об объектно-реляционных расширениях языка SQL. Эта лекция несколько перегружена материалом, и может оказаться труднее для полного усвоения. Но это направление развития SQL кажется мне настолько интересным, что я не смог отказаться от включения в этот курс соответствующего материала.

Об авторе

Кузнецов Сергей Дмитриевич

Доктор технических наук, профессор факультета ВМиК МГУ, профессор ФУПМ МФТИ, главный научный сотрудник Института системного программирования, член экспертных советов РФФИ и РГНФ (гуманитарных исследований), эксперт Центра информационных технологий, научный редактор журнала «Открытые системы», член ACM и ACM SIGMOD и IEEE Computer Society, представитель IEEE Computer Society в Москве, заместитель председателя Московской секции ACM SIGMOD.

Лекции

Лекция 1. Эволюция устройств внешней памяти и программных систем управления данными 15
Лекция 2. Введение в реляционную модель данных 37
Лекция 3. Базисные средства манипулирования реляционными данными: реляционная алгебра Кодда 54
Лекция 4. Базисные средства манипулирования реляционными данными: алгебра А Дейта и Дарвена 73
Лекция 5. Базисные средства манипулирования реляционными данными: реляционное исчисление 97
Лекция 6. Элементы теории реляционных баз данных: функциональные зависимости и декомпозиция без потерь 110
Лекция 7. Проектирование реляционных баз данных на основе принципов нормализации: первые шаги нормализации . . 124
Лекция 8. Проектирование реляционных баз данных на основе принципов нормализации: дальнейшая нормализация . . 141
Лекция 9. Проектирование реляционных баз данных с использованием семантических моделей: ER-диаграммы 155
Лекция 10. Проектирование реляционных баз данных с использованием семантических моделей: диаграммы классов языка UML 182
Лекция 11. Язык баз данных SQL: общее введение, типы данных и средства определения доменов 204
Лекция 12. Язык баз данных SQL: средства определения базовых таблиц и ограничений целостности 234
Лекция 13. Язык баз данных SQL: общая характеристика оператора SELECT и организация списка ссылок на таблицы в разделе FROM 263
Лекция 14. Язык баз данных SQL: предикаты раздела WHERE оператора SELECT 286
Лекция 15. Язык баз данных SQL: группировка и условия раздела HAVING, порождаемые и соединенные таблицы 317
Лекция 16. Язык баз данных SQL: средства формулировки аналитических и рекурсивных запросов 352
Лекция 17. Язык баз данных SQL: средства манипулирования данными . . 372
Лекция 18. Язык баз данных SQL: средства языка SQL для обеспечения авторизации доступа к данным, управления транзакциями, сессиями и подключениями . . 411
Лекция 19. Язык баз данных SQL: объектные расширения 447

Содержание

Лекция 1. Эволюция устройств внешней памяти и программных систем управления данными 15
Устройства внешней памяти 15
Файловые системы 19
Структуры файлов 20
Логическая структура файловых систем и именование файлов . 21
Авторизация доступа к файлам 23
Синхронизация многопользовательского доступа 24
Области разумного применения файлов 25
Потребности информационных систем 26
Структуры данных 28
Целостность данных 28
Языки запросов 30
Транзакции, журнализация и многопользовательский режим . . 31
СУБД как независимый системный компонент 33
Лекция 2. Введение в реляционную модель данных. 34
Основные понятия реляционных баз данных 37
Тип данных 38
Домен 39
Заголовок отношения, кортеж, тело отношения, значение отношения, переменная отношения 40
Первичный ключ и интуитивная интерпретация реляционных понятий 41
Фундаментальные свойства отношений 42
Отсутствие кортежей-дубликатов, первичный и возможные ключи отношений 42
Отсутствие упорядоченности кортежей 44
Отсутствие упорядоченности атрибутов 45
Атомарность значений атрибутов, первая нормальная форма отношения 46
Реляционная модель данных 48
Общая характеристика 48
Целостность сущности и ссылок 49
Лекция 3. Базисные средства манипулирования реляционными данными: реляционная алгебра Кодда 54
Обзор реляционной алгебры Кодда 56
Общая интерпретация реляционных операций 56
Замкнутость реляционной алгебры и операция переименования . . 58
Особенности теоретико-множественных операций реляционной алгебры 59
Операции объединения, пересечения, взятия разности. Совмесимость по объединению 59

Операция расширенного декартова произведения и совместимость отношений относительно этой операции 63
Специальные реляционные операции 65
Операция ограничения 65
Операция взятия проекции 66
Операция соединения отношений 67
Операция деления отношений 71
Лекция 4. Базисные средства манипулирования реляционными данными: Алгебра А Дейта и Дарвена 73
Базовые операции Алгебры А 74
Операция реляционного дополнения 75
Операция удаления атрибута 75
Операция переименования 76
Операция реляционной конъюнкции 77
Операция реляционной дизъюнкции 78
Полнота Алгебры А 82
Выводимость операции взятия разности 83
Интерпретация операции ограничения 85
Соединения общего вида 90
Реляционное деление 90
Избыточность Алгебры А 92
Реляционные аналоги штриха Шеффера и стрелки Пирса 92
Избыточность операции переименования 95
Лекция 5. Базисные средства манипулирования реляционными данными: реляционное исчисление 97
Исчисление кортежей 99
Правильно построенные формулы 99
Целевые списки и выражения реляционного исчисления 106
Исчисление доменов 107
Условия членства 107
Выражения исчисления доменов 108
Лекция 6. Элементы теории реляционных баз данных: функциональные зависимости и декомпозиция без потерь 110
Функциональные зависимости 111
Общие определения 111
Замыкание множества функциональных зависимостей. Аксиомы Армстронга. Замыкание множества атрибутов . . . 113
Минимальное покрытие множества функциональных зависимостей 116
Декомпозиция без потерь и функциональные зависимости 118
Корректные и некорректные декомпозиции отношений. Теорема Хита 119
Диаграммы функциональных зависимостей 122
Лекция 7. Проектирование реляционных баз данных на основе принципов нормализации: первые шаги нормализации. . . 124

Минимальные функциональные зависимости и вторая нормальная форма . . . 126
Аномалии обновления по причине наличия неминимальных функциональных зависимостей . . . 127
Возможная декомпозиция . . . 128
Вторая нормальная форма . . . 129
Нетранзитивные функциональные зависимости и третья нормальная форма . . . 130
Аномалии обновлений по причине наличия транзитивных функциональных зависимостей . . . 130
Возможная декомпозиция . . . 131
Третья нормальная форма . . . 131
Независимые проекции отношений. Теорема Риссанена . . . 133
Перекрывающиеся возможные ключи и нормальная форма Бойса-Кодда . . . 134
Аномалии обновлений, связанные с наличием перекрывающихся возможных ключей . . . 134
Нормальная форма Бойса-Кодда . . . 136
Всегда ли следует стремиться к BCNF? . . . 136
Лекция 8. Проектирование реляционных баз данных на основе принципов нормализации: дальнейшая нормализация . . . 141
Многозначные зависимости и четвертая нормальная форма . . . 142
Аномалии обновлений при наличии многозначных зависимостей и возможная декомпозиция . . . 143
Многозначные зависимости. Теорема Фейджина. Четвертая нормальная форма . . . 144
Зависимости проекции/соединения и пятая нормальная форма . . 147
N-декомпозируемые отношения . . . 147
Зависимость проекции/соединения . . . 148
Аномалии, вызываемые наличием зависимости проекции/соединения . . . 150
Устранение аномалий обновления в 3-декомпозиции . . . 151
Пятая нормальная форма . . . 151
Лекция 9. Проектирование реляционных баз данных с использованием семантических моделей: ER-диаграммы . 155
Ограниченность реляционной модели при проектировании баз данных . . . 156
Семантические модели данных . . . 156
Семантическая модель Entity-Relationship (Сущность-Связь) . . 160
Основные понятия ER-модели . . . 161
Уникальные идентификаторы типов сущности . . . 164
Нормальные формы ER-диаграмм . . . *167*
Первая нормальная форма ER-диаграммы . . . 168
Вторая нормальная форма ER-диаграммы . . . 169
Третья нормальная форма ER-диаграммы . . . 171

Более сложные элементы ER-модели . . . 171
Наследование типов сущности и типов связи . . . 173
Взаимно исключающие связи . . . 175
Получение реляционной схемы из ER-диаграммы . . . 176
Базовые приемы . . . 176
Представление в реляционной схеме супертипов и подтипов сущности . . . 177
Представление в реляционной схеме взаимно исключающих связей . . . 179
Лекция 10. Проектирование реляционных баз данных с использованием семантических моделей: диаграммы классов языка UML . . . 182
Основные понятия диаграмм классов UML . . . 183
Классы, атрибуты, операции . . . 183
Категории связей. Связь-зависимость . . . 185
Связи-обобщения и механизм наследования классов в UML . . . 186
Связи-ассоциации: роли, кратность, агрегация . . . 189
Ограничения целостности и язык OCL . . . 193
Общая харакеристика языка OCL . . . 194
Инвариант класса . . . 195
Операции над множествами, мультимножествами и последовательностями . . . 196
Примеры инвариантов . . . 198
Плюсы и минусы использования языка OCL при проектировании реляционных баз данных . . . 201
Получение схемы реляционной базы данных из диаграммы классов UML . . . 201
Лекция 11. Язык баз данных SQL: общее введение, типы данных и средства определения доменов . . . 204
Краткая история языка SQL . . . 205
Структура языка SQL . . . 210
Типы данных SQL . . . *212*
Точные числовые типы . . . 213
Приближенные числовые типы . . . 214
Типы символьных строк . . . 215
Типы битовых строк . . . 216
Типы даты и времени . . . 218
Типы временных интервалов . . . 218
Булевский тип . . . 221
Типы коллекций . . . 221
Анонимные строчные типы . . . 224
Типы, определяемые пользователем . . . 224
Ссылочные типы . . . 225
Средства определения, изменения определения и отмены определения доменов . . . 225

Определение домена 226
Примеры определений доменов 227
Изменение определения домена 228
Примеры изменения определения домена 229
Отмена определения домена 230
Неявные и явные преобразования типа или домена. 230
Неявные преобразования типов в SQL 231
Явные преобразования типов или доменов и оператор CAST. . 231
Лекция 12. Язык баз данных SQL: средства определения базовых таблиц и ограничений целостности 234
Средства определения, изменения и ликвидации базовых таблиц . 236
Определение базовой таблицы 236
Определение табличного ограничения 239
Табличное ограничение внешнего ключа 240
Примеры определений базовых таблиц 244
Изменение определения базовой таблицы 249
Отмена определения (уничтожение) базовой таблицы 254
Средства определения и отмены общих ограничений целостности. . 254
Определение общих ограничений целостности 255
Отмена определения общего ограничения целостности 259
Немедленная и откладываемая проверка ограничений 260
Лекция 13. Язык баз данных SQL: общая характеристика оператора SELECT и организация списка ссылок на таблицы в разделе FROM 263
Скалярные выражения 264
Общие синтаксические правила построения скалярных выражений 265
Численные выражения 266
Выражения, значениями которых являются символьные или битовые строки 267
Выражения даты-времени 269
Булевские выражения 270
Выражения с переключателем 271
Общая структура оператора выборки в языке SQL 273
Семантика оператора выборки 274
Ссылки на таблицы раздела FROM 278
Представляемые таблицы, или представления (VIEW) 284
Лекция 14. Язык баз данных SQL: предикаты раздела WHERE оператора SELECT. 286
Логические выражения раздела WHERE 287
Предикат сравнения 289
Предикат between 294
Предикат null 295
Предикат in 297
Предикат like 298
Предикат similar 300

Предикат exists 304
Предикат unique 306
Предикат overlaps 307
Предикат сравнения с квантором 308
Предикат match 312
Предикат distinct 315
Лекция 15. Язык баз данных SQL: группировка и условия раздела HAVING, порождаемые и соединенные таблицы 317
Внешние соединения 319
Агрегатные функции, группировка и условия раздела HAVING . . 321
Семантика агрегатных функций 321
Результаты запросов и агрегатные функции 323
Логические выражения раздела HAVING 324
Ссылки на порождаемые таблицы в разделе FROM 335
Еще один способ формулировки запросов 335
Случаи, в которых без порождаемых таблиц обойтись невозможно 336
Более сложные конструкции оператора выборки 338
Соединенные таблицы 338
Порождаемые таблицы с горизонтальной связью (lateral_derived_table) 349
Лекция 16. Язык баз данных SQL: средства формулировки аналитических и рекурсивных запросов 352
Возможности формулирования аналитических запросов 354
Раздел GROUP BY ROLLUP 356
Агрегатная функция GROUPING 358
Раздел GROUP BY CUBE 361
Рекурсивные запросы 363
Определения, относящиеся к рекурсии 363
Рекурсивные запросы с разделом WITH 365
Рекурсивные представления 371
Лекция 17. Язык баз данных SQL: средства манипулирования данными . . 372
Базовые средства манипулирования данными 373
Оператор INSERT для вставки строк в существующие таблицы 374
Оператор UPDATE для модификации существующих строк в существующих таблицах 378
Оператор DELETE для удаления строк в существующих таблицах 379
Представления, над которыми возможны операции обновления . . . 380
Представления, допускающие применение операций обновления в стандарте SQL/92 381
Представления, допускающие применение операций обновления, в стандарте SQL:1999 384
Раздел WITH CHECK OPTION определения представления . . 389

Исторический очерк 396
Операции обновления баз данных и механизм триггеров *397*
Понятие триггера в SQL:1999 398
Синтаксис определения триггеров и типы триггеров 399
Выполнение триггеров 403
Триггеры и ссылочные действия 408
Лекция 18. Язык баз данных SQL: средства языка SQL для обеспечения авторизации доступа к данным, управления транзакциями, сессиями и подключениями . . 411
Поддержка авторизации доступа к данным в языке SQL. 413
Пользователи и роли 415
Использование идентификаторов пользователей и имен ролей . 417
Создание и ликвидация ролей. 418
Передача привилегий и ролей. 419
Изменение текущих идентификаторов пользователей и имен ролей 423
Аннулирование привилегий и ролей 424
Управление транзакциями в SQL 428
ACID-транзакция 428
Порождение транзакций в SQL:1999. 429
Уровни изоляции SQL-транзакции 431
Завершение транзакций 435
Транзакции и ограничения целостности. 437
Точки сохранения 438
Подключения и сессии. 440
Установление соединений 441
Операторы SQL для управления соединениями. 442
Лекция 19. Язык баз данных SQL: объектные расширения 447
Истоки и краткая история объектно-реляционных баз данных. . 449
Объектная модель SQL 453
Цели лекции 455
Определяемые пользователями типы 456
Индивидуальные типы. 456
Определение структурных типов 459
Типизированные таблицы 466
Определение типизированной таблицы 466
Ссылочные значения и REF-типы 466
Выборка данных из типизированных таблиц 474
Типизированные представления 477
Литература 480

Внимание!

На сайте Интернет-университета информационных технологий Вы можете пройти тестирование по каждой лекции и курсу в целом.

Добро пожаловать на наш сайт:

www.intuit.ru

Лекция 1. Эволюция устройств внешней памяти и программных систем управления данными

В этой вводной лекции мы, прежде всего, обсудим предпосылки появления в компьютерах устройств внешней памяти, а также обоснуем принципиальную важность для организации информационных систем дисковых устройств с подвижными магнитными головками. Далее будут рассмотрены особенности организации и основное функциональное назначение одного из ключевых компонентов современных операционных систем – систем управления файлами. Наконец, в третьем разделе лекции мы покажем, почему возможностей файловых систем недостаточно для создания информационных программных систем. Будет продемонстрировано, что естественные требования информационных систем к средствам управления данными во внешней памяти приводят к необходимости наличия систем управления базами данных (СУБД). В ходе этого анализа будут определены основные черты, которыми должны обладать СУБД.

Ключевые слова: внешняя память, устройство внешней памяти, информационная система, магнитный диск с подвижными головками, система управления файлами, система управления базами данных, СУБД, целостность данных, ссылочная целостность, общее ограничение целостности, язык запросов к базе данных, полусоединение, транзакционное управление, журнализация, синхронизация параллельного доступа к данным и архитектуре «клиент-сервер».

Устройства внешней памяти

В самом широком смысле информационная система представляет собой программный комплекс, функции которого состоят в поддержке надежного хранения информации в памяти компьютера, выполнении специфических для данного приложения преобразований информации и/или вычислений, предоставлении пользователям удобного и легко осваиваемого интерфейса. Обычно объемы данных, с которыми приходится иметь дело таким системам, достаточно велики, а сами данные обладают достаточно сложной структурой. Классическими примерами информационных систем являются банковские системы, системы резервирования авиационных или железнодорожных билетов, мест в гостиницах и т. д.

О надежном и долговременном хранении информации можно говорить только при наличии запоминающих устройств, сохраняющих информацию после выключения электропитания. Оперативная (основная) память этим свойством обычно не обладает. В первые десятилетия развития вычислитель-

ной техники использовались два вида устройств внешней памяти: магнитные ленты и магнитные барабаны. При этом емкость магнитных лент была достаточно велика, но по своей природе они обеспечивали последовательный доступ к данным. Емкость магнитной ленты пропорциональна ее длине. Чтобы получить доступ к требуемой порции данных, нужно в среднем перемотать половину ее длины. Но чисто механическую операцию перемотки нельзя выполнить очень быстро. Поэтому быстрый произвольный доступ к данным на магнитной ленте, очевидно, невозможен.

Магнитный барабан представлял собой массивный металлический цилиндр с намагниченной внешней поверхностью и неподвижным пакетом магнитных головок. Такие устройства обеспечивали возможность достаточно быстрого произвольного доступа к данным, но позволяли сохранять сравнительно небольшой объем хранения данных. Быстрый произвольный доступ осуществлялся благодаря высокой скорости вращения барабана и наличию отдельной головки на каждую дорожку магнитной поверхности; ограниченность объема была обусловлена наличием всего одной магнитной поверхности.

Указанные ограничения не очень существенны для систем численных расчетов. Обсудим более подробно, какие реальные потребности возникают у разработчиков систем численных расчетов. Прежде всего, для получения требуемых результатов серьезные вычислительные программы должны проработать достаточно долгое время (недели, месяцы и даже, может быть, годы). Наличие гарантий надежности со стороны производителей аппаратных компьютерных средств не избавляет программистов от необходимости использования программного сохранения частичных результатов вычислений, чтобы при возникновении непредвиденных сбоев аппаратуры можно было продолжить выполнение расчетов с некоторой контрольной точки. Для сохранения промежуточных результатов идеально подходят магнитные ленты: при выполнении процедуры установки контрольной точки данные последовательно сбрасываются на ленту, а при необходимости перезапуска от сохраненной контрольной точки данные также последовательно с ленты считываются.

Вторая традиционная потребность численных программистов – максимально большой объем оперативной памяти. Большая оперативная память требуется, во-первых, для того, чтобы обеспечить программе быстрый доступ к большому количеству обрабатываемых данных. Во-вторых, сложные вычислительные программы сами могут иметь большой объем. Поскольку объем реально доступной в ЭВМ оперативной памяти всегда являлся недостаточным для удовлетворения текущих потребностей вычислений, требовалась быстрая внешняя память для организации оверлеев и/или виртуальной памяти. Мы не будем здесь вдаваться в детали организации этих механизмов программного расширения оперативной памяти, но заметим, что для этого идеально подходили магнитные барабаны. Они обеспечивают быстрый до-

ступ к внешней памяти, а для расширения оперативной памяти одной программы (сложные вычислительные программы, как правило, выполняются на компьютере в одиночку) большой объем внешней памяти не требуется.

Далее заметим, что, даже если программа должна обработать (или произвести) большой объем информации, при программировании можно продумать расположение этой информации во внешней памяти, чтобы программа работала как можно быстрее. Развитая поддержка работы с внешней памятью со стороны общесистемных программных средств не обязательна, а иногда и вредна, поскольку приводит к дополнительным накладным расходам аппаратных ресурсов.

Однако для информационных систем, в которых объем постоянно хранимых данных определяется спецификой бизнес-приложения, а потребность в текущих данных определяется пользователем приложения, одних только магнитных барабанов и лент недостаточно. Емкость магнитного барабана просто не позволяет долговременно хранить данные большого объема. Что же касается лент, то представьте себе состояние человека, который, стоя у билетной кассы, должен дождаться полной перемотки магнитной ленты. Естественным требованиям к таким системам является обеспечение высокой средней скорости выполнения операций при наличии больших объемов данных.

Именно требования к устройствам внешней памяти со стороны бизнес-приложений вызвали появление устройств внешней памяти со съемными пакетами магнитных дисков и подвижными головками чтения/записи, что явилось революцией в истории вычислительной техники. Эти устройства памяти обладали существенно большей емкостью, чем магнитные барабаны (за счет наличия нескольких магнитных поверхностей), обеспечивали удовлетворительную скорость доступа к данным в режиме произвольной выборки, а возможность смены дискового пакета на устройстве позволяла иметь архив данных практически неограниченного объема.

Магнитные диски представляют собой пакеты магнитных пластин (поверхностей), между которыми на одном рычаге двигается пакет магнитных головок (рис. 1.1). Шаг движения пакета головок является дискретным, и каждому положению пакета головок логически соответствует цилиндр пакета магнитных дисков. На каждой поверхности цилиндр «высекает» дорожку, так что каждая поверхность содержит число дорожек, равное числу цилиндров. При разметке магнитного диска (специальном действии, предшествующем использованию диска) каждая дорожка размечается на одно и то же количество блоков; таким образом, предельная емкость каждого блока составляет одно и то же число байтов. Для произведения обмена с магнитным диском на уровне аппаратуры нужно указать номер цилиндра, номер поверхности, номер блока на соответствующей дорожке и число байтов, которое нужно записать или прочитать от начала этого блока.

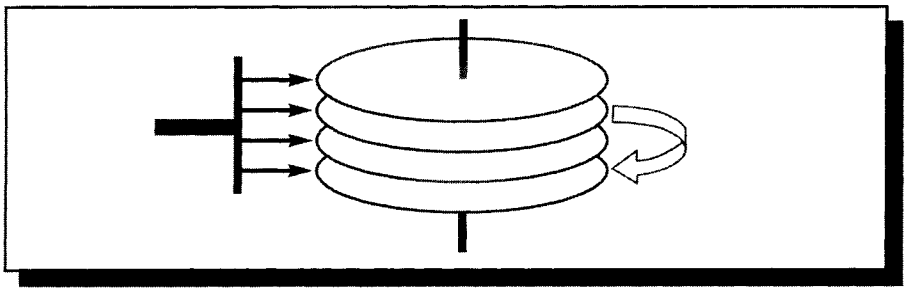


Рис. 1.1. Грубая схема дискового устройства памяти с подвижными головками

При выполнении обмена с диском аппаратура выполняет три основных действия: подвод головок к нужному цилиндру (обозначим время выполнения этого действия как $t_{пг}$), поиск на дорожке нужного блока (время выполнения – $t_{пб}$) и собственно обмен с этим блоком (время выполнения – $t_{об}$). Тогда, как правило, $t_{пг} \gg t_{пб} \gg t_{об}$, потому что подвод головок – это механическое действие, причем в среднем нужно переместить головки на расстояние, равное половине радиуса поверхности, а скорость передвижения головок не может быть слишком большой по физическим соображениям. Поиск блока на дорожке требует прокручивания пакета магнитных дисков в среднем на половину длины внешней окружности; скорость вращения диска может быть существенно больше скорости движения головок, но она тоже ограничена законами физики. Для выполнения же обмена нужно прокрутить пакет дисков всего лишь на угловое расстояние, соответствующее размеру блока. Таким образом, из всех этих действий в среднем наибольшее время занимает первое, и поэтому существенный выигрыш в суммарном времени обмена при считывании или записи только части блока получить практически невозможно.

С появлением магнитных дисков началась история систем управления данными во внешней памяти. До этого каждая прикладная программа, которой требовалось хранить данные во внешней памяти, сама определяла расположение каждой порции данных на магнитной ленте или барабане и выполняла обмены между оперативной и внешней памятью с помощью программно-аппаратных средств низкого уровня (машинных команд или вызовов соответствующих программ операционной системы). Такой режим работы не позволял или очень затруднял поддержание на одном внешнем носителе нескольких архивов долговременно хранимой информации. Кроме того, каждой прикладной программе приходилось решать проблемы именования частей данных и структуризации данных во внешней памяти.

Файловые системы

Историческим шагом стал переход к использованию систем управления файлами. С точки зрения прикладной программы файл – это именованная область внешней памяти, в которую можно записывать и из которой можно считывать данные. Правила именования файлов, способ доступа к данным, хранящимся в файле, и структура этих данных зависят от конкретной системы управления файлами и, возможно, от типа файла. Система управления файлами берет на себя распределение внешней памяти, отображение имен файлов в соответствующие адреса внешней памяти и обеспечение доступа к данным.

В этом разделе мы рассмотрим историю файловых систем, их основные черты и области разумного применения. Однако сначала сделаем два замечания. Во-первых, в области управления файлами исторически существует некоторая терминологическая путаница. Термин *файловая система (file system)* используется для обозначения программной системы, управляющей файлами, и архива файлов, хранящегося во внешней памяти. Было бы лучше в первом случае использовать термин *система управления файлами*, оставив за термином *файловая система* только второе значение. Однако принятая практика заставляет нас использовать термин *файловая система* в обоих смыслах. Будем надеяться, что точный смысл термина будет понятен из контекста. (Заметим, что среди непрофессионалов аналогичная путаница возникает при использовании терминов *база данных* и *система управления базами данных*. В этом курсе мы будем строго разделять эти термины.) Во-вторых, мы ограничимся описанием свойств так называемых традиционных файловых систем, не обсуждая особенности современных систем с повышенной надежностью, поскольку это заставило бы нас сильно отклониться от основной темы курса.

Первая развитая файловая система была разработана специалистами IBM в середине 60-х гг. для выпускавшейся компанией серии компьютеров «360». В этой системе поддерживались как чисто последовательные, так и индексно-последовательные файлы, а реализация во многом опиралась на возможности только появившихся к этому времени контроллеров управления дисковыми устройствами. Контроллеры обеспечивали возможность обмена с дисковыми устройствами порциями данных произвольного размера, а также индексный доступ к записям файлов, и эти функции контроллеров активно использовались в файловой системе OS/360.

Файловая система OS/360 обеспечила будущих разработчиков уникальным опытом использования дисковых устройств с подвижными головками, который отражается во всех современных файловых системах.

Структуры файлов

Практически во всех современных компьютерах основными устройствами внешней памяти являются магнитные диски с подвижными головками, и именно они служат для хранения файлов. Как отмечалось ранее, аппаратура магнитных дисков допускает выполнение обмена с дисками порциями данных произвольного размера. Однако возможность обмениваться с магнитными дисками порциями, размеры которых меньше полного объема блока, в настоящее время в файловых системах не используется. Это связано с двумя обстоятельствами.

Во-первых, как указывалось в разделе «Устройства внешней памяти», считывание или запись только части блока не приводит к существенному выигрышу в суммарном времени обмена. Во-вторых, для работы с частями блоков файловая система должна обеспечить буферы оперативной памяти соответствующего размера, что существенно усложняет распределение оперативной памяти. Алгоритмы распределения памяти порциями произвольного размера плохи тем, что любой из них рано или поздно приводит к *внешней фрагментации* памяти. В памяти образуется большое число маленьких свободных фрагментов. Их совокупный размер может быть больше размера любого требуемого буфера, но его можно выделить, только если произвести сжатие памяти, т. е. подвижку всех занятых фрагментов таким образом, чтобы они располагались вплотную один к другому. Во время выполнения операции сжатия памяти нужно приостановить выполнение обменов, а сама эта операция занимает много времени.

Поэтому во всех современных файловых системах явно или неявно выделяется уровень, обеспечивающий работу с *базовыми файлами*, которые представляют собой наборы блоков, последовательно нумеруемых в адресном пространстве файла и отображаемых на физические блоки диска (рис. 1.2). Размер логического блока файла совпадает с размером физического блока диска или кратен ему; обычно размер логического блока выбирается равным размеру страницы виртуальной памяти, поддерживаемой аппаратурой компьютера совместно с операционной системой.

В некоторых файловых системах базовый уровень был доступен пользователю, но чаще он прикрывался некоторым более высоким уровнем, стандартным для пользователей. Существуют два основных подхода. При первом подходе, свойственном, например, файловым системам операционных систем компании DEC RSX и VMS, пользователи представляют файл как последовательность записей. Каждая запись – это последовательность байтов, имеющая постоянный или переменный размер. Можно читать или писать записи последовательно либо позиционировать файл на запись с указанным номером. Некоторые файловые системы позволяют структурировать записи на поля и объявлять некие поля ключами записи.

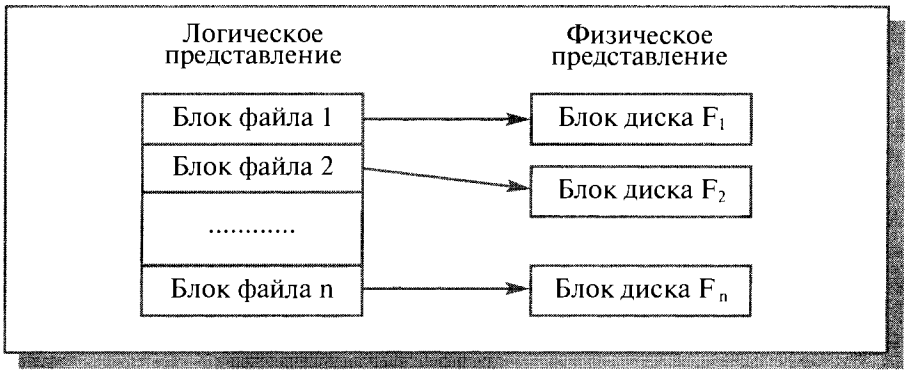


Рис. 1.2. Схематичное изображение базового файла

В таких файловых системах можно потребовать выборку записи из файла по ее заданному ключу. Естественно, в этом случае файловая система поддерживает в том же (или другом, служебном) базовом файле дополнительные, невидимые пользователю, служебные структуры данных. Распространенные способы организации ключевых файлов основываются на технике хэширования и B-деревьев. Существуют и многоключевые способы организации файлов (у одного файла объявляется несколько ключей, и можно выбирать записи по значению каждого ключа).

Второй подход, получивший распространение вместе с операционной системой UNIX, состоит в том, что любой файл представляется как непрерывная последовательность байтов. Из файла можно прочитать указанное число байтов, либо начиная с его начала, либо предварительно выполнив его позиционирование на байт с указанным номером. Аналогично можно записать указанное число байтов либо в конец файла, либо предварительно выполнив позиционирование файла. Тем не менее заметим, что скрытым от пользователя, но существующим во всех разновидностях файловых систем ОС UNIX является базовое блочное представление файла.

Конечно, в обоих случаях можно обеспечить набор преобразующих функций, приводящих представление файла к другому виду. Примером тому может служить поддержка стандартной файловой среды UNIX в среде операционных систем компании DEC.

Логическая структура файловых систем и именование файлов

Во всех современных файловых системах обеспечивается многоуровневое именование файлов за счет наличия во внешней памяти каталогов – дополнительных файлов со специальной структурой. Каждый каталог содержит имена каталогов и/или файлов, хранящихся в данном каталоге.

Таким образом, полное имя файла состоит из списка имен каталогов плюс имя файла в каталоге, непосредственно содержащем данный файл.

Поддержка многоуровневой схемы именования файлов обеспечивает несколько преимуществ, основным из которых является простая и удобная схема логической классификации файлов и генерации их имен. Можно сопоставить каталог или цепочку каталогов с пользователем, подразделением, проектом и т. д. и затем образовывать в этом каталоге файлы или каталоги, не опасаясь коллизий с именами других файлов или каталогов.

Разница между способами именования файлов в разных файловых системах состоит в том, с чего начинается эта цепочка имен. В любом случае первое имя должно соответствовать корневому каталогу файловой системы. Вопрос заключается в том, как сопоставить этому имени корневой каталог – где его искать? В связи с этим имеются два радикально различных подхода.

Во многих системах управления файлами требуется, чтобы каждый архив файлов (полное дерево каталогов) целиком располагался на одном дисковом пакете или логическом диске – разделе физического дискового пакета, логически представляемом в виде отдельного диска с помощью средств операционной системы. В этом случае полное имя файла начинается с имени дискового устройства, на котором установлен соответствующий диск. Такой способ именования использовался в файловых системах компаний IBM и DEC; очень близки к этому и файловые системы, реализованные в операционных системах семейства Windows компании Microsoft. Можно назвать такую организацию поддержкой изолированных файловых систем.

Другой крайний вариант был реализован в файловых системах операционной системы Multics. Эта система заслуживает отдельного разговора, в ней был реализован целый ряд оригинальных идей, но мы остановимся только на особенностях организации архива файлов. В файловой системе Multics пользователям обеспечивалась возможность представлять всю совокупность каталогов и файлов в виде единого дерева. Полное имя файла начиналось с имени корневого каталога, и пользователь не обязан был заботиться об установке на дисковое устройство каких-либо конкретных дисков. Сама система, выполняя поиск файла по его имени, запрашивала у оператора установку необходимых дисков. Такую файловую систему можно назвать полностью централизованной.

Конечно, во многом централизованные файловые системы удобнее изолированных: система управления файлами выполняет больше рутинной работы. В частности, администратор файловой системы автоматически оповещается о потребности установки требуемых дисковых пакетов; система обеспечивает равномерное распределение памяти на известных ей дисковых томах; возможна организация автоматического перемещения редко используемых файлов на более медленные носители внешней памяти; облегчается рутинная работа, связанная с резервным копированием.

Но в таких системах возникают существенные проблемы, если требуется перенести поддерево файловой системы на другую вычислительную установку. Поскольку файлы и каталоги любого логического поддерева могут быть физически разбросаны по разным дисковым пакетам и даже магнитным лентам, для такого переноса требуется специальная утилита, собирающая все объекты требуемого поддерева на одном внешнем носителе, не входящем в состав штатных устройств централизованной файловой системы. Конечно, даже при наличии такой утилиты выполнение процедуры физической сборки требует существенного времени.

Компромиссное решение применяется в файловых системах ОС UNIX. На базовом уровне в этих файловых системах поддерживаются изолированные архивы файлов. Один из таких архивов объявляется корневой файловой системой. Это делается на этапе генерации операционной системы, и после запуска операционная система «знает», на каком дисковом устройстве (физическом или логическом) располагается корневая файловая система. После запуска системы можно «смонтировать» корневую файловую систему и ряд изолированных файловых систем в одну общую файловую систему. Технически это осуществляется посредством создания в корневой файловой системе специальных пустых каталогов (точек монтирования).

Специальный системный вызов *mount* ОС UNIX позволяет подключить к одному из пустых каталогов корневой каталог указанного архива файлов. Выполнение такого действия приводит к «наложению» корневого каталога монтируемой файловой системы на каталог точки монтирования; корневой каталог приобретает имя каталога точки монтирования. После монтирования общей файловой системы именование файлов производится так же, как если бы она с самого начала была централизованной. Если учесть, что обычно монтирование файловой системы производится при раскрутке системы (при выполнении стартового командного файла), пользователи ОС UNIX, как правило, и не задумываются о происхождении общей файловой системы.

Кроме того, поддерживается системный вызов *unmount*, «отторгающий» ранее смонтированную файловую систему от общей иерархии. Конечно, все это заметно облегчает перенос частей файловой системы на другие установки.

Авторизация доступа к файлам

Поскольку файловая система является общим хранилищем файлов, принадлежащих, вообще говоря, разным пользователям, системы управления файлами должны обеспечивать *авторизацию* доступа к файлам. В общем виде подход состоит в том, что по отношению к каждому зарегистрированному пользователю данной вычислительной системы для каждого

существующего файла указываются действия, которые разрешены или запрещены данному пользователю (так называемый *мандатный* способ защиты – каждый пользователь имеет отдельный мандат для работы с каждым файлом или не имеет его). Применение мандатного способа защиты влечет за собой существенные накладные расходы, связанные с потребностью хранения избыточной информации и использованием этой информации для проверки правомочности доступа.

Поэтому в большинстве современных систем управления файлами применяется подход к защите файлов, впервые реализованный в ОС UNIX (так называемый *дискреционный* подход). В этой системе каждому зарегистрированному пользователю соответствует пара целочисленных идентификаторов: идентификатор группы, к которой относится пользователь, и его собственный идентификатор. Этими же идентификаторами снабжается каждый процесс, запущенный от имени данного пользователя и имеющий возможность обращаться к системным вызовам файловой системы. Соответственно, при каждом файле хранится полный идентификатор пользователя (собственный идентификатор плюс идентификатор группы), который создал этот файл, и помечается, какие действия с файлом может производить он сам, какие действия с файлом доступны для остальных пользователей той же группы и что могут делать с файлом пользователи других групп. Для каждого файла контролируется возможность выполнения трех действий: чтение, запись и выполнение. Хранимая информация очень компактна (два целых числа для представления идентификаторов и шкала из 9 бит для характеристики возможных действий), при проверке требуется небольшое количество действий, и этот способ контроля доступа в большинстве случаев удовлетворителен.

Синхронизация многопользовательского доступа

Последнее, на чем мы остановимся в связи с файлами, – это способы применения файлов в многопользовательской среде. Если операционная система поддерживает многопользовательский режим, может возникнуть ситуация, когда два или более пользователей одновременно пытаются работать с одним и тем же файлом. Если все эти пользователи собираются только читать файл, ничего страшного не произойдет. Но если хотя бы один из них будет изменять файл, для корректной работы этой группы требуется взаимная синхронизация.

В файловых системах обычно применялся следующий подход. В операции открытия файла (первой и обязательной операции, с которой должен начинаться сеанс работы с файлом) помимо прочих параметров указывался режим работы (чтение или изменение). Если к моменту выполнения этой операции от имени некоторого процесса *A* файл уже был открыт некоторым

другим процессом *B*, причем существующий режим открытия был несовместим с требуемым режимом (совместимы только режимы чтения), то в зависимости от особенностей системы либо процессу *A* сообщалось о невозможности открытия файла в нужном режиме, либо процесс *A* блокировался до тех пор, пока процесс *B* не выполнит операцию закрытия файла.

Области разумного применения файлов

После краткого экскурса в историю и современное состояние файловых систем обсудим возможные области их применения. Прежде всего, конечно, файлы используются для хранения текстовых данных: документов, текстов программ и т. д. Такие файлы обычно создаются и модифицируются с помощью различных текстовых редакторов. Эти редакторы могут быть очень простыми, такими, как **ed** в мире UNIX или утилиты редактирования **Norton Commander**, **FAR Manager** и других интерактивных сред Windows. Они могут быть сложными и многофункциональными, синтаксически ориентированными, как, например, GNU **Emacs**. Но обычно структура текстовых файлов очень проста (с точки зрения файловой системы): это либо последовательность записей, содержащих строки текста, либо последовательность байтов, среди которых встречаются специальные символы (например, символы конца строки). Конечно же, сложность логической структуры текстового файла определяется текстовым редактором, но в любом случае файловой системе она не видна.

Файлы, содержащие тексты программ, используются как входные файлы компиляторов (чтобы правильно воспринять текст программы, компилятор должен понимать логическую структуру текстового файла), которые, в свою очередь, формируют файлы, содержащие объектные модули. С точки зрения файловой системы объектные файлы также обладают очень простой структурой – последовательность записей или байтов. Система программирования накладывает на такую структуру более сложную и специфичную для этой системы структуру объектного модуля. Подчеркнем, что логическая структура объектного модуля файловой системе неизвестна; эта структура поддерживается инструментами системы программирования.

Аналогично обстоит дело с файлами, формируемыми редакторами связей (редактор связей должен понимать логическую структуру файлов объектных модулей) и содержащими образы выполняемых программ. Логическая структура таких файлов остается известной только редактору связей и загрузчику – программе операционной системы. Общая схема взаимодействия программных компонентов при построении программы показана на рис. 1.3. Мы кратко обозначили способы использования файлов в процессе разработки программ, но можно сказать, что ситуация

аналогична и в других случаях: например, при образовании и использовании файлов, содержащих графическую, аудио- и видеоинформацию.

Одним словом, файловые системы обычно обеспечивают хранение слабо структурированной информации, оставляя дальнейшую структуризацию прикладным программам. В перечисленных выше случаях использования файлов это даже хорошо, потому что при разработке любой новой прикладной системы, опираясь на простые, стандартные и сравнительно дешевые средства файловой системы, можно реализовать те структуры хранения, которые наиболее точно соответствуют специфике данной прикладной области.

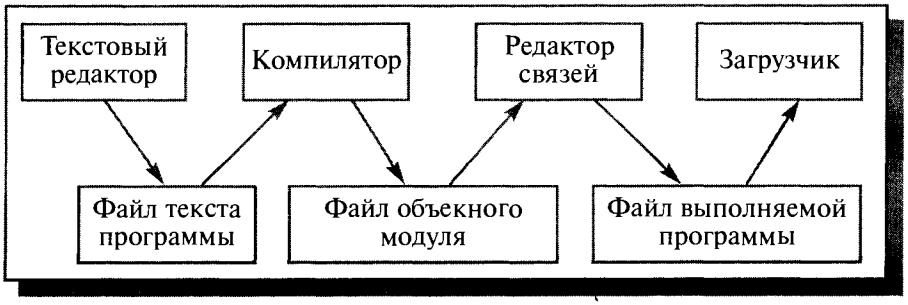


Рис. 1.3. Связи между программными компонентами по пониманию логической структуры файлов

Потребности информационных систем

Удовлетворяют ли рассмотренные выше базовые возможности файловых систем потребности информационных систем? Типовая информационная система, главным образом, ориентирована на хранение, выбор и модификацию данных соответствующей прикладной области. Структура таких данных зачастую очень сложна, и, хотя структуры данных различны в разных информационных системах, между ними часто бывает много общего.

На начальном этапе использования вычислительной техники для построения информационных систем проблемы структуризации данных решались индивидуально в каждой информационной системе. Производились необходимые надстройки над файловыми системами (библиотеки программ), подобно тому как это делается в компиляторах, редакторах и т. д. (рис. 1.4).

Но поскольку информационные системы требуют сложных структур данных, эти дополнительные индивидуальные средства управления данными являлись существенной частью информационных систем и практически повторялись от одной системы к другой. Стремление выделить общую часть информационных систем, ответственную за управление сложно структурированными данными, явилось, на мой взгляд, первой побудительной причиной со-

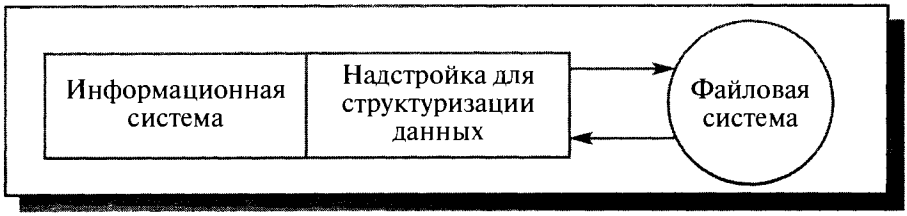


Рис. 1.4. Примитивная схема структуризации данных в информационной системе

здания СУБД. Очень скоро стало понятно, что невозможно обойтись общей библиотекой программ (рис. 1.5), реализующей над стандартной базовой файловой системой более сложные методы хранения данных.

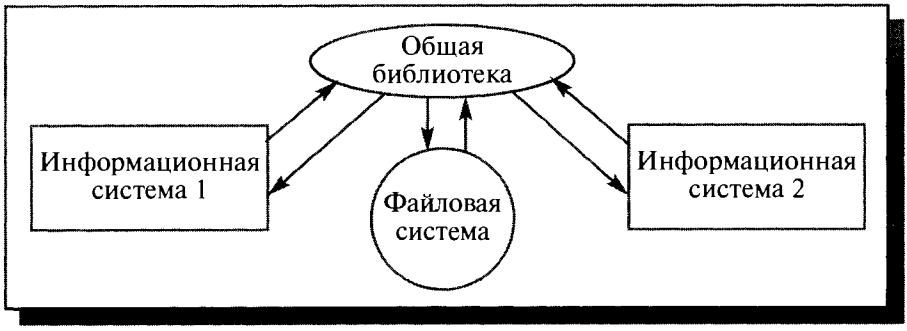


Рис. 1.5. Две информационные системы с общей библиотекой

Поясним это на примере. Предположим, что требуется реализовать простую информационную систему, поддерживающую учет служащих некоторой организации. Система должна выполнять следующие действия:

- выдавать списки служащих по отделам;
- поддерживать возможность перевода служащего из одного отдела в другой;
- обеспечивать средства поддержки приема на работу новых служащих и увольнения работающих служащих.

Кроме того, для каждого отдела должна поддерживаться возможность получения:

- имени руководителя отдела;
- общей численности отдела;
- общей суммы зарплаты служащих отдела, среднего размера зарплаты и т. д.

Для каждого служащего должна поддерживаться возможность получения:

- номера удостоверения по полному имени служащего (для простоты допустим, что имена всех служащих различны);

- полного имени по номеру удостоверения;
- информации о соответствии служащего занимаемой должности и о размере его зарплаты.

Структуры данных

Предположим, что мы решили основывать эту информационную систему на файловой системе и пользоваться одним файлом СЛУЖАЩИЕ, расширив базовые возможности файловой системы за счет специальной библиотеки функций. Поскольку минимальной информационной единицей в нашем случае является служащий, в этом файле должна содержаться одна запись для каждого служащего. Чтобы можно было удовлетворить указанные выше требования, запись о служащем должна иметь следующие поля:

- полное имя служащего (СЛУ_ИМЯ);
- номер его удостоверения (СЛУ_НОМЕР);
- данные о соответствии служащего занимаемой должности (СЛУ_СТАТ; для простоты «да» или «нет»);
- размер зарплаты (СЛУ_ЗАРП);
- номер отдела (СЛУ_ОТД_НОМЕР).

Поскольку мы решили ограничиться одним файлом СЛУЖАЩИЕ, та же запись должна содержать имя руководителя отдела (СЛУ_ОТД_РУК). (Иначе было бы невозможно, например, получить имя руководителя отдела с известным номером.)

Чтобы информационная система могла эффективно выполнять свои базовые функции, необходимо обеспечить многоключевой доступ к файлу СЛУЖАЩИЕ по уникальным ключам (ключ называется уникальным, если его значения гарантированно различны во всех записях файла) СЛУ_ИМЯ и СЛУ_НОМЕР. Очевидно, что в противном случае для выполнения наиболее часто используемых операций получения данных о конкретном служащем понадобится последовательный просмотр в среднем половины записей файла. Кроме того, должна обеспечиваться возможность эффективного выбора всех записей с общим значением СЛУ_ОТД_НОМЕР, т. е. доступ по неуникальному ключу. Если не поддерживать специальный механизм доступа, то для получения данных об отделе в целом в общем случае потребуется полный просмотр файла. Требуемая общая структура файла СЛУЖАЩИЕ показана на рис. 1.6. Но даже в этом случае, чтобы получить численность отдела или общий размер зарплаты, система должна будет выбрать все записи о служащих указанного отдела и посчитать соответствующие общие значения.

Таким образом, мы видим, что при реализации даже такой простой информационной системы на базе файловой системы возникают следующие затруднения:

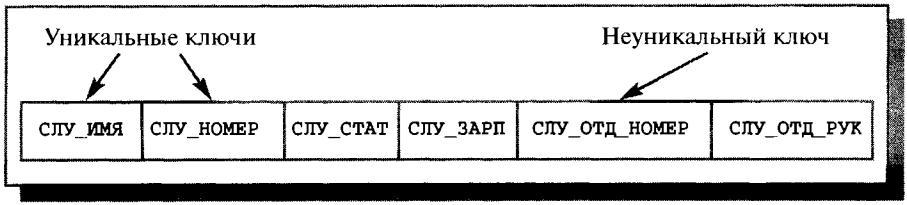


Рис. 1.6. Структура файла СЛУЖАЩИЕ на уровне приложения (случай одного файла)

- требуется создание достаточно сложной надстройки для многоключевого доступа к файлам;
- возникает существенная избыточность данных (для каждого служащего повторяется имя руководителя его отдела);
- требуется выполнение массовой выборки и вычислений для получения суммарной информации об отделах.

Кроме того, если в ходе эксплуатации системы потребуется, например, обеспечить операцию выдачи списков служащих, получающих указанную зарплату, то либо придется при выполнении каждой такой операции полностью просматривать файл, либо нужно будет реструктурировать файл СЛУЖАЩИЕ, объявляя ключевым и поле СЛУ_ЗАРП.

Для улучшения ситуации можно было бы поддерживать два многоключевых файла: СЛУЖАЩИЕ и ОТДЕЛЫ. Первый файл должен был бы содержать поля СЛУ_ИМЯ, СЛУ_НОМЕР, СЛУ_СТАТ, СЛУ_ЗАРП и СЛУ_ОТД_НОМЕР, а второй – ОТД_НОМЕР, ОТД_РУК (номер удостоверения служащего, являющегося руководителем отдела), ОТД_СЛУ_ЗАРП (общий размер зарплаты служащих данного отдела) и ОТД_РАЗМЕР (общее число служащих в отделе). Структура этих файлов показана на рис. 1.7.

Введение этих двух файлов позволило бы преодолеть большинство неудобств, перечисленных в предыдущем абзаце. Каждый из файлов содержал

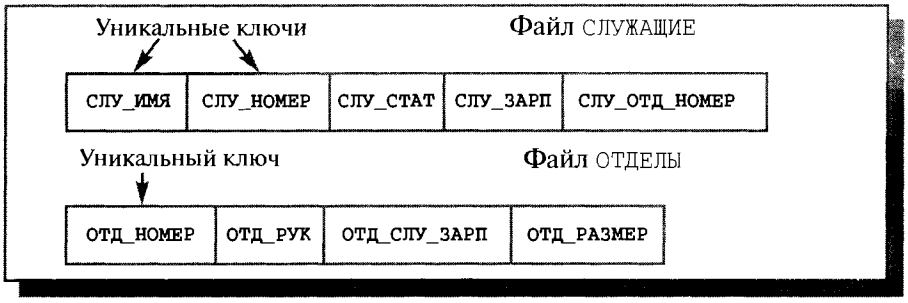


Рис. 1.7. Структура файла СЛУЖАЩИЕ и ОТДЕЛЫ на уровне приложения (случай двух файлов)

бы только не дублируемую информацию, не возникала бы необходимость в динамических вычислениях суммарной информации по отделам. Но заметим, что при таком переходе наша информационная система должна обладать некоторыми новыми особенностями, сближающими ее с СУБД.

Целостность данных

Теперь система должна «знать», что она работает с двумя информационно связанными файлами (это шаг в сторону схемы базы данных), должна иметь информацию о структуре и смысле каждого поля. Например, системе должно быть известно, что у полей СЛУ_ОТД_НОМЕР в файле СЛУЖАЩИЕ и ОТД_НОМЕР в файле ОТДЕЛЫ один и тот же смысл – номер отдела.

Кроме того, система должна учитывать, что в ряде случаев изменение данных в одном файле должно автоматически вызывать модификацию второго файла, чтобы общее содержимое файлов было согласованным. Например, если на работу принимается новый служащий, то нужно добавить запись в файл СЛУЖАЩИЕ, а также должным образом изменить поля ОТД_СЛУ_ЗАРП и ОТД_РАЗМЕР в записи файла ОТДЕЛЫ, соответствующей отделу этого служащего. Более точно, система должна руководствоваться следующими правилами:

1. если в файле СЛУЖАЩИЕ содержится запись со значением поля СЛУ_ОТД_НОМЕР, равным *n*, то и в файле ОТДЕЛЫ должна содержаться запись со значением поля ОТД_НОМЕР, также равным *n*;
2. если в файле ОТДЕЛЫ содержится запись со значением поля ОТД_РУК, равным *m*, то и в файле СЛУЖАЩИЕ должна содержаться запись со значением поля СЛУ_НОМЕР, также равным *m*; в следующих лекциях мы увидим, что правила (1) и (2) являются частными случаями общего правила *ссылочной целостности:* поле СЛУ_ОТД_НОМЕР содержит «ссылки» на записи таблицы ОТДЕЛЫ, и поле ОТД_РУК содержит «ссылки» на записи таблицы СЛУЖАЩИЕ;
3. при любом корректном состоянии информационной системы значение поля ОТД_СЛУ_ЗАРП любой записи отд_k файла ОТДЕЛЫ должно быть равно сумме значений поля СЛУ_ЗАРП всех тех записей файла СЛУЖАЩИЕ, в которых значение поля СЛУ_ОТД_НОМЕР совпадает со значением поля ОТД_НОМЕР записи отд_k;
4. при любом корректном состоянии информационной системы значение поля ОТД_РАЗМЕР любой записи отд_k файла ОТДЕЛЫ должно быть равно числу всех тех записей файла СЛУЖАЩИЕ, в которых значение поля СЛУ_ОТД_НОМЕР совпадает со значением поля ОТД_НОМЕР записи отд_k; в следующих лекциях мы увидим, что правила (3) и (4) представляют собой примеры общих ограничений целостности базы данных.

Понятие согласованности, или целостности, данных является ключевым понятием баз данных. Фактически, если информационная система

(даже такая простая, как в нашем примере) поддерживает согласованное хранение данных в нескольких файлах, можно говорить о том, что она поддерживает базу данных (БД). Если же некоторая вспомогательная система управления данными позволяет работать с несколькими файлами, обеспечивая их согласованность, можно назвать ее системой управления базами данных (СУБД).

Уже только требование поддержания согласованности данных в нескольких файлах не позволяет при построении информационной системы обойтись библиотекой функций: такая система должна обладать некоторыми собственными данными (их принято называть метаданными), определяющими целостность данных. В нашем примере информационная система должна отдельно сохранять метаданные о структуре файлов СЛУЖАЩИЕ и ОТДЕЛЫ, а также правила, определяющие условия целостности данных в этих файлах (принято считать, что правила также составляют часть метаданных).

Языки запросов

Но обеспечение целостности данных – это далеко не все, что обычно требуется от СУБД. Начнем с того, что даже в нашем примере пользователю информационной системы будет не слишком просто получить, например, общую численность отдела, в котором работает Петр Иванович Сидоров. Придется сначала узнать номер отдела, в котором работает указанный служащий, а затем установить численность этого отдела. Было бы гораздо проще, если бы СУБД позволяла сформулировать такой запрос на языке, более близком пользователям. Такие языки называются *языками запросов к базам данных*. Например, на языке запросов SQL наш запрос можно было бы выразить в следующей форме (*запрос 1*):

```
SELECT ОТД_РАЗМЕР
   FROM СЛУЖАЩИЕ, ОТДЕЛЫ
   WHERE СЛУ_ИМЯ = 'ПЕТР ИВАНОВИЧ СИДОРОВ' AND
      СЛУ_ОТД_НОМЕР = ОТД_НОМЕР;
```

Это пример запроса на языке SQL с «*полусоединением*»: с одной стороны, запрос адресуется к двум файлам – СЛУЖАЩИЕ и ОТДЕЛЫ, но с другой стороны, данные выбираются только из файла ОТДЕЛЫ. Условие СЛУ_ОТД_НОМЕР = ОТД_НОМЕР всего лишь «ограничивает» интересующий нас набор записей об отделах до одной записи, если Петр Иванович Сидоров действительно работает на данном предприятии. Если же Петр Иванович Сидоров не работает на предприятии, то условие СЛУ_ИМЯ = 'ПЕТР ИВАНОВИЧ СИДОРОВ' не будет удовлетворяться ни для одной записи файла СЛУЖАЩИЕ, и поэтому запрос выдаст пустой результат.

Возможна и другая формулировка того же запроса (*запрос2*):

```
SELECT ОТД_РАЗМЕР
   FROM ОТДЕЛЫ
   WHERE ОТД_НОМЕР =
      (SELECT СЛУ_ОТД_НОМЕР
         FROM СЛУЖАЩИЕ
         WHERE СЛУ_ИМЯ = 'ПЕТР ИВАНОВИЧ СИДОРОВ');
```

Это пример запроса на языке SQL с *вложенным подзапросом*. Во вложенном подзапросе выбирается значение поля СЛУ_ОТД_НОМЕР из записи файла СЛУЖАЩИЕ, в которой значение поля СЛУ_ИМЯ равняется строковой константе 'ПЕТР ИВАНОВИЧ СИДОРОВ'. Если такая запись существует, то она единственная, поскольку поле СЛУ_ИМЯ является уникальным ключом файла СЛУЖАЩИЕ. Тогда результатом выполнения подзапроса будет единственное значение – номер отдела, в котором работает Петр Иванович Сидоров. Во внешнем запросе это значение будет ключом доступа к файлу ОТДЕЛЫ, и снова будет выбрана только одна запись, поскольку поле ОТД_НОМЕР является уникальным ключом файла ОТДЕЛЫ. Если же на данном предприятии Петр Иванович Сидоров не работает, то подзапрос выдаст пустой результат, и внешний запрос тоже выдаст пустой результат.

Приведенные примеры показывают, что при формулировке запроса с использованием SQL можно не задумываться о том, как будет выполняться этот запрос. Среди метаданных базы данных будет содержаться информация о том, что поле СЛУ_ИМЯ является ключевым для файла СЛУЖАЩИЕ (т. е. по заданному значению имени служащего можно быстро найти соответствующую запись или убедиться в том, что запись с таким значением поля СЛУ_ИМЯ в файле отсутствует), а поле ОТД_НОМЕР – ключевое для файла ОТДЕЛЫ (и более того, оба ключа в соответствующих файлах являются уникальными), и система сама воспользуется этим. Можно формально доказать, что формулировки *запрос1* и *запрос2* эквивалентны, т. е. вне зависимости от состояния данных всегда производят один и тот же результат. Наиболее вероятным способом выполнения запроса в обеих формулировках будет выборка записи из файла СЛУЖАЩИЕ со значением поля СЛУ_ИМЯ, равным строке 'ПЕТР ИВАНОВИЧ СИДОРОВ', взятие из этой записи значения поля СЛУ_ОТД_НОМЕР и выборка из таблицы ОТДЕЛЫ записи с таким же значением поля ОТД_НОМ.

Если же, например, возникнет потребность в получении списка сотрудников, не соответствующих занимаемой должности, то достаточно обратиться к системе с запросом (*запрос3*):

```
SELECT СЛУ_ИМЯ, СЛУ_НОМЕР
   FROM СЛУЖАЩИЕ
   WHERE СЛУ_СТАТ = "НЕТ";
```

и система сама выполнит необходимый полный просмотр файла СЛУЖАЩИЕ, поскольку поле СЛУ_СТАТ не является ключевым, и другого способа выполнения не существует.

Транзакции, журнализация и многопользовательский режим

Далее, представим себе, что в первоначальной реализации информационной системы, основанной на использовании библиотек расширенных методов доступа к файлам, обрабатывается операция принятия на работу нового служащего. Следуя требованиям согласованного изменения файлов, информационная система вставляет новую запись в файл СЛУЖАЩИЕ и собирается модифицировать соответствующую запись файла ОТДЕЛЫ (или вставлять в этот файл новую запись, если служащий является первым в своем отделе), но именно в этот момент происходит (например) аварийное выключение питания компьютера.

Очевидно, что после перезапуска системы ее база данных будет находиться в рассогласованном состоянии (точно будут нарушены правила (3) и (4), а может быть, и правила (1) и (2)). Потребуется выяснить это (а для этого нужно явно проверить соответствие данных в файлах СЛУЖАЩИЕ и ОТДЕЛЫ) и привести данные в согласованное состояние. Проверку и коррекцию можно выполнить, например, следующим образом. Сгруппировать записи файла СЛУЖАЩИЕ по значениям поля СЛУ_ОТД_НОМЕР. Для каждой группы (a) проверить, существует ли в файле ОТДЕЛЫ запись, значение поля ОТД_НОМ которой равняется значению поля СЛУ_ОТД_НОМЕР записей данной группы; если такой записи в файле ОТДЕЛЫ нет, то (b) исключить группу из файла СЛУЖАЩИЕ и перейти к обработке следующей группы; иначе (c) посчитать число записей в группе и вычислить суммарное значение заработной платы; (d) обновить полученными значениями поля ОТД_РАЗМЕР и ОТД_СЛУ_ЗАРП соответствующей записи файла ОТДЕЛЫ и перейти к обработке следующей группы.

Настоящие СУБД берут такую работу на себя, поддерживая *транзакционное управление* и *журнализацию* изменений базы данных. Прикладная система не обязана заботиться о поддержке корректности состояния базы данных, хотя и должна знать, какие цепочки операций изменения данных являются допустимыми.

Представим теперь, что в информационной системе требуется обеспечить параллельную (например, многотерминальную) работу с базой данных служащих и отделов. Если опираться только на использование файлов, то для

обеспечения корректности на все время модификации любого из двух файлов доступ других пользователей к этому файлу будет блокирован (вспомните возможности файловых систем в отношении синхронизации параллельного доступа, упоминавшиеся в разд. «Файловые системы»). Таким образом, зачисление на работу Петра Ивановича Сидорова существенно затормозит получение информации о служащем Иване Сидоровиче Петрове, даже если они работают в разных отделах. Настоящие СУБД обеспечивают гораздо более тонкую синхронизацию параллельного доступа к данным.

СУБД как независимый системный компонент

До сих пор мы не вычленяли СУБД из состава информационной системы, имея в виду общую организацию системы, подобную той, которая показана на рис. 1.8.

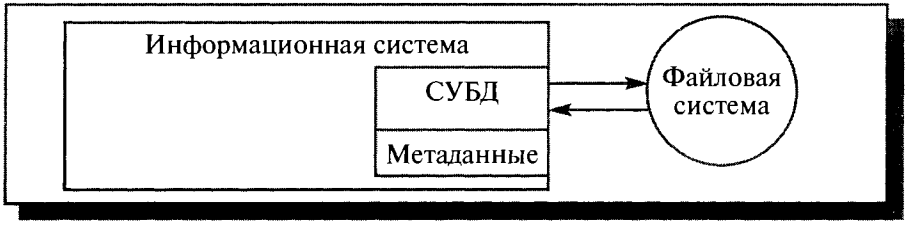


Рис. 1.8. СУБД в составе информационной системы

Здесь видны два дефекта. Во-первых, очевидно, что СУБД должна поддерживать достаточно развитую функциональность. Повторять эту функциональность в каждой информационной системе неразумно. С другой стороны, неясно, каким образом можно обеспечить готовый к использованию компонент СУБД, который можно было бы встраивать в информационные системы. Во-вторых, уже должно быть понятно, что набор файлов можно назвать базой данных только при наличии метаданных. На рис. 1.8 метаданные являются принадлежностью информационной системы, и поэтому, например, файлы СЛУЖАЩИЕ и ОТДЕЛЫ можно эффективно использовать только через нашу гипотетическую систему регистрации служащих.

Предположим, что предприятию нужна еще и информационная бухгалтерская система. Очевидно, что для ее работы также потребуются данные о служащих и отделах. При показанной выше организации системы возможны два варианта выполнения задачи, ни один из которых не является удовлетворительным.

1. Внедрить бухгалтерскую систему в состав системы регистрации сотрудников. Но ведь, как правило, бухгалтерские системы покупаются в виде готовых и отдельных продуктов, не приспособленных к подобному «внедрению».

2. Скопировать метаданные системы регистрации служащих в бухгалтерскую систему. Но метаданные (как и данные) не обязательно являются статичными. Структура базы данных может со временем изменяться, могут исчезать одни правила целостности и появляться другие. Как согласовывать копии метаданных, поддерживаемые независимыми информационными системами?

Так мы приходим к организации системы, показанной на рис. 1.9.

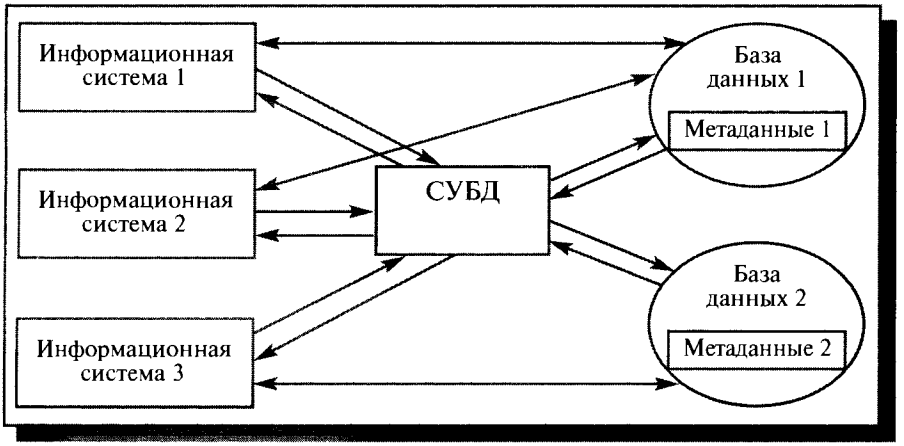


Рис. 1.9. Отдельная СУБД и базы данных с метаданными

Здесь мы видим три информационные системы, которые через одну СУБД работают с двумя разными базами данных, причем первая и вторая системы работают с общей базой данных. Это возможно, поскольку метаданные каждой базы данных содержатся в самих базах данных, и достаточно лишь указать СУБД, с какой базой данных желает работать данное приложение. Поскольку СУБД функционирует отдельно от приложений, и ее работа с базами данных регулируется метаданными, совместное использование одной базы данных двумя информационными системами не вызовет потери согласованности данных, и доступ к данным будет должным образом синхронизироваться. Заметим, что рис. 1.9 вплотную приближает нас к наиболее распространенной в последние десятилетия архитектуре «клиент-сервер». СУБД играет роль «сервера», обсуживающего нескольких «клиентов» – прикладных информационных систем.

Таким образом, СУБД решают множество проблем, которые затруднительно или вообще невозможно решить при использовании файловых систем. При этом существуют приложения, для которых вполне достаточно файлов; приложения, для которых необходимо решать, какой уровень работы с данными во внешней памяти для них требуется, и приложения, для которых, безусловно, нужны базы данных.

Заключение

Мы начали эту лекцию с рассказа об истории систем управления внешней памятью. Развитие аппаратных и программных средств управления внешней памятью диктовалось потребностями информационных систем, для построения которых требовалась возможность надежного долговременного хранения больших объемов данных, а также обеспечение достаточно быстрого доступа к этим данным.

Системы управления файлами во внешней памяти обеспечивают минимальные потребности информационных систем, предоставляя средства распределения и структуризации дисковой памяти, именования файлов, авторизации доступа и поддержки многопользовательского режима. По мере развития технологии информационных систем их потребности возрастают, выходя за пределы возможностей, обеспечиваемых файловыми системами.

Следует особо обратить внимание на то, что и сегодня основной класс устройств внешней памяти базируется на магнитных дисках с подвижными головками. Поэтому временные соотношения, приведенные в связи с рис. 1.1, по-прежнему весьма актуальны. На этих соотношениях, главным образом, базируются оптимизационные методы, применяемые в современных системах управления данными во внешней памяти.

Далее, на примере тривиальной информационной системы были показаны ситуации, в которых возможности файловых систем явно недостаточны. Более того, попытки расширения возможностей файловой системы путем включения в приложение дополнительных программных компонентов во многих случаях не приводят к успеху. В пределе такие попытки могут привести к появлению самостоятельного программного продукта, обладающего некоторыми чертами СУБД. Однако настоящие СУБД являются настолько большими и сложными программными системами, что вероятность успешного создания «самодельной» СУБД ничтожно мала.

Еще один вывод заключается в том, что при выборе технологии построения информационной системы нужно тщательно оценивать и прогнозировать ее потенциальные потребности в средствах управления данными. Конечно, любую информационную систему можно основывать на использовании промышленной, большой и мощной СУБД. Но вполне может оказаться так, что в действительности приложение будет использовать доли процентов общих возможностей СУБД. Накладные расходы (затраты на дополнительную аппаратуру, лицензирование дорогостоящего программного продукта, увеличение общего времени выполнения операций) могут оказаться неоправданными.

Лекция 2. Введение в реляционную модель данных

В этом курсе, главным образом, обсуждаются различные аспекты реляционных баз данных. Принято считать, что реляционный подход к организации баз данных был заложен в конце 1960-х гг. Эдгаром Коддом. В последние десятилетия этот подход является наиболее распространенным (с оговоркой, что в называемых в обиходе реляционными системах баз данных, основанных на языке SQL, в действительности нарушаются некоторые важные принципы классического реляционного подхода). Достоинствами реляционного подхода принято считать следующие свойства:

- **реляционный подход основывается на небольшом числе интуитивно понятных абстракций, на основе которых возможно простое моделирование наиболее распространенных предметных областей; эти абстракции могут быть точно и формально определены;**
- **теоретическим базисом реляционного подхода к организации баз данных служит простой и мощный математический аппарат теории множеств и математической логики;**
- **реляционный подход обеспечивает возможность ненавигационного манипулирования данными без необходимости знания конкретной физической организации баз данных во внешней памяти.**

Компьютерный мир далеко не сразу признал реляционные системы. В 70-е года прошлого века, когда уже были получены почти все основные теоретические результаты и даже существовали первые прототипы реляционных СУБД, многие авторитетные специалисты отрицали возможность добиться эффективной реализации таких систем. Однако преимущества реляционного подхода и развитие методов и алгоритмов организации и управления реляционными базами данных привели к тому, что к концу 80-х годов реляционные системы заняли на мировом рынке СУБД доминирующее положение.

В этой лекции на сравнительно неформальном уровне вводятся основные понятия реляционных баз данных, а также определяется сущность реляционной модели данных. Основной целью лекции является демонстрация простоты и возможности интуитивной интерпретации этих понятий. В следующих лекциях будут приводиться более формальные определения, на которых основана теория реляционных баз данных.

Ключевые слова: Реляционный подход к организации баз данных, реляционные базы данных, реляционная модель данных, тип данных, домен, атрибут, кортеж, отношение, первичный ключ, ограничение домена, заголовок отношения, тело отношения, значение отношения, переменная отношения, степень отношения, схема реляционной базы данных, возможный ключ, ограничение целостности, первая нормальная форма, целостность сущности, внешний ключ, целостность по ссылкам, неопределенное значение (NULL).

Основные понятия реляционных баз данных

Выделим следующие основные понятия реляционных баз данных: *тип данных, домен, атрибут, кортеж, отношение, первичный ключ.*

Для начала покажем смысл этих понятий на примере отношения СЛУЖАЩИЕ, содержащего информацию о служащих некоторого предприятия (рис. 2.1).

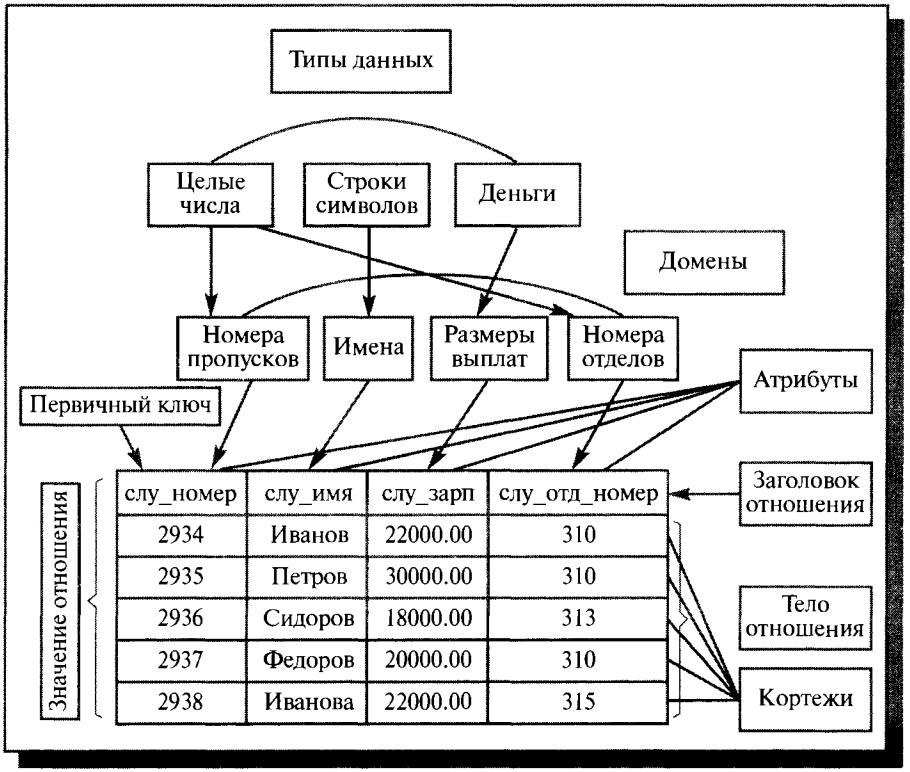


Рис. 2.1. Соотношение основных понятий реляционного подхода

Тип данных

Значения данных, хранимые в реляционной базе данных, являются типизированными, т. е. известен тип каждого хранимого значения. Понятие *типа данных* в реляционной модели данных полностью соответствует понятию типа данных в языках программирования. Напомним, что традиционное (нестрогое) определение типа данных состоит из трех основных компонентов: определение множества значений данного типа; определение набора операций, применимых к значениям типа; определение способа внешнего представления значений типа (литералов).

Обычно в современных реляционных базах данных допускается хранение символьных, числовых данных (точных и приблизительных), специализированных числовых данных (таких, как «деньги»), а также специальных «темпоральных» данных (дата, время, временной интервал). Кроме того, в реляционных системах поддерживается возможность определения пользователями собственных типов данных (более подробно мы обсудим это в лекции 19, посвященной объектным расширениям языка SQL).

В примере на рис. 2.1 мы имеем дело с данными трех типов: строки символов, целые числа и «деньги».

Домен

Понятие *домена* более специфично для баз данных, хотя и имеются аналогии с подтипами в некоторых языках программирования (более того, в своем *«Третьем манифесте»* Кристофер Дейт и Хью Дарвен вообще ликвидируют различие между доменом и типом данных). В общем виде домен определяется путем задания некоторого базового типа данных, к которому относятся элементы домена, и произвольного логического выражения, применяемого к элементу этого типа данных (*ограничения домена)*. Элемент данных является элементом домена в том и только в том случае, если вычисление этого логического выражения дает результат *истина* (для логических значений мы будем попеременно использовать обозначения *истина* и *ложь* или *true* и *false*). С каждым доменом связывается имя, уникальное среди имен всех доменов соответствующей базы данных.

Наиболее правильной интуитивной трактовкой понятия домена является его восприятие как допустимого потенциального, ограниченного подмножества значений данного типа. Например, домен ИМЕНА в нашем примере определен на базовом типе символьных строк, но в число его значений могут входить только те строки, которые могут представлять имена (в частности, для возможности представления русских имен такие строки не могут начинаться с мягкого или твердого знака и не могут быть длиннее, например, 20 символов). Если некоторый атрибут отношения определяется на некотором домене (как, например, на рис. 2.1 атрибут СЛУ_ИМЯ определяется на домене ИМЕНА), то в дальнейшем ограничение домена, в частности, играет роль ограничения целостности, накладываемого на значения этого атрибута.

Следует отметить также семантическую нагрузку понятия домена: данные считаются сравнимыми только в том случае, когда они относятся к одному домену. В нашем примере значения доменов НОМЕРА ПРОПУСКОВ и НОМЕРА ОТДЕЛОВ относятся к типу целых чисел, но не являются сравнимыми (допускать их сравнение было бы бессмысленно).

Заголовок отношения, кортеж, тело отношения, значение отношения, переменная отношения

Понятие отношения является наиболее фундаментальным в реляционном подходе к организации баз данных, поскольку *n*-арное отношение является единственной родовой структурой данных, хранящихся в реляционной базе данных. Это отражено и в общем названии подхода – термин *реляционный (relational)* происходит от *relation (отношение)*. Однако сам термин *отношение* является исключительно неточным, поскольку, говоря про любые сохраняемые данные, мы должны иметь в виду *тип* этих данных, *значения* этого типа и *переменные*, в которых сохраняются значения. Соответственно, для уточнения термина *отношение* выделяются понятия *заголовка отношения*, *значения отношения* и *переменной отношения*. Кроме того, нам потребуется вспомогательное понятие *кортежа*.

Итак, *заголовком* (или *схемой*) отношения `r` (`Hr`) называется конечное множество упорядоченных пар вида <`A`, `T`>, где `A` называется именем атрибута, а `T` обозначает имя некоторого базового типа или ранее определенного домена. По определению требуется, чтобы все имена атрибутов в заголовке отношения были различны. В примере на рис. 2.1 заголовком отношения СЛУЖАЩИЕ является множество пар {<слу_номер, номера_пропусков>, <слу_имя, имена>, <слу_зарп, размеры_выплат>, <слу_отд_номер, номера_отделов>}.

Если все атрибуты заголовка отношения определены на разных доменах, то, чтобы не плодить лишних имен, разумно использовать для именования атрибутов имена соответствующих доменов (не забывая, конечно, о том, что это всего лишь удобный способ именования, который не устраняет различия между понятиями домена и атрибута).

Кортежем `tr`, соответствующим заголовку `Hr`, называется множество упорядоченных триплетов вида <`A`, `T`, `v`>, по одному такому триплету для каждого атрибута в `Hr`. Третий элемент – `v` – триплета <`A`, `T`, `v`> должен являться допустимым значением типа данных или домена `T`. Заголовку отношения СЛУЖАЩИЕ соответствуют, например, следующие кортежи: {<слу_номер, номера_пропусков, 2934>, <слу_имя, имена, Иванов>, <слу_зарп, размеры_выплат, 22.000>, <слу_отд_номер, номера_отделов, 310>}, {<слу_номер, номера_пропусков, 2940>, <слу_имя, имена, Кузнецов>, <слу_зарп, размеры_выплат, 35.000>, <слу_отд_номер, номера_отделов, 320>}.

Телом `Br` отношения `r` называется произвольное множество кортежей `tr`. Одно из возможных тел отношения СЛУЖАЩИЕ показано на рис. 2.1. Заметим, что в общем случае, как это демонстрируют, в частности, рис. 2.1 и пример предыдущего абзаца, могут существовать такие кортежи `tr`, которые соответствуют `Hr`, но не входят в `Br`.

Значением `Vr` отношения `r` называется пара множеств `Hr` и `Br`. Одно из допустимых значений отношения СЛУЖАЩИЕ показано на рис. 2.1.

В изменчивой реляционной базе данных хранятся отношения, значения которых изменяются во времени. *Переменной* VAR*r* называется именованный контейнер, который может содержать любое допустимое значение *r*. Естественно, что при определении любой VAR*r* требуется указывать соответствующий заголовок отношения *H*.

Здесь стоит подчеркнуть, что любая принятая на практике операция обновления базы данных – INSERT (вставка кортежа в переменную отношения), DELETE (удаление кортежа из значения-отношения переменой отношения) и UPDATE (модификация кортежа значения-отношения переменной отношения) – с модельной точки зрения является операцией присваивания переменной отношения некоторого нового значения-отношения. Это совсем не означает, что перечисленные операции должны выполняться именно таким образом в СУБД: главное, чтобы результат операций соответствовал этой модельной семантике.

Заметим, что в дальнейшем в тех случаях, когда точный смысл термина понятен из контекста, мы будем использовать термин *отношение* как в смысле *значение отношения*, так и в смысле *переменная отношения*.

По определению, *степенью*, или «*арностью*», заголовка отношения, кортежа, соответствующего этому заголовку, тела отношения, значения отношения и переменной отношения является мощность заголовка отношения. Например, степень отношения СЛУЖАЩИЕ равна четырем, т. е. оно является 4-арным (*кватернарным*).

При приведенных определениях разумно считать *схемой реляционной базы данных* набор пар <имя_VAR*r*, *H*>, включающий имена и заголовки всех переменных отношения, которые определены в базе данных. Реляционная база данных – это набор пар <VAR*r*, *H*> (конечно, каждая переменная отношения в любой момент времени содержит некоторое значение-отношение, в частности, пустое).

Заметим, что в классических реляционных базах данных после определения схемы базы данных могли изменяться только значения переменных отношений. Однако теперь в большинстве реализаций допускается и изменение схемы базы данных: определение новых и изменение заголовков существующих переменных отношений. Это принято называть *эволюцией схемы базы данных*.

Первичный ключ и интуитивная интерпретация реляционных понятий

По определению, первичным ключом переменной отношения является такое подмножество* *S* множества атрибутов ее заголовка, что в лю-

* В вырожденном случае, когда заголовок переменной отношения является пустым множеством, первичный ключ этой переменной отношения состоит из пустого подмножества заголовка. Легко проверить, что этот случай не противоречит общему определению.

бое время значение первичного ключа (составное, если в состав первичного ключа входит более одного атрибута) в любом кортеже тела отношения отличается от значения первичного ключа в любом другом кортеже тела этого отношения, а никакое собственное подмножество* S этим свойством не обладает. В следующем разделе мы покажем, что существование первичного ключа у любого значения отношения является следствием одного из фундаментальных свойств отношений, а именно того свойства, что тело отношения является множеством кортежей.

Обычным житейским представлением отношения является *таблица*, *заголовком* которой является схема отношения, а *строками* – кортежи отношения-экземпляра; в этом случае имена атрибутов соответствуют именам *столбцов* данной таблицы. Поэтому иногда говорят про «столбцы таблицы», имея в виду «атрибуты отношения».

Конечно, это достаточно грубая терминология, поскольку у обычных таблиц и строки, и столбцы упорядочены, тогда как атрибуты и кортежи отношений являются элементами неупорядоченных множеств. Тем не менее, когда мы перейдем к рассмотрению практических вопросов организации реляционных баз данных и средств управления, то будем использовать эту «житейскую» терминологию. Подобной терминологии придерживаются в большинстве коммерческих реляционных СУБД. Иногда также используются термины *файл* как аналог таблицы, *запись* как аналог строки и *поле* как аналог столбца. Напомню, что этой терминологией мы пользовались в лекции 1.

Фундаментальные свойства отношений

Остановимся теперь на некоторых важных свойствах отношений, которые следуют из приведенных ранее определений.

Отсутствие кортежей-дубликатов, первичный и возможные ключи отношений

То свойство, что тело любого отношения никогда не содержит кортежей-дубликатов, следует из определения тела отношения как множества кортежей. В классической теории множеств по определению любое множество состоит из различных элементов.

Именно из этого свойства вытекает наличие у каждого значения отношения *первичного ключа* – минимального множества атрибутов, являющегося подмножеством заголовка данного отношения, составное значение которых уникально определяет кортеж отношения. Действительно, поскольку в любое время все кортежи тела любого отношения различны, у любого значения отношения свойством уникальности обладает, по

* Напомним, что S' является собственным подмножеством множества S в том и только в том случае, когда S' входит в S, но не совпадает с S (это обозначается как $S' \subset S$).

крайней мере, полный набор его атрибутов. Однако в формальном определении первичного ключа требуется обеспечение его «минимальности», т. е. в набор атрибутов первичного ключа не должны входить такие атрибуты, которые можно отбросить без ущерба для основного свойства – однозначного определения кортежа. Немного позже мы покажем, почему свойство минимальности первичного ключа является критически важным. Понятно, что если у любого отношения существует набор атрибутов, обладающий свойством уникальности, то существует и минимальный набор атрибутов, обладающий свойством уникальности.

Конечно, могут существовать значения отношения с несколькими несовпадающими минимальными наборами атрибутов, обладающими свойствами уникальности. Например, если вернуться к предположениям лекции 1 об уникальности значений атрибутов СЛУ_НОМЕР и СЛУ_ИМЯ отношения СЛУЖАЩИЕ, то для каждого значения этого отношения мы имеем два множества атрибутов, претендующих на звание первичного ключа – {СЛУ_НОМЕР} и {СЛУ_ИМЯ}. В этом случае проектировщик базы данных должен решить, какое из альтернативных множеств атрибутов назвать первичным ключом, а остальные минимальные наборы атрибутов, обладающие свойством уникальности, называются *возможными ключами*.*

Понятие первичного ключа является исключительно важным в связи с понятием целостности баз данных. Заметим, что хотя формально существование первичного ключа значения отношения является следствием того, что тело отношения – это множество, на практике первичные (и возможные) ключи *переменных отношений* появляются в результате явных указаний проектировщика отношения. Определяя переменную отношения, проектировщик моделирует часть предметной области, данные из которой будет содержать база данных. И конечно, проектировщик должен знать природу этих данных. Например, ему должно быть известно, что никакие два служащих ни в какой момент времени не могут иметь удостоверение с одним и тем же номером. Поэтому он может (и даже должен, как будет показано немного позже) явно объявить {СЛУ_НОМЕР} возможным ключом. Если на предприятии установлено, что у всех служащих должны быть разные полные имена, то проектировщик может (и опять же должен) объявить возможным ключом и {СЛУ_ИМЯ}. Затем проектировщик должен оценить, какой из возможных ключей является более надежным (свойство его уникальности никогда не будет отменено) и выбрать наиболее надежный возможный ключ в качестве первичного (в нашем случае естественным выбором был бы ключ {СЛУ_НОМЕР}, потому что решение об уникальности полных имен служащих выглядит искусственным и может быть легко отменено руководством предприятия).

Теперь поясним, почему проектировщику следует явно объявлять

* В лекции 12 мы обсудим различия между первичными и возможными ключами в языке SQL.

первичный и возможные ключи переменных отношений.* Дело в том, что в результате этого объявления СУБД получает информацию, которая в дальнейшем будет использоваться как ограничения целостности.** СУБД никогда не допустит появления в переменной отношения значения-отношения, содержащего два кортежа с одинаковым значением атрибута СЛУ_НОМЕР (определение первичного ключа для данной переменной отношения отменить нельзя). Появление двух кортежей с одинаковым значением атрибута СЛУ_ИМЯ будет также невозможно до тех пор, пока остается в силе определение {СЛУ_ИМЯ} как возможного ключа. Тем самым объявления первичного и возможных ключей дают СУБД возможность поддерживать целостность базы данных даже в случае попыток занесения в нее некорректных данных.

Наконец, вернемся к свойству минимальности первичного и возможных ключей. Как отмечалось выше, это свойство является критически важным, и важность проявляется именно при трактовке первичного и возможных ключей как ограничений целостности. В нашем примере с отношением СЛУЖАЩИЕ свойством уникальности будет обладать не только множество атрибутов {СЛУ_НОМЕР}, но и, например, множество {СЛУ_НОМЕР, СЛУ_ОТД_НОМЕР}. Но если бы мы выставили в качестве ограничения целостности требование уникальности {СЛУ_НОМЕР, СЛУ_ОТД_НОМЕР}, то СУБД гарантировала бы отсутствие кортежей с одинаковым значением атрибута СЛУ_НОМЕР не во всем значении отношения СЛУЖАЩИЕ, а только в группах кортежей с одним и тем же значением атрибута СЛУ_ОТД_НОМЕР. Понятно, что это не соответствует смыслу моделируемой предметной области.

Забегая вперед, заметим, что во многих практических реализациях реляционных СУБД допускается нарушение свойства уникальности кортежей для промежуточных отношений, порождаемых неявно при выполнении запросов. Такие отношения являются не множествами, а мультимножествами, что в ряде случаев позволяет добиться определенных преимуществ, но часто приводит к серьезным проблемам. Мы остановимся на этом подробнее при обсуждении языка SQL.

Отсутствие упорядоченности кортежей

Конечно, формально свойство отсутствия упорядоченности кортежей в значении отношения также является следствием определения тела

* Если он является сторонником классического реляционного подхода; в языке SQL допускается определение таблиц без первичного и возможных ключей.

** Кстати, заметим, что в классической реляционной модели, если при определении переменной отношения явно не указывается ее первичный ключ, то по умолчанию первичным ключом считается полный набор атрибутов заголовка отношения. Конечно, в этом случае такая переменная отношения может принимать любое значение-отношение, соответствующее заголовку, и первичный ключ не играет роль ограничения.

отношения как множества кортежей. Однако на это свойство можно взглянуть и с другой стороны. Да, то обстоятельство, что тело отношения является множеством кортежей, облегчает построение полного механизма реляционной модели данных, включая базовые средства манипулирования данными – реляционные алгебру и исчисление. Но, на мой взгляд, основная причина не в этом.

Достаточно часто у пользователей реляционных СУБД и разработчиков информационных систем вызывает раздражение тот факт, что они не могут хранить кортежи отношений на физическом уровне в нужном им порядке. И ссылки на требования реляционной теории здесь не очень уместны. Можно было бы разработать другую теорию, в которой допускались бы упорядоченные «отношения». Однако хранить упорядоченные списки кортежей в условиях интенсивно обновляемой базы данных гораздо сложнее технически, а поддержка упорядоченности влечет за собой существенные накладные расходы.

Отсутствие требования к поддержанию порядка на множестве кортежей отношения придает СУБД дополнительную гибкость при хранении баз данных во внешней памяти и при выполнении запросов к базе данных. Это не противоречит тому, что при формулировании запроса к БД, например, на языке SQL можно потребовать сортировки результирующей таблицы в соответствии со значениями некоторых столбцов. Такой результат, вообще говоря, является не отношением, а некоторым упорядоченным списком кортежей, и он может быть только окончательным результатом, к которому уже нельзя адресовать запросы.

Отсутствие упорядоченности атрибутов

Атрибуты отношений не упорядочены, поскольку по определению заголовок отношения есть множество пар <имя атрибута, имя домена>. Для ссылки на значение атрибута в кортеже отношения всегда используется имя атрибута. Легко заметить явную аналогию между заголовками отношений и структурными типами в языках программирования. Даже в языке программирования С с его практически неограниченными возможностями работы с указателями настойчиво рекомендуется обращаться к полям структур только по их именам. Если, например, на языке С определена структурная переменная

```
STRUCT {integer a; char b; integer c} d;
```

то в стандарте языка решительно не рекомендуется использовать для доступу к символьному полю *b* конструкцию `*(&d + sizeof(integer))` (взять адрес структурной переменной *d*, прибавить к нему число байтов в

целом числе и взять значение байта по полученному адресу). Это объясняется тем, что при реальном расположении в памяти полей такой структурной переменной в том порядке, как они определены, во многих компьютерах потребуется выровнять поле `c` по байту с четным адресом. Поэтому один байт просто пропадет. При расположении структурной переменной в памяти экономный компилятор (вернее, оптимизатор) переставит местами поля `b` и `c`, и указанная выше конструкция не обеспечит доступа к полю `b`. Для корректного обращения к полю `b` переменной `d` нужно использовать конструкции `d.b` или `&d->b`, т. е. явно указывать имя поля.

Аналогичными практическими соображениями оправдывается и отсутствие упорядоченности атрибутов в заголовке отношения. В этом случае СУБД сама принимает решение о том, в каком физическом порядке следует хранить значения атрибутов кортежей (хотя обычно один и тот же физический порядок поддерживается для всех кортежей каждого отношения). Кроме того, это свойство облегчает выполнение операции модификации схем существующих отношений не только путем добавления новых атрибутов, но и путем удаления существующих.

Снова забегая вперед, заметим, что в языке SQL в некоторых случаях допускается индексное указание атрибутов, причем в качестве неявного порядка атрибутов используется их порядок в линейной форме определения схемы отношения (это одна из осуждаемых особенностей языка SQL).

Атомарность значений атрибутов, первая нормальная форма отношения

Значения всех атрибутов являются атомарными (вернее, скалярными). Это следует из определения домена как потенциального множества значений скалярного типа данных, т. е. среди значений домена не могут содержаться значения с видимой структурой, в том числе множества значений (отношения). Заметим, что это не противоречит тому, что говорилось в разделе «Основные понятия реляционных баз данных» о потенциальной возможности использования при спецификации атрибутов типов данных, определяемых пользователями. Например, можно было бы добавить в схему отношения СЛУЖАЩИЕ атрибут СЛУ_ФОТО, определенный на домене (или типе данных) ФОТОГРАФИИ. Главное в атомарности значений атрибутов состоит в том, что реляционная СУБД не должна обеспечивать пользователям явной видимости внутренней структуры значения. Со всеми значениями можно обращаться только с помощью операций, определенных в соответствующем типе данных.

Принято говорить, что в реляционных базах данных допускаются только нормализованные отношения, или отношения, представленные в *первой нормальной форме*.

Пример ненормализованного отношения показан на рис. 2.2. Можно сказать, что здесь мы имеем бинарное отношение, в котором значениями атрибута ОТДЕЛЫ являются отношения. Заметим, что исходное отношение СЛУЖАЩИЕ является нормализованным вариантом отношения ОТДЕЛЫ-СЛУЖАЩИЕ. Нормализованный вариант показан на рис. 2.3.

Нормализованные отношения составляют основу классического реляционного подхода к организации баз данных. Они обладают некоторыми ограничениями (не всякую информацию удобно представлять в виде плоских таблиц)*, но существенно упрощают манипулирование данными. Рассмотрим, например, два идентичных оператора занесения кортежа:

- зачислить служащего Кузнецова (пропуск номер 3000, зарплата 25000.00) в отдел номер 320;

НОМЕР_ОТДЕЛА	ОТДЕЛ		
	СЛУ_НОМЕР	СЛУ_ИМЯ	СЛУ_ЗАРП
310	2934	Иванов	22000.00
	2935	Петров	30000.00
	2937	Федоров	20000.00
313	2936	Сидоров	18000.00
315	2938	Иванова	22000.00

Рис. 2.2. Ненормализованное отношение ОТДЕЛЫ-СЛУЖАЩИЕ

СЛУ_НОМЕР	СЛУ_ИМЯ	СЛУ_ЗАРП	СЛУ_ОТД_НОМЕР
2934	Иванов	22000.00	310
2935	Петров	30000.00	310
2936	Сидоров	18000.00	313
2937	Федоров	20000.00	310
2938	Иванова	22000.00	315

Рис. 2.3. Отношение СЛУЖАЩИЕ: нормализованный вариант отношения ОТДЕЛЫ-СЛУЖАЩИЕ

* Эти ограничения все более ослабляются в последовательности стандартов языка SQL.

- зачислить служащего Кузнецова (пропуск номер 3000, зарплата 25000.00) в отдел номер 310.

Если информация о служащих представлена в виде отношения СЛУЖАЩИЕ, оба оператора будут выполняться одинаково (вставить кортеж в отношение СЛУЖАЩИЕ). Если же работать с ненормализованным отношением ОТДЕЛЫ-СЛУЖАЩИЕ, то первый оператор приведет к простой вставке кортежа, а второй – к добавлению кортежа в значение-отношение атрибута ОТДЕЛ кортежа с первичным ключом 310.

При работе с ненормализованными отношениями аналогичные затруднения возникают при выполнении операций удаления и модификации кортежей.

Реляционная модель данных

Когда в предыдущих разделах мы говорили об основных понятиях реляционных баз данных, мы не опирались на какую-либо конкретную реализацию. Эти рассуждения в равной степени относятся к любой системе, при построении которой использовался реляционный подход.

Другими словами, мы использовали понятия так называемой реляционной модели данных. Модель данных (в контексте области баз данных) описывает некий набор родовых понятий и признаков, которыми должны обладать все конкретные СУБД и управляемые ими базы данных, если они основываются на этой модели. Наличие модели данных позволяет сравнивать конкретные реализации, используя один общий язык.

Хотя понятие модели данных является общим, и можно говорить об иерархической, сетевой, семантической и других моделях данных, нужно отметить, что в области баз данных это понятие было введено Эдгаром Коддом применительно к реляционным системам и наиболее эффективно используется именно в данном контексте. Попытки прямолинейного применения аналогичных моделей к дореляционным организациям показывают, что реляционная модель слишком «велика», а для постреляционных организаций она оказывается «мала».

Общая характеристика

Хотя понятие реляционной модели данных первым ввел основоположник реляционного подхода Эдгар Кодд, наиболее распространенная трактовка реляционной модели данных, по-видимому, принадлежит известному популяризатору идей Кодда Кристоферу Дейту, который воспроизводит ее (с различными уточнениями) практически во всех своих книгах (см., например, К. Дейт. Введение в системы баз данных. 6-е изд., М.; СПб.: Вильямс.– 2000). Согласно трактовке Дейта, реляционная мо-

дель состоит из трех частей, описывающих разные аспекты реляционного подхода: структурной части, манипуляционной части и целостной части.

В структурной части модели фиксируется, что единственной родовой* структурой данных, используемой в реляционных БД, является нормализованное n-арное отношение. Определяются понятия доменов, атрибутов, кортежей, заголовка, тела и переменной отношения. По сути дела, в двух предыдущих разделах этой лекции мы рассматривали именно понятия и свойства структурной составляющей реляционной модели.

В манипуляционной части модели определяются два фундаментальных механизма манипулирования реляционными БД – реляционная алгебра и реляционное исчисление. Первый механизм базируется в основном на классической теории множеств (с некоторыми уточнениями и добавлениями), а второй – на классическом логическом аппарате исчисления предикатов первого порядка. Мы рассмотрим эти механизмы более подробно в следующих лекциях, а пока лишь заметим, что основной функцией манипуляционной части реляционной модели является обеспечение меры реляционности любого конкретного языка реляционных БД: язык называется реляционным, если он обладает не меньшей выразительностью и мощностью, чем реляционная алгебра или реляционное исчисление.

Целостность сущности и ссылок

Наконец, в целостной части реляционной модели данных фиксируются два базовых требования целостности, которые должны поддерживаться в любой реляционной СУБД. Первое требование называется *требованием целостности сущности (entity integrity)*. Объекту или сущности реального мира в реляционных БД соответствуют кортежи отношений. Конкретно требование состоит в том, что любой кортеж любого значения-отношения любой переменной отношения должен быть отличим от любого другого кортежа этого значения отношения по составным значе-

* Уже второй раз в этой лекции утверждается, что нормализованное n-арное отношение является единственной родовой структурой данных, используемой в реляционных БД. Пришло время пояснить, что мы имеем в виду под термином *родовая структура*. В языках программирования с развитыми системами типов обычно имеются конструкции, называемые *родовыми типами*, *параметризуемыми типами*, *конструкторами типов*, *генераторами типов* и т.д., позволяющие породить конкретный тип данных на основе его абстрактной (обычно, предопределенной) спецификации. Особенность таких типов состоит в том, что и основные операции конкретного типа определяются на уровне этой абстрактной спецификации. Одним из наиболее известных примеров является *тип множества*, например, в языке Pascal. В случае реляционной модели данных мы не говорим явно, что отношение является родовым типом, но, по существу, это именно так. Операции реляционной алгебры определяются на уровне абстрактного отношения и применимы к любым значениям-отношениям с конкретными заголовками.

ниям заранее определенного множества атрибутов переменной отношения, т. е., другими словами, любая переменная отношения должно обладать первичным ключом. Как мы видели в предыдущем разделе, это требование автоматически удовлетворяется, если в системе не нарушаются базовые свойства отношений.

На самом деле, требование целостности сущности полностью звучит следующим образом: *у любой переменной отношения должен существовать первичный ключ, и никакое значение первичного ключа в кортежах значения-отношения переменной отношения не должно содержать неопределенных значений*. Чтобы эта формулировка была полностью понятна, мы должны хотя бы кратко обсудить понятие *неопределенного значения* (NULL).

Конечно, теоретически любой кортеж, заносимый в сохраняемое отношение, должен содержать все характеристики моделируемой им сущности реального мира, которые мы хотим сохранить в базе данных. Однако на практике не все эти характеристики могут быть известны к тому моменту, когда требуется зафиксировать сущность в базе данных. Простым примером может быть процедура принятия на работу человека, размер заработной платы которого еще не определен. В этом случае служащий отдела кадров, который заносит в отношение СЛУЖАЩИЕ кортеж, описывающий нового служащего, просто не может обеспечить значение атрибута СЛУ_ЗАРП (любое значение домена РАЗМЕРЫ_ВЫПЛАТ будет неверно характеризовать зарплату нового служащего).

Эдгар Кодд предложил использовать в таких случаях неопределенные значения. Неопределенное значение не принадлежит никакому типу данных и может присутствовать среди значений любого атрибута, определенного на любом типе данных (если это явно не запрещено при определении атрибута). Если *a* – это значение некоторого типа данных или NULL, *op* – любая двуместная «арифметическая» операция этого типа данных (например, +), а *lop* – операция сравнения значений этого типа (например, =), то по определению:

```
a op NULL = NULL
NULL op a = NULL
a lop NULL = unknown
NULL lop a = unknown
```

Здесь *unknown* – это третье значение логического, или булевского, типа, обладающее следующими свойствами:

```
NOT unknown = unknown
true AND unknown = unknown
true OR unknown = true
```

```
false AND unknown = false
false OR unknown = unknown
```

(напомним, что операции AND и OR являются коммутативными)*. В данной лекции нам достаточно приведенного краткого введения в неопределенные значения, но в следующих лекциях мы будем неоднократно возвращаться к этой теме.

Так вот, первое из требований — требование целостности сущности — означает, что первичный ключ должен полностью идентифицировать каждую сущность, а поэтому в составе любого значения первичного ключа не допускается наличие неопределенных значений. (В классической реляционной модели это требование распространяется и на возможные ключи; как будет показано в следующих лекциях, в SQL-ориентированных СУБД такое требование для возможных ключей не поддерживается.)

Второе требование, которое называется *требованием целостности по ссылкам (referential integrity)*, является более сложным. Очевидно, что при соблюдении нормализованности отношений сложные сущности реального мира представляются в реляционной БД в виде нескольких кортежей нескольких отношений. Например, представим, что требуется представить в реляционной базе данных сущность ОТДЕЛ с атрибутами ОТД_НОМЕР (номер отдела), ОТД_РАЗМ (количество служащих) и ОТД_СЛУ (множество служащих отдела). Для каждого служащего нужно хранить СЛУ_НОМЕР (номер служащего), СЛУ_ИМЯ (имя служащего) и СЛУ_ЗАРП (заработная плата служащего). Как мы увидим в лекции 7, при правильном проектировании соответствующей БД в ней появятся два отношения: ОТДЕЛЫ {ОТД_НОМЕР, ОТД_РАЗМ} (первичный ключ – {ОТД_НОМЕР}) и СЛУЖАЩИЕ {СЛУ_НОМЕР, СЛУ_ИМЯ, СЛУ_ЗАРП, СЛУ_ОТД_НОМ} (первичный ключ – {СЛУ_НОМЕР}).

Как видно, атрибут СЛУ_ОТД_НОМ вводится в отношение СЛУЖАЩИЕ не потому, что номер отдела является собственным свойством служащего, а лишь для того, чтобы иметь возможность при необходимости восстановить полную сущность ОТДЕЛ. Значение атрибута СЛУ_ОТД_НОМ в любом кортеже отношения СЛУЖАЩИЕ должно соответствовать значению атрибута ОТД_НОМ в некотором кортеже отношения ОТДЕЛЫ. Атрибут такого рода (возможно, составной) называется *внешним ключом (foreign key)*, поскольку его значения однозначно характеризуют сущности, представленные кортежами некоторого другого отношения (т. е. задают значения их пер-

* Как показывает опыт автора, не всегда и не все студенты помнят базовые логические операции. Для гарантии приведем таблицы истинности операций AND (& – конъюнкция), OR (∨ – дизъюнкция) и NOT (¬ – отрицание):

AND	*true*	*false*		OR	*true*	*false*		NOT	*true*	*false*
true	*true*	*false*		*true*	*true*	*true*			*true*	*false*
false	*false*	*false*		*true*	*false*					

вичного ключа). Конечно, внешний ключ может быть составным, т. е. состоять из нескольких атрибутов. Говорят, что отношение, в котором определен внешний ключ, ссылается на соответствующее отношение, в котором такой же атрибут является первичным ключом.

Требование целостности по ссылкам, или требование целостности внешнего ключа, состоит в том, что для каждого значения внешнего ключа, появляющегося в кортеже значения-отношения ссылающейся переменной отношения, либо в значении-отношении переменной отношения, на которую указывает ссылка, должен найтись кортеж с таким же значением первичного ключа, либо значение внешнего ключа должно быть полностью неопределенным (т. е. ни на что не указывать).* Для нашего примера это означает, что если для служащего указан номер отдела, то этот отдел должен существовать.

Заметим, что, как и первичный ключ, внешний ключ должен специфицироваться при определении переменной отношения и представляет собой ограничение на допустимые значения-отношения этой переменной. Другими словами, определение внешнего ключа представляет собой определение ограничения целостности базы данных.

Ограничения целостности сущности и по ссылкам должны поддерживаться СУБД. Для соблюдения целостности сущности достаточно гарантировать отсутствие в любой переменной отношения значений-отношений, содержащих кортежи с одним и тем же значением первичного ключа (и запрещать вхождение в значение первичного ключа неопределенных значений). С целостностью по ссылкам дело обстоит несколько сложнее.

Понятно, что при обновлении ссылающегося отношения (вставке новых кортежей или модификации значения внешнего ключа в существующих кортежах) достаточно следить за тем, чтобы не появлялись некорректные значения внешнего ключа. Но как быть при удалении кортежа из отношения, на которое ведет ссылка?

Здесь существуют три подхода, каждый из которых поддерживает целостность по ссылкам. Первый подход заключается в том, что вообще запрещается производить удаление кортежа, для которого существуют ссылки (т. е. сначала нужно либо удалить ссылающиеся кортежи, либо соответствующим образом изменить значения их внешнего ключа). При втором подходе при удалении кортежа, на который имеются ссылки, во всех ссылающихся кортежах значение внешнего ключа автоматически становится полностью неопределенным. Наконец, третий подход (каскадное удаление) состоит в том, что при удалении кортежа из отношения,

* В языке SQL допускается несколько вариантов определения внешнего ключа, из которых только один полностью соответствует классическому подходу. Более подробно мы обсудим это в следующих лекциях.

на которое ведет ссылка, из ссылающегося отношения автоматически удаляются все ссылающиеся кортежи.

В развитых реляционных СУБД обычно можно выбрать способ поддержания целостности по ссылкам для каждого случая определения внешнего ключа. Конечно, для принятия такого решения необходимо анализировать требования конкретной прикладной области.

Заключение

Скорее всего, потенциальные читатели этого курса работают или будут работать с какой-либо SQL-ориентированной СУБД. Любая компания, производящая подобные СУБД, называет их реляционными системами. Очень важно отчетливо понимать, какие свойства таких систем действительно являются реляционными, а что в них не вполне соответствует исходным, ясным и строгим идеям реляционного подхода и даже противоречит им. Это поможет более правильно организовывать базы данных и строить приложения в среде SQL-ориентированной СУБД.

В нескольких лекциях данного курса достаточно подробно обсуждаются возможности текущих стандартов языка SQL:1999 и SQL:2003. Но сначала читателям предлагается материал, который представляет реляционный подход в чистом виде. В данной лекции вводится понятийная основа реляционного подхода; определяются основные термины; исследуются фундаментальные следствия базовых определений. Рассматриваемая реляционная модель данных предназначена, прежде всего, для оценки соответствия различных реализаций СУБД общему реляционному подходу.

Лекция 3. Базисные средства манипулирования реляционными данными: реляционная алгебра Кодда

В предыдущей лекции упоминались три составляющих реляционной модели данных. Две из них – структурную и целостную части – мы рассмотрели более или менее подробно, а манипуляционной части реляционной модели данных посвящается эта и следующие две лекции. Мы уделяем данной теме такое большое внимание, поскольку понимание формальных механизмов манипулирования реляционными данными исключительно важно для понимания технологии баз данных вообще. В этой лекции после небольшого введения будет рассмотрен вариант реляционной алгебры, предложенный Кристофером Дейтом около 15 лет тому назад. Мне этот вариант алгебры кажется наиболее понятным, хотя предлагаемый набор операций несколько избыточен. В следующей лекции мы обсудим новый «минимальный» вариант алгебры, предложенный Дейтом и Дарвеном в конце 1990-х гг. Возможно, новая алгебра не очень практична, но зато красива и элегантна. После этого в лекции 5 мы перейдем к реляционному исчислению, достаточно подробно рассмотрим один из вариантов реляционного исчисления кортежей и кратко обсудим особенности исчисления доменов.

Ключевые слова: механизмы манипулирования реляционными данными, реляционная алгебра, реляционно полный язык баз данных, реляционная алгебра Кодда, операция объединения отношений, операция пересечения отношений, операция взятия разности отношений, операция взятия расширенного декартова произведения отношений, операция ограничения отношения, операция проекции отношения, операция соединения отношений, операция деления отношений, операция переименования атрибутов, операция присваивания отношений, замкнутость реляционной алгебры, совместимость отношений по объединению, совместимость отношений относительно операции расширенного декартова произведения, эквисоединение, естественное соединение.

Введение

Как мы отмечали в предыдущей лекции, в манипуляционной составляющей реляционной модели данных определяются два базовых механизма манипулирования реляционными данными – основанная на теории множеств реляционная алгебра и базирующееся на математической логике (точнее, на исчислении предикатов первого порядка) реляционное исчисление. В свою очередь, обычно выделяются два вида реляционного исчисления – исчисление кортежей и исчисление доменов.

Все эти механизмы обладают одним важным свойством: они замкнуты относительно понятия отношения. Это означает, что выражения реляционной алгебры и формулы реляционного исчисления определяются над отношениями реляционных БД и результатом их «вычисления» также являются отношения (конечно, здесь имеются в виду значения-отношения). В результате любое выражение или формула могут интерпретироваться как отношения, что позволяет использовать их в других выражениях или формулах.

Как мы увидим, алгебра и исчисление обладают большой выразительной мощностью: очень сложные запросы к базе данных могут быть выражены с помощью одного выражения реляционной алгебры или одной формулы реляционного исчисления. Именно по этой причине такие механизмы включены в реляционную модель данных. Конкретный язык манипулирования реляционными БД называется реляционно-полным, если любой запрос, формулируемый с помощью одного выражения реляционной алгебры или одной формулы реляционного исчисления, может быть сформулирован с помощью одного оператора этого языка.

Известно (и мы не будем это доказывать), что механизмы реляционной алгебры и реляционного исчисления эквивалентны, т. е. для любого допустимого выражения реляционной алгебры можно построить эквивалентную (т. е. производящую такой же результат) формулу реляционного исчисления и наоборот. Почему же в реляционной модели данных присутствуют оба эти механизма?

Дело в том, что они различаются уровнем процедурности. Выражения реляционной алгебры строятся на основе алгебраических операций (высокого уровня), и подобно тому, как интерпретируются арифметические и логические выражения, выражение реляционной алгебры также имеет процедурную интерпретацию. Другими словами, запрос, представленный на языке реляционной алгебры, может быть вычислен на основе выполнения элементарных алгебраических операций с учетом их приоритетности и возможного наличия скобок. Для формулы реляционного исчисления однозначная вычислительная интерпретация, вообще говоря, отсутствует. Формула только ставит условия, которым должны удовлетворять кортежи результирующего отношения. Поэтому языки реляционного исчисления являются в большей степени непроцедурными, или декларативными.

Поскольку механизмы реляционной алгебры и реляционного исчисления эквивалентны, в конкретной ситуации для проверки степени реляционности некоторого языка БД можно пользоваться любым из этих механизмов.

Заметим, что крайне редко алгебра или исчисление принимается в качестве полной основы какого-либо языка БД. Обычно (например, в случае языка SQL) язык основывается на некоторой смеси алгебраичес-

ких и логических конструкций. Тем не менее знание алгебраических и логических основ языков баз данных часто применяется на практике.

Для экономии времени и места мы не будем вводить какие-либо строгие синтаксические конструкции, а в основном ограничимся рассмотрением материала на содержательном уровне.

Обзор реляционной алгебры Кодда

Основная идея реляционной алгебры состоит в том, что коль скоро отношения являются множествами, средства манипулирования отношениями могут базироваться на традиционных теоретико-множественных операциях, дополненных некоторыми специальными операциями, специфичными для реляционных баз данных.

Существует много подходов к определению реляционной алгебры, которые различаются наборами операций и способами их интерпретации, но, в принципе, являются более или менее равносильными. В данном разделе мы опишем немного расширенный начальный вариант алгебры, который был предложен Коддом (будем называть ее «алгеброй Кодда»). В этом варианте набор основных алгебраических операций состоит из восьми операций, которые делятся на два класса – теоретико-множественные операции и специальные реляционные операции. В состав теоретико-множественных операций входят операции:

- объединения отношений;
- пересечения отношений;
- взятия разности отношений;
- взятия декартова произведения отношений.

Специальные реляционные операции включают:

- ограничение отношения;
- проекцию отношения;
- соединение отношений;
- деление отношений.

Кроме того, в состав алгебры включается операция присваивания, позволяющая сохранить в базе данных результаты вычисления алгебраических выражений, и операция переименования атрибутов, дающая возможность корректно сформировать заголовок (схему) результирующего отношения.

Общая интерпретация реляционных операций

Если не вдаваться в некоторые тонкости, которые мы рассмотрим в следующих разделах, то почти для всех операций предложенного выше набора имеется очевидная и простая интерпретация.

- При выполнении операции *объединения* (UNION) двух отношений с одинаковыми заголовками производится отношение, включающее все кортежи, которые входят хотя бы в одно из отношений-операндов.
- Операция *пересечения* (INTERSECT) двух отношений с одинаковыми заголовками производит отношение, включающее все кортежи, которые входят в оба отношения-операнда.
- Отношение, являющееся *разностью* (MINUS) двух отношений с одинаковыми заголовками, включает все кортежи, входящие в отношение-первый операнд, такие, что ни один из них не входит в отношение, которое является вторым операндом.
- При выполнении *декартова произведения* (TIMES) двух отношений, пересечение заголовков которых пусто, производится отношение, кортежи которого производятся путем объединения кортежей первого и второго операндов.
- Результатом *ограничения* (WHERE) отношения по некоторому условию является отношение, включающее кортежи отношения-операнда, удовлетворяющее этому условию.
- При выполнении *проекции* (PROJECT) отношения на заданное подмножество множества его атрибутов производится отношение, кортежи которого являются соответствующими подмножествами кортежей отношения-операнда.
- При *соединении* (JOIN) двух отношений по некоторому условию образуется результирующее отношение, кортежи которого производятся путем объединения кортежей первого и второго отношений и удовлетворяют этому условию.
- У операции *реляционного деления* (DIVIDE BY) два операнда – бинарное и унарное отношения. Результирующее отношение состоит из унарных кортежей, включающих значения первого атрибута кортежей первого операнда таких, что множество значений второго атрибута (при фиксированном значении первого атрибута) включает множество значений второго операнда.
- Операция *переименования* (RENAME) производит отношение, тело которого совпадает с телом операнда, но имена атрибутов изменены.
- Операция *присваивания* (:=) позволяет сохранить результат вычисления реляционного выражения в существующей переменной отношения БД.

Поскольку результатом любой реляционной операции (кроме операции присваивания, которая не вырабатывает значения) является некое отношение, можно образовывать реляционные выражения, в которых вместо отношения-операнда некоторой реляционной операции находится вложенное реляционное выражение. В построении реляционного выражения могут участвовать все реляционные операции, кроме операции

присваивания. Вычислительная интерпретация реляционного выражения диктуется установленными приоритетами операций:

```
RENAME ⩾ WHERE = PROJECT ⩾ TIMES = JOIN = INTERSECT =
DIVIDE BY ⩾ UNION = MINUS
```

В другой форме приоритеты операций показаны на рис. 3.1. Вычисление выражения производится слева направо с учетом приоритетов операций и скобок.

Операция	Приоритет
RENAME	4
WHERE	3
PROJECT	3
TIMES	2
JOIN	2
INTERSECT	2
DIVIDE BY	2
UNION	1
MINUS	1

Рис. 3.1. Таблица приоритетов операций традиционной реляционной алгебры

Замкнутость реляционной алгебры и операция переименования

Как мы отмечали в предыдущей лекции, каждое значение-отношение характеризуется заголовком (или схемой) и телом (или множеством кортежей). Поэтому, если нам действительно нужна алгебра, операции которой замкнуты относительно понятия отношения, то каждая операция должна производить отношение в полном смысле, т. е. оно должно обладать и телом, и заголовком. Только в этом случае можно будет строить вложенные выражения.

Заголовок отношения представляет собой множество пар <имя-атрибута, имя-домена>. Если посмотреть на общий обзор реляционных операций, приведенный в предыдущем подразделе, то видно, что домены атрибутов результирующего отношения однозначно определяются доменами отно-

шений-операндов. Однако с именами атрибутов результата не всегда все так просто.

Например, представим себе, что у отношений-операндов операции декартова произведения имеются одноименные атрибуты с одинаковыми доменами. Каким был бы заголовок результирующего отношения? Поскольку это множество, в нем не должны содержаться одинаковые элементы. Но и потерять атрибут в результате недопустимо. А это значит, что в таком случае вообще невозможно корректно выполнить операцию декартова произведения.

Аналогичные проблемы могут возникать и в случаях других двуместных операций. Для разрешения проблем в число операций реляционной алгебры вводится операция переименования. Ее следует применять в том случае, когда возникает конфликт именования атрибутов в отношениях-операндах одной реляционной операции. Тогда к одному из операндов сначала применяется операция переименования, а затем основная операция выполняется уже без всяких проблем. Более строго мы определим операцию переименования в следующей лекции, а пока лишь заметим, что результатом этой операции является отношение, совпадающее во всем с отношением-операндом, кроме того, что имя указанного атрибута изменено на заданное имя.

В дальнейшем изложении мы будем предполагать применение операции переименования во всех конфликтных ситуациях.

Особенности теоретико-множественных операций реляционной алгебры

Хотя в основе теоретико-множественной части реляционной алгебры Кодда лежит классическая теория множеств, соответствующие операции реляционной алгебры обладают некоторыми особенностями.

Операции объединения, пересечения, взятия разности. Совместимость по объединению

Начнем с операции объединения отношений (все, что будет сказано по поводу объединения, верно и для операций пересечения и взятия разности отношений). Смысл операции объединения в реляционной алгебре в целом остается теоретико-множественным. Напомним, что в теории множеств:

- результатом объединения двух множеств $A\{a\}$ и $B\{b\}$ является такое множество $C\{c\}$, что для каждого c либо существует такой элемент a, принадлежащий множеству A, что $c=a$, либо существует такой элемент b, принадлежащий множеству B, что $c=b$;

- пересечением множеств A и B является такое множество C $\{c\}$, что для любого c существуют такие элементы a, принадлежащий множеству A, и b, принадлежащий множеству B, что $c=a=b$;
- разностью множеств A и B является такое множество C $\{c\}$, что для любого c существует такой элемент a, принадлежащий множеству A, что $c=a$, и не существует такой элемент b, принадлежащий B, что $c=b$.

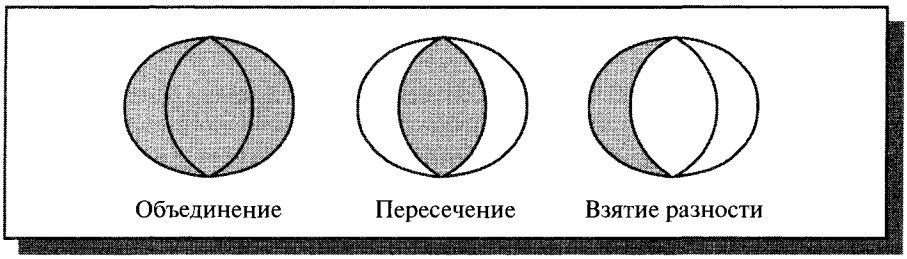


Рис. 3.2. Иллюстрация результатов теоретико-множественных операций

Но если в теории множеств операция объединения осмысленна для любых двух множеств-операндов, то в случае реляционной алгебры результатом операции объединения должно являться отношение. Если в реляционной алгебре допустить возможность теоретико-множественного объединения двух произвольных отношений (с разными заголовками), то, конечно, результатом операции будет множество, но множество разнотипных кортежей, т. е. не отношение. Если исходить из требования замкнутости реляционной алгебры относительно понятия отношения, то такая операция объединения является бессмысленной.

Эти соображения подводят к понятию *совместимости отношений по объединению: два отношения совместимы по объединению в том и только в том случае, когда обладают одинаковыми заголовками*. В развернутой форме это означает, что в заголовках обоих отношений содержится один и тот же набор имен атрибутов, и одноименные атрибуты определены на одном и том же домене (эта развернутая формулировка, вообще говоря, является излишней, но она пригодится нам в следующем абзаце).

Если два отношения совместимы по объединению, то при обычном выполнении над ними операций объединения, пересечения и взятия разности результатом операции является отношение с корректно определенным заголовком, совпадающим с заголовком каждого из отношений-операндов. Напомним, что если два отношения «почти» совместимы по объединению, т. е. совместимы во всем, кроме имен атрибутов, то до выполнения операции типа объединения эти отношения можно сделать полностью совместимыми по объединению путем применения операции переименования.

Для иллюстрации операций объединения, пересечения и взятия разности предположим, что в базе данных имеются два отношения СЛУЖАЩИЕ_В_ПРОЕКТЕ_1 и СЛУЖАЩИЕ_В_ПРОЕКТЕ_2 с одинаковыми схемами {СЛУ_НОМЕР, СЛУ_ИМЯ, СЛУ_ЗАРП, СЛУ_ОТД_НОМЕР} (имена доменов опущены по причине очевидности). Каждое из отношений содержит данные о служащих, участвующих в соответствующем проекте. На рис. 3.3 показано примерное наполнение каждого из двух отношений (некоторые служащие участвуют в обоих проектах).

СЛУЖАЩИЕ_В_ПРОЕКТЕ_1

СЛУ_НОМЕР	СЛУ_ИМЯ	СЛУ_ЗАРП	СЛУ_ОТД_НОМЕР
2934	Иванов	22000.00	310
2935	Петров	30000.00	310
2936	Сидоров	18000.00	313
2937	Федоров	20000.00	310
2938	Иванова	22000.00	315

СЛУЖАЩИЕ_В_ПРОЕКТЕ_2

СЛУ_НОМЕР	СЛУ_ИМЯ	СЛУ_ЗАРП	СЛУ_ОТД_НОМЕР
2934	Иванов	22000.00	310
2935	Петров	30000.00	310
2939	Сидоренко	18000.00	313
2940	Федоренко	20000.00	310
2941	Иваненко	22000.00	315

Рис. 3.3. Примерное наполнение отношений СЛУЖАЩИЕ _В_ПРОЕКТЕ_1 и СЛУЖАЩИЕ _В_ПРОЕКТЕ_2

Тогда выполнение операции СЛУЖАЩИЕ_В_ПРОЕКТЕ_1 UNION СЛУЖАЩИЕ_В_ПРОЕКТЕ_2 позволит получить информацию обо всех служащих, участвующих в обоих проектах. Выполнение операции СЛУЖАЩИЕ_В_ПРОЕКТЕ_1 INTERSECT СЛУЖАЩИЕ_В_ПРОЕКТЕ_2 позволит получить данные о служащих, которые одновременно участвуют в двух проектах. Наконец, операция СЛУЖАЩИЕ_В_ПРОЕКТЕ_1 MINUS СЛУЖАЩИЕ_В_ПРОЕКТЕ_2 выработает отношение, содержащее кортежи служащих, которые участвуют только в первом проекте. Результаты этих операций показаны на рис. 3.4.

СЛУЖАЩИЕ_В_ПРОЕКТЕ_1 UNION **СЛУЖАЩИЕ_В_ПРОЕКТЕ_2**

СЛУ_НОМЕР	СЛУ_ИМЯ	СЛУ_ЗАРП	СЛУ_ОТД_НОМЕР
2934	Иванов	22000.00	310
2935	Петров	30000.00	310
2939	Сидоренко	18000.00	313
2940	Федоренко	20000.00	310
2941	Иваненко	22000.00	315
2936	Сидоров	18000.00	313
2937	Федоров	20000.00	310
2938	Иванова	22000.00	315

СЛУЖАЩИЕ_В_ПРОЕКТЕ_1 INTERSECT **СЛУЖАЩИЕ_В_ПРОЕКТЕ_2**

СЛУ_НОМЕР	СЛУ_ИМЯ	СЛУ_ЗАРП	СЛУ_ОТД_НОМЕР
2934	Иванов	22000.00	310
2935	Петров	30000.00	310

СЛУЖАЩИЕ_В_ПРОЕКТЕ_1 MINUS **СЛУЖАЩИЕ_В_ПРОЕКТЕ_2**

СЛУ_НОМЕР	СЛУ_ИМЯ	СЛУ_ЗАРП	СЛУ_ОТД_НОМЕР
2936	Сидоров	18000.00	313
2937	Федоров	20000.00	310
2938	Иванова	22000.00	315

Рис. 3.4. Результаты выполнения операций UNION, INTERSECT и MINUS

Заметим, что включение в состав операций реляционной алгебры трех операций объединения, пересечения и взятия разности является, очевидно, избыточным, поскольку, например, операция пересечения выражается через операцию взятия разности*. Тем не менее Кодд в свое время решил включить все три операции, исходя из интуитивных потребностей далекого от математики потенциального пользователя системы реляционных БД.

* Легко убедиться, что *A* INTERSECT *B* = *A* MINUS (*A* MINUS *B*) = *B* MINUS (*B* MINUS *A*).

Операция расширенного декартова произведения и совместимость отношений относительно этой операции

Другие проблемы связаны с операцией взятия декартова произведения двух отношений. В теории множеств декартово произведение может быть получено для любых двух множеств, и элементами результирующего множества являются пары, составленные из элементов первого и второго множеств. Если говорить более точно, декартовым произведением множеств *A* {*a*} и *B* {*b*} является такое множество пар *C* {<*c1*, *c2*>}, что для каждого элемента <*c1*, *c2*> множества *C* существуют такой элемент *a* множества *A*, что *c1=a*, и такой элемент *b* множества *B*, что *c2=b*.

Поскольку отношения являются множествами, для любых двух отношений возможно получение прямого произведения. Но результат не будет отношением! Элементами результата будут не кортежи, а пары кортежей.

Поэтому в реляционной алгебре используется специализированная форма операции взятия декартова произведения – расширенное декартово произведение отношений. При взятии расширенного декартова произведения двух отношений элементом результирующего отношения является кортеж, который представляет собой объединение одного кортежа первого отношения и одного кортежа второго отношения.

Приведем более точное определение операции расширенного декартова произведения. Пусть имеются два отношения *R1* {a_1, a_2, …, a_n} и *R2* {b_1, b_2, …, b_m}. Тогда результатом операции *R1* TIMES *R2* является отношение *R* {a_1, a_2, …, a_n, b_1, b_2, …, b_m}, тело которого является множеством кортежей вида {r_{a1}, r_{a2}, …, r_{an}, r_{b1}, r_{b2}, …, r_{bm}} таких, что {r_{a1}, r_{a2}, …, r_{an}} входит в тело *R1*, а {r_{b1}, r_{b2}, …, r_{bm}} входит в тело *R2*.

Но теперь возникает вторая проблема – как получить корректно сформированный заголовок отношения-результата? Поскольку схема результирующего отношения является объединением схем отношений-операндов, то очевидной проблемой может быть именование атрибутов результирующего отношения, если отношения-операнды обладают одноименными атрибутами.

Эти соображения приводят к введению понятия *совместимости по взятию расширенного декартова произведения. Два отношения совместимы по взятию расширенного декартова произведения в том и только в том случае, если пересечение множеств имен атрибутов, взятых из их схем отношений, пусто.* Любые два отношения всегда могут стать совместимыми по взятию декартова произведения, если применить операцию переименования к одному из этих отношений.

Для наглядности предположим, что в придачу к введенным ранее отношениям СЛУЖАЩИЕ_В_ПРОЕКТЕ_1 и СЛУЖАЩИЕ_В_ПРОЕКТЕ_2 в базе данных содержится еще и отношение ПРОЕКТЫ со схемой {ПРОЕКТ_НАЗВ,

ПРОЕКТ_РУК} (имена доменов снова опущены) и телом, показанным на рис. 3.5. На этом же рисунке показан результат операции СЛУЖАЩИЕ_В_ПРОЕКТЕ_1 TIMES ПРОЕКТЫ.

Следует заметить, что операция взятия декартова произведения не является слишком осмысленной на практике. Во-первых, мощность тела ее результата очень велика даже при допустимых мощностях операн-

ПРОЕКТЫ

ПРОЕКТ_НАЗВ	ПРОЕКТ_РУК
ПРОЕКТ 1	Иванов
ПРОЕКТ 2	Иваненко

СЛУЖАЩИЕ_В_ПРОЕКТЕ_1 TIMES **ПРОЕКТЫ**

СЛУ_НОМЕР	СЛУ_ИМЯ	СЛУ_ЗАРП	СЛУ_ОТД_НОМЕР	ПРОЕКТ_НАЗВ	ПРОЕКТ_РУК
2934	Иванов	22000.00	310	ПРОЕКТ 1	Иванов
2935	Петров	30000.00	310	ПРОЕКТ 1	Иванов
2936	Сидоров	18000.00	313	ПРОЕКТ 1	Иванов
2937	Федоров	20000.00	310	ПРОЕКТ 1	Иванов
2938	Иванова	22000.00	315	ПРОЕКТ 1	Иванов
2934	Иванов	22000.00	310	ПРОЕКТ 2	Иваненко
2935	Петров	30000.00	310	ПРОЕКТ 2	Иваненко
2936	Сидоров	18000.00	313	ПРОЕКТ 2	Иваненко
2937	Федоров	20000.00	310	ПРОЕКТ 2	Иваненко
2938	Иванова	22000.00	315	ПРОЕКТ 2	Иваненко

Рис. 3.5. Отношение ПРОЕКТЫ и результат операции СЛУЖАЩИЕ_В_ПРОЕКТЕ_1 TIMES ПРОЕКТЫ

дов, а, во-вторых, результат операции не более информативен, чем взятые в совокупности операнды. Как будет показано далее, основной смысл включения операции расширенного декартова произведения в состав реляционной алгебры Кодда состоит в том, что на ее основе определяется действительно полезная операция соединения.

По поводу теоретико-множественных операций реляционной алгебры следует еще заметить, что все четыре операции являются ассоциативными. Т. е. если обозначить через OP любую из четырех операций, то

(*A* OP *B*) OP *C* = *A* (*B* OP *C*), и, следовательно, без внесения двусмысленности можно писать *A* OP *B* OP *C* (*A*, *B* и *C* – отношения, обладающие свойствами, необходимыми для корректного выполнения соответствующей операции). Все операции, кроме взятия разности, являются коммутативными, т. е. *A* OP *B* = *B* OP *A*.

Специальные реляционные операции

В этом разделе мы несколько подробнее рассмотрим специальные реляционные операции реляционной алгебры, такие, как ограничение, проекция, соединение и деление.

Операция ограничения

Операция ограничения WHERE требует наличия двух операндов: ограничиваемого отношения и простого условия ограничения. Простое условие ограничения может иметь:

- вид (*a* comp-op *b*), где *a* и *b* – имена атрибутов ограничиваемого отношения; атрибуты *a* и *b* должны быть определены на одном и том же домене, для значений базового типа данных которого поддерживается операция сравнения comp_op, или на базовых типах данных, над значениями которых можно выполнять эту операцию сравнения;
- или вид (*a* comp-op *const*), где *a* – имя атрибута ограничиваемого отношения, а *const* – литерально заданная константа; атрибут a должен быть определен на домене или базовом типе, для значений которого поддерживается операция сравнения comp_op.

Операцией сравнения comp-op могут быть «=», «≠», «>», «≥», «<», «≤». Простые условия вычисляются в трехзначной логике (см. разд. «Реляционная модель данных», лекция 2), и в результате выполнения операции ограничения производится отношение, заголовок которого совпадает с заголовком отношения-операнда, а в тело входят те кортежи отношения-операнда, для которых значением условия ограничения является *true*. Тем самым, если в некоторых кортежах содержатся неопределенные значения, и по данной причине вычисление простого условия дает значение *unknown*, то эти кортежи не войдут в результирующее отношение.

Для обозначения вызова операции ограничения будем использовать конструкцию *A* WHERE *comp*, где *A* – ограничиваемое отношение, а *comp* – простое условие сравнения. Пусть *comp1* и *comp2* – два простых условия ограничения. Тогда по определению:

- *A* WHERE (*comp1* AND *comp2*) обозначает то же самое, что и (*A* WHERE *comp1*) INTERSECT (*A* WHERE *comp2*);

- *A* WHERE (*comp1* OR *comp2*) обозначает то же самое, что и (*A* WHERE *comp1*) UNION (*A* WHERE *comp2*);
- *A* WHERE NOT *comp1* обозначает то же самое, что и *A* MINUS (*A* WHERE *comp1*).

Эти соглашения позволяют задействовать операции ограничения, в которых условием ограничения является произвольное булевское выражение, составленное из простых условий с использованием логических связок AND, OR, NOT и скобок.

Результат выполнения операции СЛУЖАЩИЕ_В_ПРОЕКТЕ_1 WHERE (СЛУ_ЗАРП > 20000.00 AND (СЛУ_ОТД_НОМ = 310 OR СЛУ_ОТД_НОМ = 315)) (получить данные из отношения СЛУЖАЩИЕ_В_ПРОЕКТЕ_1 о служащих, работающих в отделах 310 и 315 и получающих зарплату, превышающую 20000.00 руб.) показан на рис. 3.6.

СЛУ_НОМЕР	СЛУ_ИМЯ	СЛУ_ЗАРП	СЛУ_ОТД_НОМЕР
2934	Иванов	22000.00	310
2935	Петров	30000.00	310
2938	Иванова	22000.00	315

Рис. 3.6. Результат операции СЛУЖАЩИЕ_В_ПРОЕКТЕ_1 WHERE (СЛУ_ЗАРП > 20000.00 AND (СЛУ_ОТД_НОМ = 310 OR СЛУ_ОТД_НОМ = 315))

На интуитивном уровне операцию ограничения лучше всего представлять как взятие некоторой «горизонтальной» вырезки из отношения-операнда (выборки некоторых строк из таблицы).

Операция взятия проекции

Операция взятия проекции также требует наличия двух операндов – проецируемого отношения *A* и подмножества множества имен атрибутов, входящих в заголовок отношения *A*.

Результатом проекции отношения *A* на множество атрибутов $\{a_1, a_2, \ldots, a_n\}$ (PROJECT *A* $\{a_1, a_2, \ldots, a_n\}$) является отношение с заголовком, определяемым множеством атрибутов $\{a_1, a_2, \ldots, a_n\}$, и с телом, состоящим из кортежей вида $<a_1:v_1, a_2:v_2, \ldots, a_n:v_n>$ таких, что в отношении *A* имеется кортеж, атрибут a_1 которого имеет значение v_1, атрибут a_2 имеет значение v_2, ..., атрибут a_n имеет значение v_n. Тем самым, при выполнении операции проекции выделяется «вертикальная» вырезка отношения-операнда с естественным уничтожением потенциально возникающих кортежей-дубликатов.

Заметим, что потенциальная потребность удаления дубликатов очень сильно усложняет реализацию операции проекции, поскольку в общем случае для удаления дубликатов требуется сортировка промежуточного результата операции. Основная сложность состоит в том, что этот промежуточный результат в общем случае может быть очень большим, и для сортировки требуется применять дорогостоящие алгоритмы *внешней сортировки*, выполняемые с применением обменов с внешней памятью. (Под «стоимостью» действия понимается время его выполнения.)

Результат операции PROJECT СЛУЖАЩИЕ_В_ПРОЕКТЕ_1 {СЛУ_ОТД_НОМ} (в каких отделах работают служащие, данные о которых содержатся в отношении СЛУЖАЩИЕ_В_ПРОЕКТЕ_1?) показан на рис. 3.7.

СЛУ_ОТД_НОМЕР
310
313
315

Рис. 3.7. Результат выполнения операции PROJECT СЛУЖАЩИЕ_В_ПРОЕКТЕ_1 {СЛУ_ОТД_НОМ}

Операция соединения отношений

Общая операция соединения (называемая также соединением по условию) требует наличия двух операндов – соединяемых отношений и третьего операнда – простого условия. Пусть соединяются отношения *A* и *B*. Как и в случае операции ограничения, условие соединения *comp* имеет вид либо (*a* comp-op *b*), либо (*a* comp-op *const*), где *a* и *b* – имена атрибутов отношений *A* и *B*, *const* – литерально заданная константа, и comp-op – допустимая в данном контексте операция сравнения.

Тогда по определению результатом операции соединения *A* JOINT *B* WHERE comp совместимых по взятию расширенного декартова произведения отношений *A* и *B* является отношение, получаемое путем выполнения операции ограничения по условию comp расширенного декартова произведения отношений *A* и *B* (*A* JOINT *B* WHERE *comp* ≡ (*A* TIMES *B*) WHERE *comp*).

Если тщательно осмыслить это определение, то станет ясно, что в общем случае применение условия соединения существенно уменьшит мощность результата промежуточного декартова произведения отношений-операндов только в том случае, если условие соединения имеет вид (*a* comp-op *b*), где *a* и *b* – имена атрибутов разных отношений-операндов. Поэтому на практике обычно считают реальными операциями со-

единения именно те операции, которые основываются на условии соединения приведенного вида.

В подразделе, касающемся операции ограничения, мы определили трактовку использования в качестве ограничивающего условия произвольного булевского выражения, которое составлено из простых условий над атрибутами отношения-операнда и литеральными константами. Конечно же, и в операции соединения может задаваться произвольное логическое выражение, составленное из простых условий над атрибутами отношений-операндов и константами. Операцию соединения с таким условием *comp* разумно считать операцией *действительно* соединения, если оно имеет вид (или может быть преобразовано к виду) *comp1* AND (*a* comp-op *b*), где *a* и *b* – имена атрибутов разных отношений-операндов.

Для иллюстрации операций соединения мы немного изменим заголовки и тела отношений, которые использовались ранее в примерах этой лекции. Пусть теперь имеются отношения СЛУЖАЩИЕ {СЛУ_НОМЕР, СЛУ_ИМЯ, СЛУ_ЗАРП, ПРО_НОМ} (атрибут ПРО_НОМ содержит номера проектов, в которых участвует каждый служащий) и ПРОЕКТЫ {ПРО_НОМ, ПРОЕКТ_РУК, ПРО_ЗАРП} (ПРО_НОМ – номер проекта, ПРОЕКТ_РУК – имя служащего-руководителя проекта, ПРО_ЗАРП – средняя заработная плата служащих, участвующих в проекте). Примерное содержимое тел отношений СЛУЖАЩИЕ и ПРОЕКТЫ показано на рис. 3.8.

Тогда осмысленной операцией соединения общего вида будет СЛУЖАЩИЕ JOIN ПРОЕКТЫ WHERE (СЛУ_ЗАРП > ПРО_ЗАРП) (выдать данные о служащих, получающих заработную плату, превышающую среднюю заработную плату любого проекта). Результаты этого запроса показаны на рис. 3.9.

Хотя операция соединения в приведенной интерпретации не является примитивной (поскольку определяется с использованием операций декартова произведения и проекции), в силу особой практической важности она включается в базовый набор операций реляционной алгебры Кодда. Заметим также, что в практических реализациях соединение обычно не выполняется именно как ограничение декартова произведения. Имеются более эффективные алгоритмы, гарантирующие получение такого же результата.

Существует важный частный случай соединения – *эквисоединение* (EQUIJOIN) и простое, но важное расширение операции эквисоединения – *естественное соединение* (NATURAL JOIN). Операция соединения называется *операцией эквисоединения*, если условие соединения имеет вид (*a* = *b*), где *a* и *b* – атрибуты разных операндов соединения. Этот случай важен потому, что он чаще всего встречается на практике, и для него существуют наиболее эффективные алгоритмы реализации.

Операция естественного соединения применяется к паре отношений *A* и *B*, обладающих (возможно, составным) общим атрибутом *c* (т. е.

СЛУЖАЩИЕ

СЛУ_НОМЕР	СЛУ_ИМЯ	СЛУ_ЗАРП	ПРО_НОМ
2934	Иванов	22400.00	1
2935	Петров	29600.00	1
2936	Сидоров	18000.00	1
2937	Федоров	20000.00	1
2938	Иванова	22000.00	1
2934	Иванов	22400.00	2
2935	Петров	29600.00	2
2939	Сидоренко	18000.00	2
2940	Федоренко	20000.00	2
2941	Иваненко	22000.00	2

ПРОЕКТЫ

ПРО_НОМ	ПРОЕКТ_РУК	ПРО_ЗАРП
1	Иванов	22400.00
2	Иваненко	22400.00

Рис. 3.8. Отношения СЛУЖАЩИЕ и ПРОЕКТЫ

атрибутом с одним и тем же именем и определенным на одном и том же домене). Пусть *ab* обозначает объединение заголовков отношений *A* и *B*. Тогда естественное соединение *A* и *B* — это спроецированный на *ab* результат эквисоединения *A* и *B* по условию *A.c = B.c*.* Хотя операция естественного соединения выражается через операции переименования, соединения общего вида и проекции, для нее обычно используется сокращенная форма, называемая NATURAL JOIN.

На рис. 3.10 приведены результаты операций СЛУЖАЩИЕ JOIN (ПРОЕКТЫ RENAME (ПРО_НОМ, ПРО_НОМ1)) WHERE (СЛУ_ЗАРП = ПРО_ЗАРП) (эквисоединение отношений СЛУЖАЩИЕ и ПРОЕКТЫ: найти всех служащих, получающих зарплату, равную средней заработной плате в каком-либо проекте) и СЛУЖАЩИЕ NATURAL JOIN ПРОЕКТЫ (естественное соединение — выдать полную информацию о служащих и проектах, в которых они участвуют).

* Здесь *A.c* и *B.c* представляют собой так называемые *квалифицированные (уточненные)* имена атрибутов (часто такой способ именования называют точечной нотацией). Мы будем использовать подобную нотацию в тех случаях, когда требуется явно показать, схеме какого отношения принадлежит данный атрибут.

СЛУ_НОМЕР	СЛУ_ИМЯ	СЛУ_ЗАРП	ПРО_НОМ	ПРО_НОМ1	ПРОЕКТ_РУК	ПРО_ЗАРП
2935	Петров	29600.00	1	1	Иванов	22400.00
2935	Петров	29600.00	2	2	Иваненко	22400.00

Рис. 3.9. Результат операции
СЛУЖАЩИЕ JOIN ПРОЕКТЫ WHERE (СЛУ_ЗАРП > ПРО_ЗАРП)

Результат операции **СЛУЖАЩИЕ** JOIN (**ПРОЕКТЫ** RENAME (ПРО_НОМ, ПРО_НОМ1)) WHERE (СЛУ_ЗАРП = ПРО_ЗАРП)

СЛУ_НОМЕР	СЛУ_ИМЯ	СЛУ_ЗАРП	ПРО_НОМ	ПРО_НОМ1	ПРОЕКТ_РУК	ПРО_ЗАРП
2934	Иванов	22400.00	1	1	Иванов	22400.00
2934	Иванов	22400.00	2	2	Иваненко	22400.00

Результат операции **СЛУЖАЩИЕ** NATURAL JOIN **ПРОЕКТЫ**

СЛУ_НОМЕР	СЛУ_ИМЯ	СЛУ_ЗАРП	ПРО_НОМ	ПРОЕКТ_РУК	ПРО_ЗАРП
2934	Иванов	22400.00	1	Иванов	22400.00
2935	Петров	29600.00	1	Иванов	22400.00
2936	Сидоров	18000.00	1	Иванов	22400.00
2937	Федоров	20000.00	1	Иванов	22400.00
2938	Иванова	22000.00	1	Иванов	22400.00
2934	Иванов	22400.00	2	Иваненко	22400.00
2935	Петров	29600.00	2	Иваненко	22400.00
2939	Сидоренко	18000.00	2	Иваненко	22400.00
2940	Федоренко	20000.00	2	Иваненко	22400.00
2941	Иваненко	22000.00	2	Иваненко	22400.00

Рис. 3.10. Результаты операций эквисоединения и естественного соединения отношений СЛУЖАЩИЕ и ПРОЕКТЫ

Если вспомнить введенное нами в конце предыдущей лекции определение внешнего ключа отношения, то должно стать понятно, что основной смысл операции естественного соединения состоит в возможности восстановления сложной сущности, декомпозированной по причине требования первой нормальной формы. Операция естественного соединения не включается в состав набора операций данной реляционной алгебры Кодда, но имеет очень важное практическое значение.

Операция деления отношений

Эта операция наименее очевидна из всех операций реляционной алгебры Кодда и поэтому нуждается в более подробном объяснении. Пусть заданы два отношения – A с заголовком $\{a_1, a_2, \ldots, a_n, b_1, b_2, \ldots, b_m\}$ и B с заголовком $\{b_1, b_2, \ldots, b_m\}$. Будем считать, что атрибут b_i отношения A и атрибут b_i отношения B $(i = 1, 2, \ldots, m)$ не только обладают одним и тем же именем, но и определены на одном и том же домене. Назовем множество атрибутов $\{a_j\}$ составным атрибутом a, а множество атрибутов $\{b_j\}$ – составным атрибутом b. После этого будем говорить о реляционном делении «бинарного» отношения A $\{a, b\}$ на унарное отношение B $\{b\}$.

По определению, результатом деления A на B (A DIVIDE BY B) является «унарное» отношение C $\{a\}$, тело которого состоит из кортежей v таких, что в теле отношения A содержатся кортежи v *UNION* w такие, что множество $\{w\}$ включает тело отношения B. Операция реляционного деления не является примитивной и выражается через операции декартова произведения, взятия разности и проекции. Мы покажем это в следующей лекции.

Для иллюстрации этой операции предположим, что в базе данных служащих поддерживаются следующие отношения: СЛУЖАЩИЕ, как оно было определено ранее, и унарное отношение НОМЕРА_ПРОЕКТОВ {ПРО_НОМ} (рис. 3.11). Тогда запрос СЛУЖАЩИЕ DIVIDE BY НОМЕРА_ПРОЕКТОВ выдаст данные обо всех служащих, участвующих во всех проектах (результат операции приведен также на рис. 3.11).

Отношение **НОМЕРА_ПРОЕКТОВ**

ПРО_НОМ
1
2

Результат операции **СЛУЖАЩИЕ** DIVIDE BY **НОМЕРА_ПРОЕКТОВ**

СЛУ_НОМЕР	СЛУ_ИМЯ	СЛУ_ЗАРП
2934	Иванов	22400.00
2935	Петров	29600.00

Рис. 3.11. Пример реляционного деления

Заключение

В завершение лекции хочу отметить несколько моментов. Прежде всего, заметим, что алгебра Кодда была представлена не в ее оригинальной форме, а с некоторыми существенными коррективами, внесенными Кристофером Дейтом. С моей точки зрения, одной из наиболее значительных корректив было добавление тривиальной на первый взгляд операции переименования атрибутов. Когда Эдгар Кодд в конце 1960-х гг. впервые опубликовал свою алгебру, основное внимание в ней уделялось тому, как конструируются результирующие множества кортежей, т. е. что представляют собой тела результатов операций. Гораздо меньше внимания уделялось заголовкам отношений-результатов. Фактически Кодд пытался применить для именования атрибутов результатов операций точечную нотацию, используя для уточнения имен атрибутов имена исходных отношений-операндов. При наличии произвольно сложных и длинных алгебраических выражений этот путь, в лучшем случае, вел к порождению длинных и трудных для восприятия имен. Очевидно, что введение операции переименования атрибутов позволяет легко справиться с этой проблемой.

Далее, алгебра Кодда исключительно избыточна. Операции пересечения, декартова произведения и естественного соединения, на самом деле, являются частными случаями одной более общей операции, о которой пойдет речь в следующей лекции. Введение операции декартова произведения в качестве базовой операции алгебры может ввести в заблуждение неопытных студентов и читателей, не осознающих практическую бессмысленность этой операции.

Почему же мы начали обсуждение базовых манипуляционных механизмов реляционной модели данных с этой небезупречной и несколько устаревшей алгебры? Конечно, прежде всего, из уважения к заслугам доктора Эдгара Кодда, вклад которого в современную технологию баз данных невозможно переоценить. Более практические соображения, повлиявшие на наше решение начать обсуждение с алгебры Кодда, заключались в том, что семантика языка SQL во многом базируется именно на этой алгебре, и нам будет проще изучать SQL, предварительно познакомившись с ней.

Лекция 4. Базисные средства манипулирования реляционными данными: алгебра А Дейта и Дарвена

В этой лекции мы обсудим новый «минимальный» вариант алгебры, предложенный несколько лет тому назад Дейтом и Дарвеном. Как уже отмечалось в предыдущей лекции, возможно, новая алгебра не очень практична, но зато красива и элегантна.

Ключевые слова: алгебра А, операции реляционного отрицания (дополнения), операция реляционной конъюнкции, операции пересечения, естественного соединения и взятия декартова произведения как частные случаи операции реляционной конъюнкции, операция реляционной дизъюнкции, операция объединения как частный случай операции реляционной дизъюнкции, операция удаления атрибута, полнота Алгебры А, выводимость операции взятия разности, ограничение отношений на основе операции реляционной конъюнкции, константные отношения, соединение отношений на основе базовых операций Алгебры А, выводимость операции реляционного деления, избыточность Алгебры А, реляционные аналоги логических операций «штрих Шеффера» и «стрелка Пирса», выводимость операции переименования.

Введение

Обсуждавшаяся в предыдущей лекции алгебра Кодда в большей степени базируется на теории множеств. Базовыми операциями являются переименование атрибутов, объединение, пересечение, взятие разности, декартово произведение, проекция и ограничение. Операция соединения общего вида, хотя и включается в алгебру, является вторичной и явно представляется через другие операции. Фундаментальная же в реляционном подходе операция естественного соединения выражается через соединение общего вида и в алгебру не включается. В терминах алгебры Кодда проще всего определяются алгебраические черты языка SQL, в частности общая семантика оператора `SELECT`.

Базисом предложенной Крисом Дейтом и Хью Дарвеном Алгебры А являются операции реляционного отрицания (дополнения), реляционной конъюнкции (или дизъюнкции) и проекции (удаления атрибута). Реляционные аналоги логических операций определяются в терминах отношений на основе обычных теоретико-множественных операций и позволяют выражать напрямую операции пересечения, декартова произведения, естественного соединения, объединения отношений и т. д. Путем комбинирования базовых операций выражаются операции переименования атрибутов, со-

единения общего вида, взятия разности отношений. Алгебра А позволяет лучше осознать логические основы реляционной модели, хотя, безусловно, является в меньшей степени ориентированной на практическое применение, чем алгебра Кодда*. Даже сами авторы Алгебры А, Дейт и Дарвен, в своем учебном языке *Tutorial D* используют не Алгебру А напрямую, а некоторое ее надмножество, в большей степени напоминающее алгебру Кодда.

Базовые операции Алгебры А

Материал этой лекции излагается на несколько более формальном уровне, чем в предыдущих лекциях. Используемые понятия определяются, по существу, так же, как и в лекции 2, но для удобства и обеспечения точности изложения мы повторим определения.

Пусть r – отношение, A – имя атрибута отношения r, T – имя соответствующего типа (т. е. типа или домена атрибута A), v – значение типа T. Тогда:

- заголовком Hr отношения r называется множество атрибутов, т.е. упорядоченных пар вида $<A, T>$. По определению никакие два атрибута в этом множестве не могут содержать одно и то же имя атрибута A;
- кортеж tr, соответствующий заголовку Hr, – это множество упорядоченных триплетов вида $<A, T, v>$, по одному такому триплету для каждого атрибута в Hr;
- тело Br отношения r – это множество кортежей tr. Заметим, что (в общем случае) могут существовать такие кортежи tr, которые соответствуют Hr, но не входят в Br.

Заметим, что заголовок – это множество (упорядоченных пар вида $<A, T>$), тело – это множество (кортежей tr), и кортеж – это множество (упорядоченных триплетов вида $<A, T, v>$). Элемент заголовка – это атрибут (т. е. упорядоченная пара вида $<A, T>$); элемент тела – это кортеж; элемент кортежа – это упорядоченный триплет вида $<A, T, v>$. Любое подмножество заголовка – это заголовок, любое подмножество тела – это тело, и любое подмножество кортежа – это кортеж.

Определим несколько основных операций (как будет показано далее, некоторые из них избыточны). Каждое из последующих определений состоит из: формальной спецификации ограничений (если они имеются), применимых к операндам соответствующей операции; формальной спецификации заголовка результата этой операции; формальной спецификации тела этого результата и неформального обсуждения формальных спецификаций.

* Нельзя не упомянуть еще и о том, что «алгебра» Кодда в действительности не является алгеброй отношений в математическом смысле, поскольку ее операции применимы не ко всем отношениям. В отличие от этого алгебра А – это «настоящая» алгебра, в которой отсутствуют какие-либо ограничения на операнды.

Во всех формальных спецификациях exists обозначает *квантор существования*; exists *tr* означает «существует такой *tr*, что». Символ «⊆» означает принадлежность одного множества другому; *tr*∈*Br* означает, что элемент *tr* принадлежит множеству *Br*. Выражение *tr*∉*Br* означает, что элемент *tr* не принадлежит множеству *Br*. Операции *minus* и *union* являются традиционными теоретико-множественными операциями взятия разности и объединения множеств.

Поскольку некоторые базовые операции Алгебры А имеют названия обычных логических операций, чтобы избежать путаницы, имена реляционных операций берутся в угловые скобки: <NOT>, <AND>, <OR> и т. д. В исходный базовый набор операций входят операции реляционного дополнения <NOT>, удаления атрибута <REMOVE>, переименования атрибута <RENAME>, реляционной конъюнкции <AND> и реляционной дизъюнкции <OR>.

Операция реляционного дополнения

Пусть *s* обозначает результат операции <NOT> *r*. Тогда:

- *Hs* = *Hr* (заголовок результата совпадает с заголовком операнда);
- *Bs* = {*ts* : exists *tr* (*tr* ∉ *Br* and *ts* = *tr*)} (в тело результата входят все кортежи, соответствующие заголовку и не входящие в тело операнда).

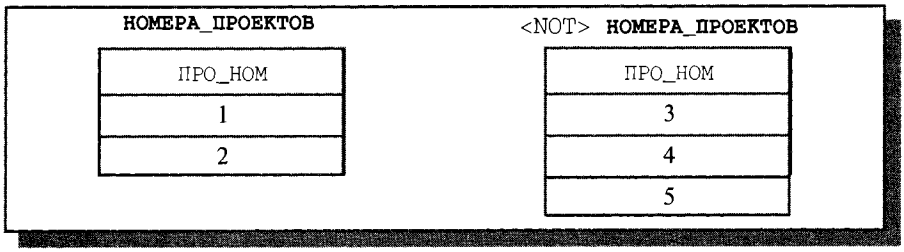
Операция <NOT> производит дополнение *s* заданного отношения *r*. Заголовком *s* является заголовок *r*. Тело *s* включает все кортежи, соответствующие этому заголовку и не входящие в тело *r*.

Видимо, следует пояснить, почему реляционный аналог операции логического отрицания называется здесь операцией реляционного дополнения. Во-первых, термин «дополнение» полностью соответствует сути операции <NOT>: тело результата операции <NOT> *r* является дополнением *Br* до полного множества кортежей, соответствующих *Hr*. Во-вторых, это не противоречит природе булевской операции NOT: у булевского типа имеются всего два значения – *true* и *false*, и NOT *true* = *false*, а NOT *false* = *true*. (Кстати, обратите внимание, что операцию NOT в трехзначной логике (см. лекцию 1) уже нельзя считать операцией дополнения.)

Чтобы привести пример использования операции <NOT>, предположим, что в состав домена ДОПУСТИМЫЕ_НОМЕРА_ПРОЕКТОВ, на котором определен атрибут ПРО_НОМ отношения НОМЕРА_ПРОЕКТОВ с рис. 4.1 слева, входит всего пять значений {1, 2, 3, 4, 5}. Тогда результат операции <NOT> НОМЕРА_ПРОЕКТОВ будет таким, как показано на рис. 4.1 справа.

Операция удаления атрибута

Пусть *s* обозначает результат операции *r* <REMOVE> A. Для обеспечения возможности выполнения операции требуется, чтобы существовал



НОМЕРА_ПРОЕКТОВ

ПРО_НОМ
1
2

<NOT> НОМЕРА_ПРОЕКТОВ

ПРО_НОМ
3
4
5

Рис. 4.1. Результат операции <NOT> НОМЕРА_ПРОЕКТОВ

некоторый тип (или домен) *T* такой, что <*A*, *T*> ∈ *Hr* (т. е. в состав заголовка отношения *r* должен входить атрибут *A*). Тогда:

- *Hs* = *Hr* minus {<*A*, *T*>}, т. е. заголовок результата получается из заголовка операнда изъятием атрибута *A*;
- *Bs* = {*ts* : exists *tr* exists *v* (*tr* ∈ *Br* and *v* ∈ *T* and <*A*,*T*,*v*> ∈ *tr* and *ts* = *tr minus* {<*A*,*T*,*v*>})}, т. е. в тело результата входят все кортежи операнда, из которых удалено значение атрибута *A*.

Операция <REMOVE> производит отношение *s*, формируемое путем удаления указанного атрибута *A* из заданного отношения *r*. Операция эквивалентна взятию проекции *r* на все атрибуты, кроме *A*. Заголовок *s* получается теоретико-множественным вычитанием из заголовка *r* множества из одного элемента {<*A*, *T*>}. Тело *s* состоит из таких кортежей, которые соответствуют заголовку s, причем каждый из них является подмножеством некоторого кортежа тела отношения *r*.

Примером операции REMOVE (конечно же, очень похожим на пример использования операции PROJECT из предыдущей лекции) является СЛУЖАЩИЕ REMOVE ПРО_НОМ (получить данные о служащих, участвующих в проектах). Результат этой операции над отношением СЛУЖАЩИЕ, тело которого приведено в верхней части рис. 4.2, показан на рис. 4.2 внизу.

Операция переименования

Пусть *s* обозначает результат операции *r* <RENAME> (*A*, *B*). Для обеспечения возможности выполнения операции требуется, чтобы существовал некоторый тип *T*, такой, что <*A*, *T*> ∈ *Hr*, и чтобы не существовал такой тип *T*, что <*B*, *T*> ∈ *Hr*. (Другими словами, в схеме отношения r должен присутствовать атрибут *A* и не должен присутствовать атрибут *B*.) Тогда:

- *Hs* = (*Hr* minus {<*A*, *T*>}) union {<*B*, *T*>}, т. е. в схеме результата *B* заменяет *A*;
- *Bs* = {*ts* : exists *tr* exists *v* (*tr* ∈ *Br* and *v* ∈ *T* and <*A*, *T*, *v*> ∈ *tr* and *ts* = (*tr* minus {<*A*, *T*, *v*>}) union {<*B*, *T*, *v*>})}, т. е. в кортежах тела результата имя значений атрибута *A* меняется на *B*.

СЛУЖАЩИЕ

СЛУ_НОМЕР	СЛУ_ИМЯ	СЛУ_ЗАРП	ПРО_НОМ
2934	Иванов	22400.00	1
2935	Петров	29600.00	1
2936	Сидоров	18000.00	1
2937	Федоров	20000.00	1
2938	Иванова	22000.00	1
2934	Иванов	22400.00	2
2935	Петров	29600.00	2
2939	Сидоренко	18000.00	2
2940	Федоренко	20000.00	2
2941	Иваненко	22000.00	2

СЛУ_НОМЕР	СЛУ_ИМЯ	СЛУ_ЗАРП
2934	Иванов	22400.00
2935	Петров	29600.00
2936	Сидоров	18000.00
2937	Федоров	20000.00
2938	Иванова	22000.00
2939	Сидоренко	18000.00
2940	Федоренко	20000.00
2941	Иваненко	22000.00

Рис. 4.2. Результат операции СЛУЖАЩИЕ REMOVE ПРО_НОМ

Операция <RENAME> производит отношение *s*, которое отличается от заданного отношения *r* только именем одного его атрибута, которое изменяется с *A* на *B*. Заголовок *s* такой же, как заголовок *r*, за исключением того, что пара <*B*, *T*> заменяет пару <*A*, *T*>. Тело *s* включает все кортежи тела *r*, но в каждом из этих кортежей триплет <*B*, *T*, *v*> заменяет триплет <*A*, *T*, *v*>.

По причине очевидности пример использования этой операции мы приводить не будем.

Операция реляционной конъюнкции

Пусть *s* обозначает результат операции *r1* <AND> *r2*. Для обеспечения возможности выполнения операции требуется, чтобы если <*A*, *T1*>∈H*r1* и <*A*, *T2*>∈H*r2*, то *T1*=*T2*. (Другими словами, если в двух отношениях-операндах имеются одноименные атрибуты, то они должны быть определены на одном и том же типе (домене).) Тогда:

- Hs = H*r1* union H*r2*, т. е. заголовок результата получается путем объединения заголовков отношений-операндов, как в операциях TIMES и JOIN из предыдущей лекции;
- Bs = { *ts* : exists *tr1* exists *tr2* ((*tr1*∈B*r1* and *tr2*∈B*r2*) and *ts* = *tr1* union *tr2*)}; обратите внимание на то, что кортеж результата определяется как *объединение кортежей операндов*; поэтому:
  - если схемы отношений-операндов имеют непустое пересечение, то операция <AND> работает как *естественное соединение*;
  - если пересечение схем операндов пусто, то <AND> работает как *расширенное декартово произведение*;
  - если схемы отношений полностью совпадают, то результатом операции является *пересечение* двух отношений-операндов.

Операция <AND> является реляционной *конъюнкцией*, в некоторых случаях выдающей в результате отношение *s*, ранее называвшееся естественным соединением двух заданных отношений *r1* и *r2*. Заголовок *s* является объединением заголовков *r1* и *r2*. Тело *s* включает каждый кортеж, соответствующий заголовку *s* и являющийся надмножеством некоторого кортежа из тела *r1* и некоторого кортежа из тела *r2*.

Для иллюстрации воспользуемся примерными отношениями, показанными на рис. 4.3, которые мы уже использовали в примерах предыдущей лекции.

На рис. 4.4(a) у отношений СЛУЖАЩИЕ и ПРОЕКТЫ имеется общий атрибут ПРО_НОМ. Поэтому операция <AND> работает как операция естественного соединения. На рис. 4.4(b) пересечение заголовков отношений СЛУЖАЩИЕ_В_ПРОЕКТЕ_1 и ПРОЕКТЫ пусто, и поэтому в результате реляционной конъюнкции производится расширенное декартово произведение этих отношений. Наконец, на рис. 4.4(c) схемы отношений СЛУЖАЩИЕ_В_ПРОЕКТЕ_1 и СЛУЖАЩИЕ_В_ПРОЕКТЕ_2 совпадают, и телом операции <AND> является пересечение тел отношений-операндов.

Операция реляционной дизъюнкции

Пусть *s* обозначает результат операции *r1* <OR> *r2*. Для обеспечения возможности выполнения операции требуется, чтобы если <*A*, *T1*>∈ H*r1*

СЛУЖАЩИЕ_В_ПРОЕКТЕ_1

СЛУ_НОМЕР	СЛУ_ИМЯ	СЛУ_ЗАРП	СЛУ_ОТД_НОМЕР
2934	Иванов	22000.00	310
2935	Петров	30000.00	310
2936	Сидоров	18000.00	313
2937	Федоров	20000.00	310
2938	Иванова	22000.00	315

СЛУЖАЩИЕ_В_ПРОЕКТЕ_2

СЛУ_НОМЕР	СЛУ_ИМЯ	СЛУ_ЗАРП	СЛУ_ОТД_НОМЕР
2934	Иванов	22000.00	310
2935	Петров	30000.00	310
2939	Сидоренко	18000.00	313
2940	Федоренко	20000.00	310
2941	Иваненко	22000.00	315

ПРОЕКТЫ

ПРО_НОМ	ПРОЕКТ_РУК
1	Иванов
2	Иваненко

СЛУЖАЩИЕ

СЛУ_НОМЕР	СЛУ_ИМЯ	СЛУ_ЗАРП	ПРО_НОМ
2934	Иванов	22400.00	1
2935	Петров	29600.00	1
2936	Сидоров	18000.00	1
2937	Федоров	20000.00	1
2938	Иванова	22000.00	1
2934	Иванов	22400.00	2
2935	Петров	29600.00	2
2939	Сидоренко	18000.00	2
2940	Федоренко	20000.00	2
2941	Иваненко	22000.00	2

Рис. 4.3. Примерные отношения для иллюстрации операции <AND>

(a) Результат операции **СЛУЖАЩИЕ** <AND> **ПРОЕКТЫ**

СЛУ_НОМЕР	СЛУ_ИМЯ	СЛУ_ЗАРП	ПРО_НОМ	ПРОЕКТ_РУК
2934	Иванов	22400.00	1	Иванов
2935	Петров	29600.00	1	Иванов
2936	Сидоров	18000.00	1	Иванов
2937	Федоров	20000.00	1	Иванов
2938	Иванова	22000.00	1	Иванов
2934	Иванов	22400.00	2	Иваненко
2935	Петров	29600.00	2	Иваненко
2939	Сидоренко	18000.00	2	Иваненко
2940	Федоренко	20000.00	2	Иваненко
2941	Иваненко	22000.00	2	Иваненко

(b) Результат операции **СЛУЖАЩИЕ_В_ПРОЕКТЕ_1** <AND> **ПРОЕКТЫ**

СЛУ_НОМЕР	СЛУ_ИМЯ	СЛУ_ЗАРП	СЛУ_ОТД_НОМЕР	ПРО_НОМ	ПРОЕКТ_РУК
2934	Иванов	22000.00	310	1	Иванов
2935	Петров	30000.00	310	1	Иванов
2936	Сидоров	18000.00	313	1	Иванов
2937	Федоров	20000.00	310	1	Иванов
2938	Иванова	22000.00	315	1	Иванов
2934	Иванов	22000.00	310	2	Иваненко
2935	Петров	30000.00	310	2	Иваненко
2936	Сидоров	18000.00	313	2	Иваненко
2937	Федоров	20000.00	310	2	Иваненко
2938	Иванова	22000.00	315	2	Иваненко

(c) Результат операции **СЛУЖАЩИЕ_В_ПРОЕКТЕ_1** <AND> **СЛУЖАЩИЕ_В_ПРОЕКТЕ_2**

СЛУ_НОМЕР	СЛУ_ИМЯ	СЛУ_ЗАРП	СЛУ_ОТД_НОМЕР
2934	Иванов	22000.00	310
2935	Петров	30000.00	310

Рис. 4.4. Иллюстрации операции реляционной конъюнкции

и <A, T2>∈Hr2, то должно быть T1 = T2 (одноименные атрибуты должны быть определены на одном и том же типе). Тогда:

- Hs = Hr1 union Hr2 (из схемы результата удаляются атрибуты-дубликаты);
- Bs = { ts : exists tr1 exists tr2 ((tr1∈Br1 or tr2∈Br2) and ts = tr1 union tr2)}; очевидно, что при этом:
  - если у операндов нет общих атрибутов, то в тело результирующего отношения входят все такие кортежи ts, которые являются объединением кортежей tr1 и tr2, соответствующих заголовкам отношений-операндов, и хотя бы один из этих кортежей принадлежит телу одного из операндов;
  - если у операндов имеются общие атрибуты, то в тело результирующего отношения входят все такие кортежи ts, которые являются объединением кортежей tr1 и tr2, соответствующих заголовкам отношений-операндов, если хотя бы один из этих кортежей принадлежит телу одного из операндов, и значения общих атрибутов tr1 и tr2 совпадают;
  - если же схемы отношений-операндов совпадают, то тело отношения-результата является объединением тел операндов.

Операция <OR> является реляционной *дизъюнкцией* и обобщением того, что ранее называлось объединением. Заголовок s есть объединение заголовков r1 и r2. Тело s состоит из всех кортежей, соответствующих заголовку s и являющихся надмножеством *либо* некоторого кортежа из тела r1, *либо* некоторого кортежа из тела r2.

Предположим, у нас имеются отношения ПРОЕКТЫ_1 {ПРОЕКТ_НАЗВ, ПРОЕКТ_РУК} и НОМЕРА_ПРОЕКТОВ {ПРО_НОМ} (рис. 4.5). Предположим также, что домен атрибута ПРОЕКТ_НАЗВ включает значения ПРОЕКТ_1, ПРОЕКТ_2, ПРОЕКТ_3, домен атрибута ПРОЕКТ_РУК ограничен значениями *Иванов*, *Иваненко*, а доменом атрибута ПРО_НОМ является множество {1, 2, 3}. Результат операции ПРОЕКТЫ <OR> НОМЕРА_ПРОЕКТОВ показан на рис. 4.5.

Как показано на рис. 4.5, операция <OR> при наличии операндов с несовпадающими схемами производит результат, гораздо более мощный, чем результат операции взятия расширенного декартова произведения из лекции 3, и еще менее осмысленный с практической точки зрения.

Для иллюстрации операции <OR> над операндами, схемы которых имеют непустое пересечение, воспользуемся отношением ПРОЕКТЫ_2 {ПРО_НОМ, ПРОЕКТ_РУК} (рис. 4.6) и унарным отношением НОМЕРА_ПРОЕКТОВ, схема и тело которого показаны на рис. 4.5. Будем предполагать, что множества значений доменов атрибутов такие же, как в предыдущем примере. Результат операции ПРОЕКТЫ_2 <OR> НОМЕРА_ПРОЕКТОВ показан на рис. 4.6.

ПРОЕКТЫ_1

ПРОЕКТ_НАЗВ	ПРОЕКТ_РУК
ПРОЕКТ 1	Иванов
ПРОЕКТ 2	Иваненко

НОМЕРА_ПРОЕКТОВ

ПРО_НОМ
1
2

Результат операции **ПРОЕКТЫ** <OR> **НОМЕРА_ПРОЕКТОВ**

ПРОЕКТ_НАЗВ	ПРОЕКТ_РУК	ПРО_НОМ
ПРОЕКТ 1	Иванов	1
ПРОЕКТ 2	Иванов	1
ПРОЕКТ 3	Иванов	1
ПРОЕКТ 1	Иваненко	1
ПРОЕКТ 2	Иваненко	1
ПРОЕКТ 3	Иваненко	1
ПРОЕКТ 1	Иванов	2
ПРОЕКТ 2	Иванов	2
ПРОЕКТ 3	Иванов	2
ПРОЕКТ 1	Иваненко	2
ПРОЕКТ 2	Иваненко	2
ПРОЕКТ 3	Иваненко	2
ПРОЕКТ 1	Иванов	3
ПРОЕКТ 2	Иваненко	3

Рис. 4.5. Результат операции <OR> над операндами без общих атрибутов

Как уже отмечалось, при совпадении схем отношений-операндов результатом выполнения над ними операции <OR> является объединение отношений. Это непосредственно следует из спецификации операции. Если этот факт кажется неочевидным, еще раз внимательно посмотрите на спецификацию. Иллюстрирующий пример мы приводить не будем.

ПРОЕКТЫ_2

ПРО_НОМ	ПРОЕКТ_РУК
1	Иванов
2	Иваненко

ПРОЕКТЫ_2 <OR> **НОМЕРА_ПРОЕКТОВ**

ПРО_НОМ	ПРОЕКТ_РУК
1	Иванов
2	Иваненко
2	Иванов
1	Иваненко

Рис. 4.6. Результат операции <OR> над операндами, схемы которых частично пересекаются

Полнота Алгебры А

Покажем, что Алгебра А является полной, т. е. на основе введенных операций выражаются все операции алгебры Кодда, рассмотренной в предыдущей лекции.

К настоящему моменту в состав базовых операций Алгебры А входят операция <REMOVE> в качестве аналога операции PROJECT, а также операция переименования атрибутов <RENAME>. UNION является частным случаем операции <OR>, TIMES, INTERSECT и NATURAL JOIN – частные случаи операции <AND>. Нам осталось показать, что через операции Алгебры А выражаются операции взятия разности MINUS, ограничения (WHERE), соединения общего вида (JOIN) и реляционного деления (DIVIDE BY).

Выводимость операции взятия разности

Покажем, что операция MINUS выражается через другие операции Алгебры А. Для наглядности снова воспользуемся отношениями СЛУЖАЩИЕ_В_ПРОЕКТЕ_1 и СЛУЖАЩИЕ_В_ПРОЕКТЕ_2 с рис. 4.3 (для удобства повторим его в верхней части рис. 4.7). Для простоты (хотя это несущественно) будем предполагать, что множества значений доменов, на которых определены атрибуты СЛУ_НОМЕР, СЛУ_ИМЯ, СЛУ_ЗАРП и СЛУ_ОТД_НОМЕР, ограничены значениями, содержащимися в телах отношений. Также для удобства покажем результат операции СЛУЖАЩИЕ_В_ПРОЕКТЕ_1 MINUS СЛУЖАЩИЕ_В_ПРОЕКТЕ_2 на рис. 4.7а. Заметим, что тело результата содержит все кортежи первого операнда, кроме кортежей Иванова и Петрова, поскольку они входят и в тело второго операнда.

СЛУЖАЩИЕ_В_ПРОЕКТЕ_1

СЛУ_НОМЕР	СЛУ_ИМЯ	СЛУ_ЗАРП	СЛУ_ОТД_НОМЕР
2934	Иванов	22000.00	310
2935	Петров	30000.00	310
2936	Сидоров	18000.00	313
2937	Федоров	20000.00	310
2938	Иванова	22000.00	315

СЛУЖАЩИЕ_В_ПРОЕКТЕ_2

СЛУ_НОМЕР	СЛУ_ИМЯ	СЛУ_ЗАРП	СЛУ_ОТД_НОМЕР
2934	Иванов	22000.00	310
2935	Петров	30000.00	310
2939	Сидоренко	18000.00	313
2940	Федоренко	20000.00	310
2941	Иваненко	22000.00	315

(a) **СЛУЖАЩИЕ_В_ПРОЕКТЕ_1** MINUS **СЛУЖАЩИЕ_В_ПРОЕКТЕ_2**

СЛУ_НОМЕР	СЛУ_ИМЯ	СЛУ_ЗАРП	СЛУ_ОТД_НОМЕР
2936	Сидоров	18000.00	313
2937	Федоров	20000.00	310
2938	Иванова	22000.00	315

(b) <NOT> **СЛУЖАЩИЕ_В_ПРОЕКТЕ_2**

СЛУ_НОМЕР	СЛУ_ИМЯ	СЛУ_ЗАРП	СЛУ_ОТД_НОМЕР
2935	Иванов	22000.00	310
2936	Иванов	22000.00	310
...	...	...	...
2941	Иванов	22000.00	310
2935	Иванов	30000.00	310
...	...	...	...
2934	Петров	30000.00	310
2936	Петров	30000.00	310
...	...	...	...

2934	Сидоров	18000.00	313
...	...	v	...
2937	Федоров	20000.00	310
....	...	...	...
2938	Иванова	22000.00	315
...	...	...	...

(c) **СЛУЖАЩИЕ_В_ПРОЕКТЕ_1** <AND> <NOT> **СЛУЖАЩИЕ_В_ПРОЕКТЕ_2**

СЛУ_НОМЕР	СЛУ_ИМЯ	СЛУ_ЗАРП	СЛУ_ОТД_НОМЕР
2936	Сидоров	18000.00	313
2937	Федоров	20000.00	310
2938	Иванова	22000.00	315

Рис. 4.7. Выразимость операции MINUS через операции <NOT> и <AND>

Посмотрим теперь, что является телом результата операции <NOT> СЛУЖАЩИЕ_В_ПРОЕКТЕ_2 (рис. 4.7b). В него входят все кортежи, соответствующие схеме отношения СЛУЖАЩИЕ_В_ПРОЕКТЕ_2 (и схеме отношения СЛУЖАЩИЕ_В_ПРОЕКТЕ_1), которые не входят в тело отношения СЛУЖАЩИЕ_В_ПРОЕКТЕ_2. В том числе в тело результата этой операции входят и кортежи Сидорова, Федорова и Ивановой из тела отношения СЛУЖАЩИЕ_В_ПРОЕКТЕ_1.

Тогда очевидно, что результат операции СЛУЖАЩИЕ_В_ПРОЕКТЕ_1 <AND> <NOT> СЛУЖАЩИЕ_В_ПРОЕКТЕ_2 (пересечение тела первого операнда с телом результата операции <NOT>) является в точности тем же, что и результат операции СЛУЖАЩИЕ_В_ПРОЕКТЕ_1 MINUS СЛУЖАЩИЕ_В_ПРОЕКТЕ_2 (рис. 4.7c).

В общем случае нетрудно доказать, что если отношения *r1* и *r2* совместимы по объединению, то *r1* MINUS *r2* = *r1* <AND> <NOT> *r2*.

Интерпретация операции ограничения

В лекции 3 мы определяли операцию ограничения *r* WHERE *comp*, где *r* – отношение, а *comp* – простое условие ограничения вида (*a* comp-op *b*), где *a* и *b* – имена атрибутов ограничиваемого отношения, для которых осмыслена операция сравнения comp-op, либо вида (*a* comp-op *const*), где *a* – имя атрибута ограничиваемого отношения, а *const* – литерально за-

данная константа. Операцией сравнения comp-op может быть «=», «≠», «>», «<», «≥», «≤». Покажем на нескольких примерах, как можно выразить операцию ограничения с помощью базовых операций Алгебры А для всех простых допустимых условий.

Для иллюстрации будем использовать отношение СЛУЖАЩИЕ_1 {СЛУ_НОМЕР, СЛУ_ИМЯ, СЛУ_ЗАРП, РУК_НОМ} (рис. 4.8). Атрибут РУК_НОМ содержит уникальные номера служащих, являющихся руководителями проектов, и определен на том же домене, что и СЛУ_НОМЕР. Мы снова предположим (для упрощения примеров), что множества значений доменов, на которых определены атрибуты отношения СЛУЖАЩИЕ_1, ограничены значениями, содержащимися в теле этого отношения. Начнем с обсуждения операции WHERE с условием вида *a comp-op const*.

Предположим, что мы хотим найти всех служащих с заработной платой, равной 20000.00 руб. Возьмем отношение ЗАРП_20000 {СЛУ_ЗАРП}*. Мы видим, что результат операции СЛУЖАЩИЕ_1 <AND> ЗАРП_20000 в точности совпадает с результатом операции СЛУЖАЩИЕ_1 WHERE СЛУ_ЗАРП = 20000.00 (рис. 4.8).

Если требуется найти служащих, чья заработная плата превышает 20000.00 руб., то возьмем отношение ЗАРП_БОЛЬШЕ_20000 (рис. 4.9)**. Тогда снова результат операции СЛУЖАЩИЕ_1 <AND> ЗАРП_БОЛЬШЕ_20000.00 будет совпадать с результатом операции СЛУЖАЩИЕ_1 WHERE СЛУ_ЗАРП > 20000.00 (рис. 4.9).

Понятно, что аналогичным образом выражаются через <AND> операции ограничения с условиями вида *a comp_op const*, в которых comp_op является «<», «≥» или «>». Некоторый особый случай представляет условие вида *a ≠ const*, и мы проиллюстрируем этот случай на примере запроса «Выбрать всех служащих, не получающих заработную плату в размере 22 000.00 руб.». Возьмем отношение ЗАРП_НЕ_22000 (рис. 4.10)***. Результат операции СЛУЖАЩИЕ_1 <AND> ЗАРП_НЕ_22000 будет совпадать с результатом операции СЛУЖАЩИЕ_1 WHERE СЛУ_ЗАРП 22000.00 (рис. 4.10).

---

* Здесь необходимо пояснить, что отношение ЗАРП_20000 в действительности представляет собой литеральную константу соответствующего типа отношений. Мы не вводим здесь строгого понятия типа отношения; для понимания данного подраздела нужно всего лишь осознать, что по своей природе отношение ЗАРП_20000 и числовой литерал 20000.00 не различаются.

** Отношение ЗАРП_БОЛЬШЕ_20000 – это тоже литеральная константа того же типа отношения, что и ЗАРП_20000, однако мощность тела этого литерального отношения в общем случае (если бы мы не ввели ограничения на множество значений домена СЛУ_ЗАРП) могла бы быть очень большой.

*** Особенность этого случая состоит в том, что число кортежей в теле литеральной константы ЗАРП_НЕ_22000 всего лишь на единицу меньше мощности множества значений домена СЛУ_ЗАРП. Конечно, эта мощность конечна, поскольку мы имеем дело с компьютерными типами данных, но в общем случае может быть очень большой. Поэтому принципиальная возможность выражения операции ограничения через операцию реляционной конъюнкции не означает, что было бы разумно реализовывать ее таким образом на практике.

СЛУЖАЩИЕ_1

СЛУ_НОМЕР	СЛУ_ИМЯ	СЛУ_ЗАРП	РУК_НОМ
2934	Иванов	22000.00	2934
2935	Петров	30000.00	2934
2936	Сидоров	18000.00	2934
2937	Федоров	20000.00	2934
2938	Иванова	22000.00	2941
2939	Сидоренко	18000.00	2941
2940	Федоренко	20000.00	2941
2941	Иваненко	22000.00	2941

ЗАРП_20000

СЛУ_ЗАРП
20000.00

СЛУЖАЩИЕ_1 <AND> **ЗАРП_20000**

СЛУ_НОМЕР	СЛУ_ИМЯ	СЛУ_ЗАРП	РУК_НОМ
2937	Федоров	20000.00	2934
2940	Федоренко	20000.00	2941

Рис. 4.8. Выражение `WHERE (a = const)` через <AND>

Теперь обратимся к ограничениям с простым условием вида *a* comp-op *b*. Опять начнем со случая, когда comp-op = «=». Предположим, что нам требуется найти данные о служащих, являющихся руководителями проектов, т. е. выполнить операцию СЛУЖАЩИЕ_1 WHERE СЛУ_НОМЕР = РУК_НОМ. Утверждается, что результат этой операции совпадает с результатом следующего выражения*:

СЛУЖАЩИЕ_1 <AND> ((((СЛУЖАЩИЕ_1 <REMOVE> СЛУ_НОМЕР) <REMOVE> СЛУ_ИМЯ) <REMOVE> СЛУ_ЗАРП) <RENAME> (РУК_НОМ, СЛУ_НОМЕР))

Результат вычисления правого операнда операции < AND> и окончательный результат операции показаны на рис. 4.11.

Конечно же, можно выразить операцию СЛУЖАЩИЕ_1 WHERE СЛУ_НОМЕР = РУК_НОМ через операцию <AND>, используя «константное» отношение. Для этого можно воспользоваться отношением СЛУ_НОМЕР_РУК_НОМ, показанным на рис. 4.12.** Очевидно, что в результате вы-

* Конечно, тот же результат даст и выражение СЛУЖАЩИЕ_1 <AND> ((((СЛУЖАЩИЕ_1 <REMOVE> РУК_НОМ) <REMOVE> СЛУ_ИМЯ) <REMOVE> СЛУ_ЗАРП) <RENAME> (СЛУ_НОМЕР, РУК_НОМ)).

** Конечно, в общем случае мощность тела такого константного отношения будет равна мощности соответствующего домена.

ЗАРП_БОЛЬШЕ_20000

СЛУ_ЗАРП
22000.00
30000.00

СЛУЖАЩИЕ_1 <AND> **ЗАРП_БОЛЬШЕ_20000**

СЛУ_НОМЕР	СЛУ_ИМЯ	СЛУ_ЗАРП	РУК_НОМ
2934	Иванов	22000.00	2934
2935	Петров	30000.00	2934
2938	Иванова	22000.00	2941
2941	Иваненко	22000.00	2941

Рис. 4.9. Выражение WHERE (*a* > *const*) через <AND>

полнения операции СЛУЖАЩИЕ_1 <AND> СЛУ_НОМЕР_РУК_НОМ будет получен тот же результат, что показан на рис. 4.11.

Чтобы показать возможность выполнения операции ограничения вида *r* WHERE (*a* > *b*), предположим, что имеется отношение СЛУЖАЩИЕ_2 {СЛУ_НОМЕР, СЛУ_ИМЯ, СЛУ_ЗАРП, СЛУ_ПРЕМ} (рис. 4.12), причем атрибут СЛУ_ПРЕМ содержит значения премиального вознагражде-

ЗАРП_НЕ_22000

СЛУ_ЗАРП
18000.00
20000.00
30000.00

СЛУЖАЩИЕ_1 <AND> **ЗАРП_НЕ_22000**

СЛУ_НОМЕР	СЛУ_ИМЯ	СЛУ_ЗАРП	РУК_НОМ
2935	Петров	30000.00	2934
2936	Сидоров	18000.00	2934
2937	Федоров	20000.00	2934
2939	Сидоренко	18000.00	2941
2940	Федоренко	20000.00	2941

Рис. 4.10. Выражение WHERE (*a* ≠ *const*) через <AND>

(((**СЛУЖАЩИЕ_1** <REMOVE> СЛУ_НОМЕР) <REMOVE> СЛУ_ИМЯ)
<REMOVE> СЛУ_ЗАРП) <RENAME> (РУК_НОМ, СЛУ_НОМЕР)

СЛУ_НОМЕР
2934
2941

СЛУЖАЩИЕ_1 <AND> ((((**СЛУЖАЩИЕ_1** <REMOVE> СЛУ_НОМЕР)
<REMOVE> СЛУ_ИМЯ) <REMOVE> СЛУ_ЗАРП) <RENAME>
(РУК_НОМ, СЛУ_НОМЕР))

СЛУ_НОМЕР	СЛУ_ИМЯ	СЛУ_ЗАРП	РУК_НОМ
2934	Иванов	22000.00	2934
2941	Иваненко	22000.00	2941

Рис. 4.11. Выражение WHERE (*a* = *b*) через <REMOVE>, <RENAME> и <AND>

ния служащего. Естественно, атрибуты СЛУ_ЗАРП и СЛУ_ПРЕМ определены на одном и том же домене (напомним, что в целях наших примеров мы предполагаем, что множество значений доменов ограничено значениями, содержащимися в теле примерного отношения). Пусть нас интересуют данные о служащих, получающих дополнительные вознаграждения в размере, превышающем размер основной зарплаты, т. е. нам нужен результат операции СЛУЖАЩИЕ_2 WHERE (СЛУ_ПРЕМ > СЛУ_ЗАРП).

ПРОЕКТЫ_2

СЛУ_НОМЕР	РУК_НОМ
2934	2934
2935	2935
2936	2936
2937	2937
2938	2938
2939	2939
2940	2940
2941	2941

Рис. 4.12. Константное отношение СЛУ_НОМЕР_РУК_НОМ

Возьмем отношение ПРЕМ_БОЛЬШЕ_ЗАРП {СЛУ_ПРЕМ, СЛУ_ЗАРП}, тело которого включает все соответствующие заголовку кортежи {*b*, *s*} такие, что b > s. Другими словами, отношение ПРЕМ_БОЛЬШЕ_ЗАРП снова является литеральной константой типа отношения с двумя атрибутами СЛУ_ПРЕМ и СЛУ_ЗАРП. Конечно, даже в случае нашего примера мощность тела этого отношения достаточно велика*. Тело отношения ПРЕМ_БОЛЬШЕ_ЗАРП показано в средней части рис. 4.13.

Результат выполнения операции СЛУЖАЩИЕ_2 <AND> ПРЕМ_БОЛЬШЕ_ЗАРП показан в нижней части рис. 4.13. Мы видим, что он совпадает с результатом операции СЛУЖАЩИЕ_2 WHERE (СЛУ_ПРЕМ > СЛУ_ЗАРП).

Аналогичным образом через операции Алгебры А выражаются операции ограничения, условия сравнения которых вида a comp_op b базируются на операциях сравнения «<», «⩾», «⩽», «≠».

Соединения общего вида

При наличии того факта, что операция взятия расширенного декартова произведения TIMES является частным случаем операции <AND>, после того как мы научились с помощью Алгебры А выполнять ограничения, становится очевидно, что через операции Алгебры А выражаются и соединения общего вида. В общем случае, чтобы получить результат соединения общего вида произвольных отношений *A* и *B*, нужно:

- выполнить над одним из отношений одну или несколько операций <RENAME>, чтобы избавиться от общих имен атрибутов;
- выполнить над полученными отношениями операцию <AND>, производящую расширенное декартово произведение;
- и для полученного отношения выполнить одну или несколько операций <AND> с отношениями-константами, чтобы должным образом ограничить его.

Реляционное деление

Пусть имеются отношения *r1* {*A*, *B*} и *r2* {*B*}. Утверждается, что результат *r1* DIVIDE BY *r2* совпадает с результатом выражения (*r1* PROJECT *A*) MINUS (((*r2* TIMES (*r1* PROJECT *A*)) MINUS *r1*) PROJECT *A*) в терминах операций реляционной алгебры Кодда или (*r1* <REMOVE> *B*) <AND> <NOT> (((*r2* <AND> (*r1* <REMOVE> *B*)) <AND> <NOT> *r1*) <REMOVE> *B*) в терминах операций Алгебры А.

Действительно, результатом выполнения операции *r1* PROJECT *A* является унарное отношение со схемой {*A*}, кортежи тела которого со-

* Легко убедиться, что в общем случае, если мощность общего домена атрибутов *A* и *B* равняется *n*, то мощность тела константного отношения *A*_БОЛЬШЕ_*B* будет составлять *(n-1)n/2*.

СЛУЖАЩИЕ_2

СЛУ_НОМЕР	СЛУ_ИМЯ	СЛУ_ЗАРП	СЛУ_ПРЕМ
2934	Иванов	22000.00	18000.00
2935	Петров	30000.00	22000.00
2936	Сидоров	18000.00	20000.00
2937	Федоров	20000.00	22000.00
2938	Иванова	22000.00	20000.00
2939	Сидоренко	18000.00	22000.00
2940	Федоренко	20000.00	20000.00
2941	Иваненко	22000.00	20000.00

ПРЕМ_БОЛЬШЕ_ЗАРП

СЛУ_ЗАРП	СЛУ_ПРЕМ
18000.00	20000.00
18000.00	22000.00
18000.00	30000.00
20000.00	22000.00
20000.00	30000.00
22000.00	30000.00

СЛУЖАЩИЕ_2 WHERE (СЛУ_ПРЕМ > СЛУ_ЗАРП)

СЛУ_НОМЕР	СЛУ_ИМЯ	СЛУ_ЗАРП	СЛУ_ПРЕМ
2936	Сидоров	18000.00	20000.00
2937	Федоров	20000.00	22000.00
2939	Сидоренко	18000.00	22000.00

Рис. 4.13. Выражение WHERE (*a* > *b*) через <AND>

держат все значения атрибута *A* из тела отношения *r1*. Результат выражения *r2* TIMES (*r1* PROJECT *A*) — это бинарное отношение со схемой {*A*, *B*}, в тело которого входят все возможные комбинации значений атрибута B в теле отношения *r2* и атрибута *A* в теле отношения *r1*. В теле результата вычисления выражения (*r2* TIMES (*r1* PROJECT *A*)) MINUS *r1* останутся только те кортежи, которые не входят во второй операнд, т. е. кортежи с таким значением атрибута *A*, что значение атрибута *B*, принад-

лежащее телу *r2*, не является значением атрибута *B* ни в одном кортеже тела отношения *r1*. Следовательно, если мы возьмем проекцию результата выражения (*r2* TIMES (*r1* PROJECT *A*)) MINUS *r1* на атрибут *A*, то в результирующем унарном отношении останутся только те значения *A*, которые не должны попасть в результат операции *r1* DIVIDE BY *r2*. После выполнения завершающей операции MINUS мы получим желаемый результат.

Для иллюстрации воспользуемся отношениями СЛУЖАЩИЕ и НОМЕРА_ПРОЕКТОВ, которые мы уже применяли в предыдущих примерах. Для удобства мы воспроизводим их на рис. 4.14. На этом же рисунке показаны промежуточные и окончательный результаты вычисления выражения (СЛУЖАЩИЕ **PROJECT** {СЛУ_НОМЕР, СЛУ_ИМЯ, СЛУ_ЗАРП}) **MINUS** ((((СЛУЖАЩИЕ **PROJECT** {СЛУ_НОМЕР, СЛУ_ИМЯ, СЛУ_ЗАРП}) **TIMES** НОМЕРА_ПРОЕКТОВ) **MINUS** СЛУЖАЩИЕ) **PROJECT** {СЛУ_НОМЕР, СЛУ_ИМЯ, СЛУ_ЗАРП}).

Тем самым, мы показали, что пяти операций Алгебры А достаточно для выражения всех операций алгебры Кодда из лекции 3. Но на самом деле число операций можно еще более сократить, что мы и продемонстрируем в следующем разделе.

Избыточность Алгебры А

В формальной математической логике стандартным базисом для выражения всех возможных булевских функций является набор {NOT, AND, OR} (отрицание, дизъюнкция и конъюнкция). Известно, что этот набор традиционен, но избыточен, поскольку верны тождества *A* AND *B* ≡ NOT (NOT *A* OR NOT *B*) и *A* OR *B* ≡ NOT (NOT *A* AND NOT *B*). (Эти тождества легко проверяются по таблицам истинности операций.) Оказывается (и это тоже легко проверить, опираясь на определения операций), что аналогичные тождества справедливы для операций <NOT>, <AND> и <OR> Алгебры А. Тем самым, в наборе базовых операций Алгебры А можно оставить операции <AND> и <NOT> (или <OR> и <NOT>).

Реляционные аналоги штриха Шеффера и стрелки Пирса

Более того, в алгебре логики существуют две операции, через каждую из которых выражаются все три «базовые» операции: «штрих Шеффера» – sh (*A*, *B*) ≡ NOT *A* OR NOT *B* – и «стрелка Пирса» – pi (*A*, *B*) ≡ NOT *A* AND NOT *B*.

Легко видеть, что

- sh (*A*, *A*) ≡ **NOT** *A*;
- sh (NOT *A*, NOT *B*) ≡ *A* **OR** *B* и
- NOT sh (*A*, *B*) ≡ *A* **AND** *B*.

Аналогично,

- pi (*A*, *A*) ≡ **NOT** *A*;
- pi (NOT *A*, NOT *B*) ≡ *A* **AND** *B* и
- NOT pi (*A*, *B*) ≡ *A* **OR** *B*.

Снова нетрудно проверить, что аналогичные тождества справедливы для реляционных вариантов штриха Шеффера (<sh> (*r1*, *r2*) ≡ <NOT> *r1* <OR> <NOT> *r2*) и стрелки Пирса (<pi> (*r1*, *r2*) ≡ <NOT> *r1* <AND> <NOT> *r2*).

Поэтому можно свести набор операций Алгебры А к трем операциям: <sh> (или <pi>), <RENAME> и <REMOVE>.

СЛУЖАЩИЕ

СЛУ_НОМЕР	СЛУ_ИМЯ	СЛУ_ЗАРП	ПРО_НОМ
2934	Иванов	22400.00	1
2935	Петров	29600.00	1
2936	Сидоров	18000.00	1
2937	Федоров	20000.00	1
2938	Иванова	22000.00	1
2934	Иванов	22400.00	2
2935	Петров	29600.00	2
2939	Сидоренко	18000.00	2
2940	Федоренко	20000.00	2
2941	Иваненко	22000.00	2

НОМЕРА_ПРОЕКТОВ

ПРО_НОМ
1
2

СЛУЖАЩИЕ **PROJECT** {СЛУ_НОМЕР, СЛУ_ИМЯ, СЛУ_ЗАРП}

СЛУ_НОМЕР	СЛУ_ИМЯ	СЛУ_ЗАРП
2934	Иванов	22400.00
2935	Петров	29600.00
2936	Сидоров	18000.00
2937	Федоров	20000.00
2938	Иванова	22000.00
2939	Сидоренко	18000.00
2940	Федоренко	20000.00
2941	Иваненко	22000.00

(**СЛУЖАЩИЕ** PROJECT {СЛУ_НОМЕР, СЛУ_ИМЯ, СЛУ_ЗАРП})
TIMES **НОМЕРА_ПРОЕКТОВ**

СЛУ_НОМЕР	СЛУ_ИМЯ	СЛУ_ЗАРП	ПРО_НОМ
2934	Иванов	22400.00	1
2934	Иванов	22400.00	2
2935	Петров	29600.00	1
2935	Петров	29600.00	2
2936	Сидоров	18000.00	1
2936	Сидоров	18000.00	2
2937	Федоров	20000.00	1
2937	Федоров	20000.00	2
2938	Иванова	22000.00	1
2938	Иванова	22000.00	2
2939	Сидоренко	18000.00	1
2939	Сидоренко	18000.00	2
2940	Федоренко	20000.00	1
2940	Федоренко	20000.00	2
2941	Иваненко	22000.00	1
2941	Иваненко	22000.00	2

((**СЛУЖАЩИЕ** PROJECT {СЛУ_НОМЕР, СЛУ_ИМЯ, СЛУ_ЗАРП})
TIMES **НОМЕРА_ПРОЕКТОВ**) MINUS **СЛУЖАЩИЕ**

СЛУ_НОМЕР	СЛУ_ИМЯ	СЛУ_ЗАРП	ПРО_НОМ
2936	Сидоров	18000.00	2
2937	Федоров	20000.00	2
2938	Иванова	22000.00	2
2939	Сидоренко	18000.00	1
2940	Федоренко	20000.00	1
2941	Иваненко	22000.00	1

(**СЛУЖАЩИЕ** PROJECT {СЛУ_НОМЕР, СЛУ_ИМЯ, СЛУ_ЗАРП})
MINUS ((((**СЛУЖАЩИЕ** PROJECT { СЛУ_НОМЕР, СЛУ_ИМЯ, СЛУ_ЗАРП}) TIMES
НОМЕРА_ПРОЕКТОВ) MINUS **СЛУЖАЩИЕ**)
PROJECT { СЛУ_НОМЕР, СЛУ_ИМЯ, СЛУ_ЗАРП})

СЛУ_НОМЕР	СЛУ_ИМЯ	СЛУ_ЗАРП
2934	Иванов	22400.00
2935	Петров	29600.00

Рис. 4.14. Выражение операции DIVIDE BY через другие операции Алгебры А

Избыточность операции переименования

Наконец, покажем, что избыточна и операция <RENAME>. Для иллюстрации снова воспользуемся отношением СЛУЖАЩИЕ из рис. 4.14. Пусть нам нужен результат операции СЛУЖАЩИЕ <RENAME> (ПРО_НОМ, НОМЕР_ПРОЕКТА) (мы по-прежнему предполагаем, что множество значений домена атрибута ПРО_НОМ ограничено значениями, представленными в теле отношения СЛУЖАЩИЕ). Возьмем бинарное отношение ПРО_НОМ_НОМЕР_ПРОЕКТА (рис. 4.15), где каждый из кортежей содержит два одинаковых значения номера проекта и в тело отношения входят все значения домена атрибута ПРО_НОМ.* Тогда, как показано на рис. 4.15, вычисление выражения (СЛУЖАЩИЕ <AND> ПРО_НОМ_НОМЕР_ПРОЕКТА) <REMOVE> (ПРО_НОМ) приводит к желаемому результату.

Тем самым, можно сократить набор операций Алгебры А до двух операций: <sh> (или <pi>) и <REMOVE>.**

Заключение

Базисом Алгебры А, являющейся алгеброй отношений в строгом математическом смысле, являются операции реляционного отрицания (дополнения), реляционной конъюнкции (или дизъюнкции) и проекции (удаления атрибута). Реляционные аналоги логических операций определяются в терминах отношений на основе обычных теоретико-множественных операций и позволяют выражать напрямую операции пересечения, декартова произведения, естественного соединения и объединения отношений. Путем комбинирования базовых операций выражаются операции переименования атрибутов, соединения общего вида, взятия разности отношений. Алгебра А позволяет лучше осознать логические основы реляционной модели, хотя, безусловно, является в меньшей степени ориентированной на практическое применение, чем алгебра Кодда.

Как нам кажется, в методическом отношении Алгебра А важна, прежде всего, тем, что в ней реляционная операция естественного соединения является одной из базовых операций, в отличие от алгебры Кодда, где эта операция имела второстепенное значение. Это важно по той причине, что, как мы увидим в лекции 7, операция естественного соединения играет первостепенную роль в классическом подходе к проектированию реляционных баз данных на основе нормализации.

---

* Это «константное» отношение, тело которого не зависит от текущего содержания тела отношения СЛУЖАЩИЕ.

** И конечно, в Алгебре А, как и в алгебре Кодда, должна присутствовать операция присваивания переменной отношения.

ПРО_НОМ_НОМЕР_ПРОЕКТА

ПРО_НОМ	НОМЕР_ПРОЕКТА
1	1
2	2

СЛУЖАЩИЕ <AND> **ПРО_НОМ_НОМЕР_ПРОЕКТА**

СЛУ_НОМЕР	СЛУ_ИМЯ	СЛУ_ЗАРП	ПРО_НОМ	НОМЕР_ПРОЕКТА
2934	Иванов	22400.00	1	1
2935	Петров	29600.00	1	1
2936	Сидоров	18000.00	1	1
2937	Федоров	20000.00	1	1
2938	Иванова	22000.00	1	1
2934	Иванов	22400.00	2	2
2935	Петров	29600.00	2	2
2939	Сидоренко	18000.00	2	2
2940	Федоренко	20000.00	2	2
2941	Иваненко	22000.00	2	2

(**СЛУЖАЩИЕ** <AND> **ПРО_НОМ-НОМЕР_ПРОЕКТА**) <REMOVE> (ПРО_НОМ)

СЛУ_НОМЕР	СЛУ_ИМЯ	СЛУ_ЗАРП	НОМЕР_ПРОЕКТА
2934	Иванов	22400.00	1
2935	Петров	29600.00	1
2936	Сидоров	18000.00	1
2937	Федоров	20000.00	1
2938	Иванова	22000.00	1
2934	Иванов	22400.00	2
2935	Петров	29600.00	2
2939	Сидоренко	18000.00	2
2940	Федоренко	20000.00	2
2941	Иваненко	22000.00	2

Рис. 4.15. Избыточность операции <RENAME>

Лекция 5. Базисные средства манипулирования реляционными данными: реляционное исчисление

Эта лекция завершает цикл лекций, посвященных манипуляционному аспекту реляционной модели данных. Материал лекции интересен и сам по себе, поскольку демонстрирует, насколько аппарат математической логики упрощает формулировку запросов к базам данных.

Ключевые слова: исчисление предикатов первого порядка, исчисление кортежей, кортежная переменная, оператор RANGE, правильно построенная формула (Well-Formed Formula, WFF), операция импликации, метод вложенных циклов (nested loops join), квантор существования (EXISTS), квантор всеобщности (FORALL), свободная переменная, связанная переменная, числовые функции над множествами, целевой список (target list), выражение исчисления кортежей, исчисление доменов, условие членства, предикат членства, выражение исчисления доменов, язык запросов Query-by-Example.

Введение

Предположим, что мы работаем с базой данных, которая состоит из отношений СЛУЖАЩИЕ {СЛУ_НОМ, СЛУ_ИМЯ, СЛУ_ЗАРП, ПРО_НОМ} и ПРОЕКТЫ {ПРО_НОМ, ПРОЕКТ_РУК, ПРО_ЗАРП} (в отношении ПРОЕКТЫ атрибут ПРОЕКТ_РУК содержит имена служащих, являющихся руководителями проектов, а атрибут ПРО_ЗАРП – среднее значение зарплаты, получаемой участниками проекта), и хотим узнать имена и номера служащих, которые являются руководителями проектов со средней заработной платой, превышающей 18000 руб.

Если бы для формулировки такого запроса использовалась реляционная алгебра, то мы получили бы, например, следующее алгебраическое выражение:

```
(СЛУЖАЩИЕ JOIN ПРОЕКТЫ WHERE (СЛУ_ИМЯ = ПРОЕКТ_РУК AND
   ПРО_ЗАРП > 18000.00)) PROJECT (СЛУ_ИМЯ, СЛУ_НОМ)
```

Это выражение можно было бы прочитать, например, следующим образом:

- выполнить эквисоединение отношений СЛУЖАЩИЕ и ПРОЕКТЫ по условию СЛУ_НОМ = ПРОЕКТ_РУК;
- ограничить полученное отношение по условию ПРО_ЗАРП > 18000.00;
- спроецировать результат предыдущей операции на атрибут СЛУ_ИМЯ, СЛУ_НОМ.

Мы четко сформулировали последовательность шагов выполнения запроса, каждый из которых соответствует одной реляционной операции.

Если же сформулировать тот же запрос с использованием реляционного исчисления, которому посвящается эта лекция, то мы получили бы два определения переменных:

```
RANGE СЛУЖАЩИЙ IS СЛУЖАЩИЕ и
RANGE ПРОЕКТ IS ПРОЕКТЫ
```

и выражение

```
СЛУЖАЩИЙ.СЛУ_ИМЯ, СЛУЖАЩИЙ.СЛУ _НОМ
WHERE EXISTS (СЛУЖАЩИЙ.СЛУ_ИМЯ = ПРОЕКТ.ПРОЕКТ_РУК
   AND ПРОЕКТ.ПРО_ЗАРП > 18000.00).
```

Это выражение можно было бы прочитать, например, следующим образом: выдать значения СЛУ_ИМЯ и СЛУ_НОМ для каждого кортежа служащих такого, что существует кортеж проектов со значением ПРОЕКТ_РУК, совпадающим со значением СЛУ_НОМ этого кортежа служащих, и значением ПРО_ЗАРП, большим 18000.00.

Во второй формулировке мы указали лишь характеристики результирующего отношения, но ничего не сказали о способе его формирования. В этом случае система сама должна решить, какие операции и в каком порядке нужно выполнить над отношениями СЛУЖАЩИЕ и ПРОЕКТЫ. Обычно говорят, что алгебраическая формулировка является процедурной, т. е. задающей последовательность действий для выполнения запроса, а логическая – описательной (или декларативной), поскольку она всего лишь описывает свойства желаемого результата. Как мы указывали в начале лекции 3, на самом деле эти два механизма эквивалентны, и существуют не слишком сложные правила преобразования одного формализма в другой.

Реляционное исчисление является прикладной ветвью формального механизма исчисления предикатов первого порядка.* В основе исчисления лежит понятие переменной с определенной для нее областью допустимых значений и понятие правильно построенной формулы, опирающейся на переменные, предикаты и кванторы.

В зависимости от того, что является областью определения переменной, различают исчисление кортежей и исчисление доменов. В исчислении кортежей областями определения переменных являются тела отношений базы данных, т. е. допустимым значением каждой переменной является кортеж тела некоторого отношения. В исчислении доменов областями определения переменных являются домены, на которых определены атрибуты отношений базы данных, т. е. допустимым значением каждой

* Это совсем не означает, что для понимания этой лекции требуется знание исчисления предикатов. Автор стремился к тому, чтобы материал лекции был в основном самодостаточным.

переменной является значение некоторого домена. Мы рассмотрим более подробно исчисление кортежей, а в конце лекции коротко опишем особенности исчисления доменов.

Как и в лекциях, посвященных реляционной алгебре, в этой лекции нам не удастся избежать использования некоторого конкретного синтаксиса, который мы тем не менее формально определять не будем. Те или иные синтаксические конструкции будут вводиться по мере необходимости. В совокупности используемый синтаксис близок, но не полностью совпадает с синтаксисом языка баз данных QUEL, который долгое время являлся основным языком известной реляционной СУБД Ingres.

Исчисление кортежей

Для определения кортежной переменной используется оператор RANGE. Например, для того чтобы определить переменную СЛУЖАЩИЙ, областью определения которой является отношение СЛУЖАЩИЕ, нужно употребить конструкцию

```
RANGE СЛУЖАЩИЙ IS СЛУЖАЩИЕ
```

Как уже говорилось, из этого определения следует, что в любой момент времени переменная СЛУЖАЩИЙ представляет некоторый кортеж отношения СЛУЖАЩИЕ. При использовании кортежных переменных в формулах можно ссылаться на значение атрибута переменной (это аналогично тому, как, например, при программировании на языке С можно сослаться на значение поля структурной переменной). Например, для того, чтобы сослаться на значение атрибута СЛУ_ИМЯ переменной СЛУЖАЩИЙ, нужно употребить конструкцию СЛУЖАЩИЙ.СЛУ_ИМЯ.

Правильно построенные формулы

Правильно построенная формула (Well-Formed Formula, WFF) служит для выражения условий, накладываемых на кортежные переменные.

Простые условия

Основой WFF являются простые условия, представляющие собой операции сравнения скалярных значений (значений атрибутов переменных или литерально заданных констант). Например, конструкции

```
СЛУЖАЩИЙ.СЛУ_НОМ = 2934 и
СЛУЖАЩИЙ.СЛУ_НОМ = ПРОЕКТ.ПРОЕКТ_РУК
```

являются простыми условиями. Первое условие принимает значение *true* в том и только в том случае, когда значение атрибута СЛУ_НОМ кортежной

переменной СЛУЖАЩИЙ равно 2934. Второе условие принимает значение *true* в том и только в том случае, когда значения атрибутов СЛУ_НОМ и ПРОЕКТ_РУК переменных СЛУЖАЩИЙ и ПРОЕКТ совпадают.

По определению, простое сравнение является WFF, а WFF, заключенная в круглые скобки, представляет собой простое сравнение.

Более сложные варианты WFF строятся с помощью логических связок NOT, AND, OR и IF ... THEN* с учетом обычных приоритетов операций (NOT > AND > OR) и возможности расстановки скобок. Так, если *form* – WFF, а *comp* – простое сравнение, то NOT *form*, *comp* AND *form*, *comp* OR *form* и IF *comp* THEN *form* являются WFF.

Для примеров воспользуемся отношениями СЛУЖАЩИЕ, ПРОЕКТЫ и НОМЕРА_ПРОЕКТОВ из предыдущей лекции (см. рис. 5.1).

Правильно построенной является следующая формула:

```
IF СЛУЖАЩИЙ.СЛУ_ИМЯ = 'Иванов'
THEN (СЛУЖАЩИЙ.СЛУ_ЗАРП >= 22400.00 AND СЛУЖАЩИЙ.ПРО_НОМ = 1)
```

Эта формула будет принимать значение *true* для следующих значений кортежной переменной СЛУЖАЩИЙ:

СЛУ_НОМЕР	СЛУ_ИМЯ	СЛУ_ЗАРП	ПРО_НОМ
2934	Иванов	22400.00	1
2935	Петров	29600.00	1
2936	Сидоров	18000.00	1
2937	Федоров	20000.00	1
2938	Иванова	22000.00	1
2935	Петров	29600.00	2
2939	Сидоренко	18000.00	2
2940	Федоренко	20000.00	2
2941	Иваненко	22000.00	2

Конечно, нужно представлять себе какой-нибудь способ реализации системы, которая сможет по заданной WFF при существующем состоянии базы данных произвести такой результат. И таким очевидным способом является следующий: в некотором порядке просмотреть область определения переменной и к каждому очередному кортежу применить условие. Ре-

* Через IF ... THEN здесь обозначается одна и важных логических функций – импликация. По определению, IF *a* THEN *b* ≡ NOT *a* OR *b*. Хотя операция импликации является избыточной, она явно вводится в реляционное исчисление, поскольку часто требуется на практике для выражения условий.

СЛУЖАЩИЕ

СЛУ_НОМЕР	СЛУ_ИМЯ	СЛУ_ЗАРП	ПРО_НОМ
2934	Иванов	22400.00	1
2935	Петров	29600.00	1
2936	Сидоров	18000.00	1
2937	Федоров	20000.00	1
2938	Иванова	22000.00	1
2934	Иванов	22400.00	2
2935	Петров	29600.00	2
2939	Сидоренко	18000.00	2
2940	Федоренко	20000.00	2
2941	Иваненко	22000.00	2

ПРОЕКТЫ

ПРО_НОМ	ПРОЕКТ_РУК
1	Иванов
2	Иваненко

НОМЕРА_ПРОЕКТОВ

ПРО_НОМ
1
2

Рис. 5.1. Примерные значения отношений СЛУЖАЩИЕ, ПРОЕКТЫ и НОМЕРА_ПРОЕКТОВ

зультатом будет то множество кортежей, для которых при вычислении условия производится значение *true*. Очевидно, что результат эквивалентен выполнению алгебраической операции `СЛУЖАЩИЕ WHERE (NOT (СЛУЖАЩИЙ.СЛУ_ИМЯ = 'Иванов') OR (СЛУЖАЩИЙ.СЛУ_ЗАРП >= 22400.00 AND СЛУЖАЩИЙ.ПРО_НОМ = 1)` над отношением, тело которого представляет собой область определения кортежной переменной.

Пусть имеется следующее определение кортежной переменной `ПРОЕКТ`:

```
RANGE ПРОЕКТ IS ПРОЕКТЫ
```

Вот еще пример правильно построенной формулы:

```
СЛУЖАЩИЙ.СЛУ_ИМЯ = ПРОЕКТ.ПРОЕКТ_РУК
```

Эта формула будет принимать значение *true* для следующих пар значений кортежных переменных СЛУЖАЩИЙ и ПРОЕКТ:

СЛУЖАЩИЕ				ПРОЕКТЫ	
СЛУ_НОМЕР	СЛУ_ИМЯ	СЛУ_ЗАРП	ПРО_НОМ	ПРО_НОМ	ПРОЕКТ_ РУК
2934	Иванов	22400.00	1	1	Иванов
2941	Иваненко	22000.00	2	2	Иваненко
2941	Иванов	22400.00	2	1	Иванов

Очевидный способ реализации системы, которая по заданной WFF при существующем состоянии базы данных производит такой результат, заключается в следующем. В некотором порядке просматривать область определения (например) переменной СЛУЖАЩИЙ. Для каждого текущего кортежа из области определения переменной СЛУЖАЩИЙ просматривать область определения переменной ПРОЕКТ. Оставлять в области истинности те пары кортежей, для которых формула принимает значение *true*. Возможен и альтернативный подход: начать просмотр с области определения переменной ПРОЕКТ, и для каждого кортежа ПРОЕКТ просматривать область определения СЛУЖАЩИЙ.

Здесь нужно сделать несколько замечаний. Во-первых, если бы в данном случае формула была тождественно истинной (например, имела вид

```
(СЛУЖАЩИЙ.СЛУ_ИМЯ = СЛУЖАЩИЙ.СЛУ_ИМЯ) AND
(ПРОЕКТ.ПРОЕКТ_РУК = ПРОЕКТ.ПРОЕКТ_РУК)),
```

то областью истинности этой формулы являлось бы декартово произведение (в строгом математическом смысле) тел отношений СЛУЖАЩИЙ и ПРОЕКТ. В реляционном исчислении кортежей, как и в реляционной алгебре, принято иметь дело с операцией расширенного декартова произведения, и поэтому считается, что в подобных случаях областью истинности WFF является отношение, заголовок которого представляет собой объединение заголовков отношений, на телах которых определены кортежные переменные, а кортежи являются объединением соответствующих кортежей из областей определения переменных. При этом имя атрибута результирующего отношения уточняется именем соответствующей переменной. Поэтому правильнее было бы изображать область истинности формулы

```
СЛУЖАЩИЙ.СЛУ_ИМЯ = ПРОЕКТ.ПРОЕКТ_РУК
```

следующим образом:

СЛУЖАЩИЙ СЛУ_НОМЕР	СЛУЖАЩИЙ. СЛУ_ИМЯ	СЛУЖАЩИЙ. СЛУ_ЗАРП	СЛУЖАЩИЙ. ПРО_НОМ	ПРОЕКТ. ПРО_НОМ	ПРОЕКТ. ПРОЕКТ_РУК
2934	Иванов	22400.00	1	1	Иванов
2941	Иваненко	22000.00	2	2	Иваненко

Во-вторых, как видно, показанное результирующее отношение в точности совпадает с результатом алгебраической операции СЛУЖАЩИЕ JOIN ПРОЕКТЫ WHERE СЛУ_ИМЯ = ПРОЕКТ_РУК с учетом особенности именования атрибутов результирующего отношения. Наконец, заметим, что описанный выше способ реализации, который приводит к получению области истинности рассмотренной формулы, в действительности является наиболее общим (и зачастую неоптимальным) способом выполнения операций соединения (он называется методом *вложенных циклов – nested loops join*).

Кванторы, свободные и связанные переменные

При построении WFF допускается использование кванторов существования (EXISTS) и всеобщности (FORALL). Если *form* – это WFF, в которой участвует переменная var, то конструкции EXISTS *var* (*form*) и FORALL *var* (*form*) представляют собой WFF. По определению, формула EXISTS *var* (*form*) принимает значение *true* в том и только в том случае, если в области определения переменной *var* найдется хотя бы одно значение (кортеж), для которого WFF *form* принимает значение *true*. Формула FORALL *var* (*form*) принимает значение *true*, если для всех значений переменной *var* из ее области определения WFF *form* принимает значение *true*.

Переменные, входящие в WFF, могут быть *свободными* или *связанными*. По определению, все переменные, входящие в WFF, при построении которой не использовались кванторы, являются свободными. Фактически, это означает, что если для какого-то набора значений свободных кортежных переменных при вычислении WFF получено значение *true*, то эти значения кортежных переменных могут входить в результирующее отношение. Если же имя переменной использовано сразу после квантора при построении WFF вида EXISTS *var* (*form*) или FORALL *var* (*form*), то в этой WFF и во всех WFF, построенных с ее участием, *var* является связанной переменной. Это означает, что такая переменная не видна за пределами минимальной WFF, связавшей эту переменную. При вычислении значения такой WFF используется не одно значение связанной переменной, а вся область ее определения.

Пусть здесь и далее в этом разделе СЛУ1 и СЛУ2 представляют собой две кортежные переменные, определенные на отношении СЛУЖАЩИЕ. Тогда WFF

```
EXISTS СЛУ2 (СЛУ1.СЛУ_ЗАРП > СЛУ2.СЛУ_ЗАРП)
```

для текущего кортежа переменной СЛУ1 принимает значение *true* в том и только в том случае, если во всем отношении СЛУЖАЩИЕ найдется такой кортеж (ассоциированный с переменной СЛУ2), чтобы значение его атрибута СЛУ_ЗАРП удовлетворяло внутреннему условию сравнения. Легко видеть, что эта формула принимает значение *true* только для тех значений кортежной переменной СЛУ1, которые соответствуют служащим, не получающим минимальную зарплату. Соответствующее множество кортежей показано на рис. 5.2а (для тела отношения СЛУЖАЩИЕ из рис. 5.1).

(a) Область истинности WFF

EXISTS СЛУ2 (СЛУ1.СЛУ_ЗАРП > СЛУ2.СЛУ_ЗАРП)

СЛУ_НОМЕР	СЛУ_ИМЯ	СЛУ_ЗАРП	ПРО_НОМ
2934	Иванов	22400.00	1
2935	Петров	29600.00	1
2937	Федоров	20000.00	1
2938	Иванова	22000.00	1
2934	Иванов	22400.00	2
2935	Петров	29600.00	2
2940	Федоренко	20000.00	2
2941	Иваненко	22000.00	2

(b) Область истинности WFF

FORALL СЛУ2 (СЛУ1.СЛУ_ЗАРП ⩾ СЛУ2.СЛУ_ЗАРП)

СЛУ_НОМЕР	СЛУ_ИМЯ	СЛУ_ЗАРП	ПРО_НОМ
2935	Петров	29600.00	1
2935	Петров	29600.00	2

Рис. 5.2. Примеры правильно построенных формул с кванторами

Правильно построенная формула

```
FORALL СЛУ2 (СЛУ1.СЛУ_ЗАРП ⩾ СЛУ2.СЛУ_ЗАРП)
```

для текущего кортежа переменной СЛУ1 принимает значение *true* в том и только в том случае, если для всех кортежей отношения СЛУЖАЩИЕ (связанных с переменной СЛУ2) значения атрибута СЛУ_ЗАРП удовлетворяют условию сравнения. Снова легко видеть, что формула принимает значение *true* только для тех значений кортежной переменной СЛУ1, которые

соответствуют служащим, получающим максимальную зарплату.* Соответствующее множество кортежей показано на рис. 5.2b.

Очевидно, что показанные на рис. 5.2 отношения соответствуют условиям обеих формул. Но как в данном случае можно реализовать систему, которая по заданной формуле производит правильный результат? Наиболее очевидный способ интерпретации обеих обсуждавшихся выше формул следующий. В некотором порядке просматривать область определения свободной кортежной переменной СЛУ1. Для каждого очередного кортежа из области определения СЛУ1 просматривать область определения связанной переменной СЛУ2 до тех пор, пока не будет установлено истинностное значение формулы для данного кортежа СЛУ1 (в случае наличия квантора существования процесс просмотра для СЛУ2 можно остановить после нахождения первого кортежа, для которого значением подформулы, находящейся под знаком квантора, станет *true*; при наличии квантора всеобщности необходимо просмотреть всю область определения СЛУ2). Заметим, что здесь мы снова получаем два цикла, как и при интерпретации WFF с двумя свободными переменными. Но в данном случае во внешнем цикле обязательно просматривается область определения свободной переменной.

На самом деле, правильнее говорить не о свободных и связанных переменных, а о свободных и связанных вхождениях переменных. Если переменная *var* является связанной в WFF *form*, то во всех WFF, включающих *form*, вне *form* может использоваться вхождение того же имени переменной *var*, которое может быть свободным или связанным, но в любом случае не имеет никакого отношения к вхождению переменной *var* в WFF *form*. Вот пример:

```
EXISTS СЛУ2 (СЛУ1.ПРО_НОМ = СЛУ2.ПРО_НОМ
   AND СЛУ1.СЛУ_НОМЕР СЛУ2.СЛУ_НОМЕР)
   AND FORALL СЛУ2 (IF СЛУ1.ПРО_НОМ = СЛУ2.ПРО_НОМ
      THEN СЛУ1.СЛУ_ЗАРП = СЛУ2.СЛУ_ЗАРП)
```

Эта формула принимает значение *true* только для тех значений переменной СЛУ1, которые соответствуют служащим, участвующим в проектах с более чем одним участником, причем все участники проекта получают одну и ту же зарплату. Здесь мы имеем два связанных вхождения переменной СЛУ2 с совершенно разным смыслом. Грубо говоря, для текущего значения переменной СЛУ1 переменная СЛУ2 два раза «пробежит» свою область определения – первый раз при вычислении части формулы с кван-

* Упражнение для читателей. Почему в первой формуле (с EXISTS) использовано условие СЛУ1.СЛУ_ЗАР > СЛУ2.СЛУ_ЗАРП, а второй формуле (с FORALL) – СЛУ1.СЛУ_ЗАР ≥ СЛУ2.СЛУ_ЗАРП?

тором существования, а второй при вычислении части с квантором всеобщности. Кстати, к тому же результату приведет формула с одним квантором всеобщности вида:

```
FORALL СЛУ2 (IF (СЛУ1.ПРО_НОМ = СЛУ2.ПРО_НОМ AND
     СЛУ1.СЛУ_НОМЕР СЛУ2.СЛУ_НОМЕР)
   THEN СЛУ1.СЛУ_ЗАРП = СЛУ2.СЛУ_ЗАРП)
```

Легко заметить, что кванторы можно трактовать как булевские функции (функции, принимающие значения *true* или *false*) над множеством значений связанной кортежной переменной. С тем же успехом можно ввести в реляционное исчисление числовые функции над множествами, такие, как MIN (минимальное значение), MAX (максимальное значение), AVG (среднее значение) и т. д.

В этом случае можно было бы написать, например, WFF

```
СЛУ1.СЛУ_ЗАРП > MIN СЛУ2.СЛУ_ЗАРП (СЛУ1.ПРО_НОМ = СЛУ2.ПРО_НОМ)
```

в области истинности которой содержатся все кортежи отношения СЛУЖАЩИЕ, соответствующие тем служащим, которые получают заработную плату, превышающую минимальную зарплату служащих, участвующих в том же проекте. Понятно, что для получения результирующего отношения можно интерпретировать формулу таким же образом, как в обсуждавшемся выше случае наличия кванторов.

Целевые списки и выражения реляционного исчисления

Итак, WFF обеспечивают средства формулировки условия выборки из отношений БД. Чтобы можно было использовать исчисление для реальной работы с БД, требуется еще один компонент, который определяет набор и имена атрибутов результирующего отношения. Этот компонент называется целевым списком (*target list*).

Целевой список строится из целевых элементов, каждый из которых может иметь следующий вид:

- *var.attr*, где *var* — имя свободной переменной соответствующей WFF, а *attr* — имя атрибута отношения, на котором определена переменная *var*;
- *var*, что эквивалентно наличию подсписка $var.attr_1$, $var.attr_2$, ..., $var.attr_n$, где $\{attr_1, attr_2, \ldots, attr_n\}$ включает имена всех атрибутов определяющего отношения;
- *new_name = var.attr*; *new_name* — новое имя соответствующего атрибута результирующего отношения.

Последний вариант требуется в тех случаях, когда в WFF используется несколько свободных переменных с одинаковой областью определения. Фактически применение целевого списка к области истинности WFF эквивалентно действию алгебраической операции проекции, а последний из приведенных вариантов представляет собой некоторую разновидность алгебраической операции переименования атрибута.

Выражением реляционного исчисления кортежей называется конструкция вида `target_list WHERE WFF`. Значением выражения является отношение, тело которого определяется WFF, а множество атрибутов и их имена – целевым списком.

В качестве простого примера покажем выражение реляционного исчисления кортежей, результат которого совпадает с результатом операции СЛУЖАЩИЕ DIVIDE BY НОМЕРА_ПРОЕКТОВ (рис. 3.11 из лекции 3):

```
СЛУ1, СЛУ2 RANGE IS СЛУЖАЩИЕ
НОМЕР_ПРОЕКТА range is НОМЕРА_ПРОЕКТОВ
СЛУ1.СЛУ_НОМЕР, СЛУ1.СЛУ_ИМЯ, СЛУ1.СЛУ_ЗАРП
WHERE FORALL НОМЕР_ПРОЕКТА EXISTS СЛУ2
   (СЛУ1.СЛУ_НОМЕР = СЛУ2.СЛУ_НОМЕР AND
   СЛУ1.ПРО_НОМ = НОМЕРА_ПРОЕКТОВ.ПРО_НОМ)
```

Конечно, результатом этого выражения является отношение

СЛУ_НОМЕР	СЛУ_ИМЯ	СЛУ_ЗАРП
2934	Иванов	22400.00
2935	Петров	29600.00

Исчисление доменов

В исчислении доменов областью определения переменных являются не отношения, а домены. Применительно к базе данных СЛУЖАЩИЕ-ПРОЕКТЫ можно говорить, например, о доменных переменных ИМЯ (значения – допустимые имена) или НОСЛУ (значения – допустимые номера служащих).

Условия членства

Основным формальным отличием исчисления доменов от исчисления кортежей является наличие дополнительного множества предикатов, позволяющих выражать так называемые условия членства. Если R – это n-арное отношение с атрибутами $a_1, a_2, \ldots, a_n$, то условие членства имеет вид $R\ (a_{i1} : v_{i1},\ a_{i2} : v_{i2},\ \ldots,\ a_{im} : v_{im})\ \ (m \leqslant n)$, где v_{ij} – это либо литерально задаваемая константа, либо имя доменной переменной. Условие членства принимает значение *true* в том и только в том случае, ес-

ли в отношении R существует кортеж, содержащий указанные значения указанных атрибутов. Если v_{ij} – константа, то на атрибут a_{ij} накладывается жесткое условие, не зависящее от текущих значений доменных переменных; если же v_{ij} – имя доменной переменной, то условие членства может принимать разные значения при разных значениях этой переменной.

Для большей ясности приведем пару примеров. Для простоты будем считать, что мы определили доменные переменные, имена которых совпадают с именами атрибутов отношения СЛУЖАЩИЕ, а в случае, когда требуется несколько доменных переменных, определенных на одном домене, мы будем добавлять в конце имени цифры. WFF исчисления доменов

```
СЛУЖАЩИЕ (СЛУ_НОМ:2934, СЛУ_ИМЯ:'Иванов',
   СЛУ_ЗАРП:22400.00, ПРО_НОМ:1)
```

примет значение *true* в том и только в том случае, когда в теле отношения СЛУЖАЩИЕ содержится кортеж <2934, 'Иванов', 22400.00, 1>. Соответствующие значения доменных переменных образуют область истинности этой WFF. С другой стороны, WFF

```
СЛУЖАЩИЕ (СЛУ_НОМ:2934, СЛУ_ИМЯ:'Иванов',
   СЛУ_ЗАРП:22400.00, ПРО_НОМ:ПРО_НОМ)
```

будет принимать значение *true* для всех комбинаций явно заданных значений и допустимых значений переменной ПРО_НОМ, которые соответствуют кортежам, входящим в тело отношения СЛУЖАЩИЕ. При наличии тела отношения СЛУЖАЩИЕ, показанного на рис. 5.1, областью истинности этой WFF являются два следующих набора значений доменных переменных: <2934, 'Иванов', 22400.00, 1> и <2934, 'Иванов', 22400.00, 2>.

Выражения исчисления доменов

Во всех остальных отношениях формулы и выражения исчисления доменов выглядят похожими на формулы и выражения исчисления кортежей. В частности, формулы могут включать кванторы, и различаются свободные и связанные вхождения доменных переменных.

Для примера выражения исчисления доменов сформулируем с использованием исчисления доменов запрос «Выдать номера и имена служащих, не получающих минимальную заработную плату»:

```
СЛУ_НОМ, СЛУ_ИМЯ WHERE EXISTS СЛУ_ЗАРП1
(СЛУЖАЩИЕ (СЛУ_ЗАРП1) AND
   СЛУЖАЩИЕ (СЛУ_НОМ, СЛУ_ИМЯ, СЛУ_ЗАРП) AND
   СЛУ_ЗАРП > СЛУ_ЗАРП1)
```

Реляционное исчисление доменов является основой большинства языков запросов, основанных на использовании форм. В частности, на этом исчислении базировался известный язык Query-by-Example, который был первым (и наиболее интересным) языком в семействе языков, основанных на табличных формах.

Заключение

Этой лекцией мы завершаем обзор реляционной модели данных. В последних трех лекциях рассматривалась манипуляционная составляющая реляционной модели данных. Были представлены два варианта реляционной алгебры. Конечно, с формальной точки зрения можно было бы обойтись одним из вариантов, поскольку их выразительные средства эквивалентны. Но алгебра Кодда в большей степени базируется на теории множеств. Базовыми операциями являются переименование атрибутов, объединение, пересечение, взятие разности, декартово произведение, проекция и ограничение. Операция соединения общего вида, хотя и включается в алгебру, является вторичной и явно представляется через другие операции. Фундаментальная же в реляционном подходе операция естественного соединения выражается через соединение общего вида и в алгебру не включается. В терминах алгебры Кодда проще всего определяются алгебраические черты языка SQL, в частности общая семантика оператора `SELECT` (см. лекцию 12).

Базисом Алгебры А являются операции реляционного отрицания (дополнения), реляционной конъюнкции (или дизъюнкции) и проекции (удаления атрибута). Реляционные аналоги логических операций определяются в терминах отношений на основе обычных теоретико-множественных операций и позволяют выражать напрямую операции пересечения, декартова произведения, естественного соединения, объединения отношений. Путем комбинирования базовых операций выражаются операции переименования атрибутов, соединения общего вида, взятия разности отношений. Алгебра А позволяет лучше осознать логические основы реляционной модели, хотя, безусловно, является в меньшей степени ориентированной на практическое применение, чем алгебра Кодда.

Реляционному исчислению мы отвели меньше места, поскольку не ставили перед собой задачу определить какой-либо полноценный логический язык запросов. Цель состояла в том, чтобы показать возможность декларативной логической формулировки запросов. В этом случае выполнение запроса происходит путем интерпретации логической формулы, а не вычисления алгебраического выражения. Были рассмотрены два варианта реляционного исчисления, первый из которых – реляционное исчисление кортежей – был определен сравнительно полно, а для второго – реляционного исчисления доменов – были только отмечены и проиллюстрированы основные отличительные черты.

Лекция 6. Элементы теории реляционных баз данных: функциональные зависимости и декомпозиция без потерь

Эта и две следующие лекции посвящены вопросам теории реляционных баз данных. Поскольку все направление реляционного подхода к организации баз данных является сугубо практическим, эта теория, главным образом, прагматическая. Основная проблема, на решение которой направлена теория реляционных баз данных, состоит в обнаружении полезных свойств некоторых схем баз данных и выработке способов построения таких схем. Принято кратко называть эту проблему *проблемой проектирования реляционных баз данных*.

Ключевые слова: теория реляционных баз данных, проектирование реляционных баз данных, функциональная зависимость, многозначная зависимость, зависимость соединения, детерминант, тривиальная функциональная зависимость, замыкание множества функциональных зависимостей, транзитивная функциональная зависимость, правила вывода функциональных зависимостей, аксиомы Армстронга, замыкание множества атрибутов над множеством функциональных зависимостей, суперключ отношения, покрытие множества функциональных зависимостей, минимальное множество функциональных зависимостей, минимальное покрытие множества функциональных зависимостей, декомпозиция без потерь, теорема Хита, минимально зависимые атрибуты, диаграммы функциональных зависимостей.

Введение

Несмотря на свою практическую ориентированность, теория реляционных баз данных является самостоятельным научным направлением, в котором работали (и продолжают работать) многие известные исследователи, чьи имена будут встречаться в наших лекциях. Мы не планировали в данном курсе подробно описывать основные результаты в области теории реляционных баз данных. Наша цель состоит в том, чтобы обеспечить только определения и утверждения, необходимые для общего понимания процесса проектирования реляционных баз данных на основе нормализации.

Поскольку наиболее важные с практической точки зрения свойства реляционных баз данных базируются на понятии *функциональной зависимости*, мы выделили в отдельную лекцию краткое обсуждение соответствующих теоретических вопросов. Среди этих вопросов наибольший интерес представляют замыкания и покрытия множеств функциональных зависимостей, аксиомы Армстронга и теорема Хита

о достаточном условии декомпозиции отношения без потерь. Понятия и утверждения данной лекции действительно нужны для усвоения материала лекции 7, но мы стремились еще и продемонстрировать читателям на несложных примерах, что собой представляет теория реляционных баз данных, каков уровень ее сложности и насколько она понятна интуитивно.

Заметим, что мы не выделяли в отдельные лекции теоретический материал, касающийся *многозначных зависимостей* и *зависимостей соединения*. Это было сделано по двум причинам. Во-первых, эти виды зависимостей реже встречаются при моделировании предметной области средствами баз данных. Поэтому мы сочли достаточным представить внутри лекции 8 только основы соответствующего теоретического материала. Во-вторых, хотя теория многозначных зависимостей и зависимостей соединения, по сути, не намного сложнее теории функциональных зависимостей, ее определения и утверждения слишком громоздки для данного курса.

Функциональные зависимости

Наиболее важные с практической точки зрения нормальные формы отношений основываются на фундаментальном в теории реляционных баз данных понятии *функциональной зависимости*. Для дальнейшего изложения нам потребуется несколько определений и утверждений (по ходу изложения мы будем пояснять их и иллюстрировать).

Общие определения

Пусть задана переменная отношения `r`, и `X` и `Y` являются произвольными подмножествами заголовка `r` («составными» атрибутами).

Определение 6.1. Функциональная зависимость

В значении переменной отношения `r` атрибут `Y` *функционально зависит* от атрибута `X` в том и только в том случае, если каждому значению `X` соответствует в точности одно значение `Y`. В этом случае говорят также, что атрибут `X` *функционально определяет* атрибут `Y` (`X` является *детерминантом (определителем)* для `Y`, а `Y` является зависимым от `X`). Будем обозначать это как `r.X→r.Y`. **Конец определения.**

Для примера будем использовать отношение СЛУЖАЩИЕ_ПРОЕКТЫ {СЛУ_НОМ, СЛУ_ИМЯ, СЛУ_ЗАРП, ПРО_НОМ, ПРОЕКТ_РУК} (рис. 6.1). Очевидно, что если СЛУ_НОМ является первичным ключом отношения СЛУЖАЩИЕ, то для этого отношения справедлива функциональная зависимость (Functional Dependency – FD) СЛУ_НОМ→СЛУ_ИМЯ.

На самом деле, для тела отношения СЛУЖАЩИЕ_ПРОЕКТЫ в том виде, в котором оно показано на рис. 6.1, выполняются еще и следующие FD (1):

СЛУ_НОМЕР	СЛУ_ИМЯ	СЛУ_ЗАРП	ПРО_НОМ	ПРОЕКТ_РУК
2934	Иванов	22400.00	1	Иванов
2935	Петров	29600.00	1	Иванов
2936	Сидоров	18000.00	1	Иванов
2937	Федоров	20000.00	1	Иванов
2938	Иванова	22000.00	1	Иванов
2939	Сидоренко	18400.00	2	Иваненко
2940	Федоренко	20400.00	2	Иваненко
2941	Иваненко	22600.00	2	Иваненко

Рис. 6.1. Пример возможного тела отношения СЛУЖАЩИЕ_ПРОЕКТЫ

СЛУ_НОМ→СЛУ_ИМЯ
СЛУ_НОМ→СЛУ_ЗАРП
СЛУ_НОМ→ПРО_НОМ
СЛУ_НОМ→ПРОЕКТ_РУК
{СЛУ_НОМ, СЛУ_ИМЯ}→СЛУ_ЗАРП
{СЛУ_НОМ, СЛУ_ИМЯ}→ПРО_НОМ
{СЛУ_НОМ, СЛУ_ИМЯ}→{СЛУ_ЗАРП, ПРО_НОМ}
...
ПРО_НОМ→ПРОЕКТ_РУК и т.д.

Поскольку имена всех служащих различны, то выполняются и такие FD (2):

СЛУ_ИМЯ→СЛУ_НОМ
СЛУ_ИМЯ→СЛУ_ЗАРП
СЛУ_ИМЯ→ПРО_НОМ и т.д.

Более того, для примера на рис. 6.1 выполняется и FD (3):

СЛУ_ЗАРП→ПРО_НОМ

Однако заметим, что природа FD группы (1) отличается от природы FD групп (2) и (3). Логично предположить, что идентификационные номера служащих должны быть всегда различны, а у каждого проекта имеется только один руководитель. Поэтому FD группы (1) должны быть верны для любого допустимого значения переменной отношения СЛУЖАЩИЕ_ПРОЕКТЫ и могут рассматриваться как *инварианты*, или *ограничения целостности* этой переменной отношения.

FD группы (2) базируются на менее естественном предположении о том, что имена всех служащих различны. Это соответствует действительности для примера из рис. 6.1, но возможно, что с течением времени FD

группы (2) не будут выполняться для какого-либо значения переменной отношения СЛУЖАЩИЕ_ПРОЕКТЫ.

Наконец, FD группы (3) основана на совсем неестественном предположении, что никакие двое служащих, участвующие в разных проектах, не получают одинаковую зарплату. Опять же, данное предположение верно для примера из рис. 6.1, но, скорее всего, это случайное совпадение.

В дальнейшем нас будут интересовать только те функциональные зависимости, которые должны выполняться для всех возможных значений переменных отношений.

Заметим, что если атрибут A отношения r является возможным ключом, то для любого атрибута B этого отношения всегда выполняется FD $A \rightarrow B$ (в группе (1) к этим FD относятся все FD, детерминантом которых является СЛУ_НОМ). Обратите внимание, что наличие в отношении СЛУЖАЩИЕ_ПРОЕКТЫ FD ПРО_НОМ→ПРОЕКТ_РУК приводит к некоторой *избыточности* этого отношения. Имя руководителя проекта является характеристикой проекта, а не служащего, но в нашем случае содержится в теле отношения столько раз, сколько служащих работает над проектом.

Итак, мы будем иметь дело с FD, которые выполняются для всех возможных состояний тела соответствующего отношения и могут рассматриваться как ограничения целостности. Как показывает (неполный) список (1), таких зависимостей может быть очень много. Поскольку они трактуются как ограничения целостности, за их соблюдением должна следить СУБД. Поэтому важно уметь сократить набор FD до минимума, поддержка которого гарантирует выполнение всех зависимостей. Мы займемся этим в следующих подразделах.

Определение 6.2. Тривиальная функциональная зависимость

FD $A \rightarrow B$ называется тривиальной, если $A \supseteq B$ (т. е. множество атрибутов A включает множество B или совпадает с множеством B). **Конец определения.**

Очевидно, что любая тривиальная FD всегда выполняется. Например, в отношении СЛУЖАЩИЕ_ПРОЕКТЫ всегда выполняется FD {СЛУ_ЗАРП, ПРО_НОМ}→СЛУ_ЗАРП. Частным случаем тривиальной FD является $A \rightarrow A$.

Поскольку тривиальные FD выполняются всегда, их нельзя трактовать как ограничения целостности, и поэтому они не представляют интереса с практической точки зрения. Однако в теоретических рассуждениях их наличие необходимо учитывать.

Замыкание множества функциональных зависимостей. Аксиомы Армстронга. Замыкание множества атрибутов

Определение 6.3. Замыкание множества FD

Замыканием множества FD S является множество FD S^+, включающее все FD, логически выводимые из FD множества S. **Конец определения.**

Для начала приведем два примера FD, из которых следуют (или *выводятся*) другие FD. Будем снова пользоваться отношением СЛУЖАЩИЕ_ПРОЕКТЫ. Для этого отношения выполняется, например, FD СЛУ_НОМ→{СЛУ_ЗАРП, ПРО_НОМ}. Из этой FD выводятся FD СЛУ_НОМ→СЛУ_ЗАРП и СЛУ_НОМ→ПРО_НОМ.

В отношении СЛУЖАЩИЕ_ПРОЕКТЫ имеется также пара FD СЛУ_НОМ→ПРО_НОМ и ПРО_НОМ→ПРОЕКТ_РУК. Из них выводится FD СЛУ_НОМ→ПРОЕКТ_РУК. Заметим, что FD вида СЛУ_НОМ→ПРОЕКТ_РУК называются *транзитивными*, поскольку ПРОЕКТ_РУК зависит от СЛУ_НОМ «транзитивно», через ПРО_НОМ.

Определение 6.4. Транзитивная функциональная зависимость FD $A \rightarrow C$ называется транзитивной, если существует такой атрибут B, что имеются функциональные зависимости $A \rightarrow B$ и $B \rightarrow C$ и отсутствует функциональная зависимость $C \rightarrow A$. **Конец определения.**

Подход к решению проблемы поиска замыкания S^+ множества FD S впервые предложил Вильям Армстронг*. Им был предложен набор *правил вывода* новых FD из существующих (эти правила обычно называют *аксиомами Армстронга*, хотя справедливость правил доказывается на основе определения FD). Обычно принято формулировать эти правила вывода в следующей форме. Пусть A, B и C являются (в общем случае, составными) атрибутами отношения r. Множества A, B и C могут иметь непустое пересечение. Для краткости будем обозначать через AB A UNION B. Тогда:

1) если $B \subseteq A$, то $A \rightarrow B$ (рефлексивность);
2) если $A \rightarrow B$, то $AC \rightarrow BC$ (пополнение);
3) если $A \rightarrow B$ и $B \rightarrow C$, то $A \rightarrow C$ (транзитивность).

Истинность первой аксиомы Армстронга следует из того, что при $B \subseteq A$ FD $A \rightarrow B$ является тривиальной.

Справедливость второй аксиомы докажем от противного. Предположим, что FD $AC \rightarrow BC$ не соблюдается. Это означает, что в некотором допустимом теле отношения найдутся два кортежа $t1$ и $t2$, такие, что $t1\ \{AC\} = t2\ \{AC\}$ (a), но $t1\ \{BC\} \neq t2\ \{BC\}$ (b) (здесь $t\ \{A\}$ обозначает проекцию кортежа t на множество атрибутов A). По аксиоме рефлексивности из равенства (a) следует, что $t1\ \{A\} = t2\ \{A\}$. Поскольку имеется FD $A \rightarrow B$, должно соблюдаться равенство $t1\ \{B\} = t2\ \{B\}$. Тогда из неравенства (b) следует, что $t1\ \{C\} \neq t2\ \{C\}$, что противоречит наличию тривиальной FD $AC \rightarrow C$. Следовательно, предположение об отсутствии FD $AC \rightarrow BC$ не является верным, и справедливость второй аксиомы доказана.

Аналогично докажем истинность третьей аксиомы Армстронга. Предположим, что FD $A \rightarrow C$ не соблюдается. Это означает, что в некото-

* К сожалению, классическая статья Армстронга – *W.W. Armstrong. «Dependency Structures of Data Base Relationships», Proc. IFIP Congress, Stockholm, Sweden, 1974* – так и не переведена на русский язык (на самом деле, ее нелегко найти и в оригинале). Поэтому я не могу рекомендовать ее для дополнительного чтения, хотя обязан сослаться.

ром допустимом теле отношения найдутся два кортежа *t1* и *t2*, такие, что *t1* {*A*} = *t2* {*A*}, но *t1* {*C*} ≠ *t2* {*C*}. Но из наличия FD *A*→*B* следует, что *t1* {*B*} = *t2* {*B*}, а потому из наличия FD *B*→*C* следует, что *t1* {*C*} = *t2* {*C*}. Следовательно, предположение об отсутствии FD *A*→*C* не является верным, и справедливость третьей аксиомы доказана.

Можно доказать, что система правил вывода Армстронга *полна* и *совершенна (sound and complete)* в том смысле, что для данного множества FD *S* любая FD, потенциально выводимая из *S*, может быть выведена на основе аксиом Армстронга, и применение этих аксиом не может привести к выводу лишней FD. Тем не менее, Дейт по практическим соображениям предложил расширить базовый набор правил вывода еще пятью правилами:

(4) *A*→*A* (самодетерминированность) – прямо следует из правила (1);

(5) если *A*→*BC*, то *A*→*B* и *A*→*C* (декомпозиция) – из правила (1) следует, что *BC*→*B*; по правилу (3) *A*→*B*; аналогично, из *BC*→*C* и правила (3) следует *A*→*C*;

(6) если *A*→*B* и *A*→*C*, то *A*→*BC* (объединение) – из правила (2) следует, что *A*→*AB* и *AB*→*BC*; из правила (3) следует, что *A*→*BC*;

(7) если *A*→*B* и *C*→*D*, то *AC*→*BD* (композиция) – из правила (2) следует, что *AC*→*BC* и *BC*→*BD*; из правила (3) следует, что *AC*→*BD*;

(8) если *A*→*BC* и *B*→*D*, то *A*→*BCD* (накопление) – из правила (2) следует, что *BC*→*BCD*; из правила (3) следует, что *A*→*BCD*.

Определение 6.5. Замыкание множества атрибутов

Пусть заданы отношение *r*, множество *Z* атрибутов этого отношения (подмножество заголовка *r*, или составной атрибут *r*) и некоторое множество FD S, выполняемых для *r*. Тогда *замыканием Z над S* называется наибольшее множество Z^+ таких атрибутов *Y* отношения *r*, что FD *Z*→*Y* входит в S^+. **Конец определения.**

Алгоритм вычисления Z^+ очень прост. Один из его вариантов показан на рис. 6.2.

```
K := 0; Z[0] := Z;
DO
   K := K+1;
   Z[K] := Z[K-1];
   FOR EACH FD A→B IN S DO
      IF A⊆Z[K] THEN Z[K] := (Z[K] UNION B) END DO;
UNTIL Z[K] = Z[K-1];
Z+ := Z[K];
```

Рис. 6.2. Алгоритм построения замыкания атрибутов над заданным множеством FD

Докажем корректность алгоритма по индукции. На нулевом шаге Z[0] = Z, FD Z→Z[I], очевидно, принадлежит S^+ (тривиальная FD «выводится» из любого множества FD). Пусть для некоторого K выполняется FD Z→Z[K], и пусть мы нашли в S такую FD A→B, что A⊆Z[K]. Тогда можно представить Z[K] в виде AC, и, следовательно, выполняется FD Z→AC. Но по правилу (8) мы имеем FD Z→ACB, т.е. FD Z→(Z[K] UNION B) входит во множество S^+, что переводит нас на следующий шаг индукции.

Пусть для примера имеется отношение с заголовком {A, B, C, D, E, F} и заданным множеством FD S = {A→D, AB→E, BF→E, CD→F, E→C}. Пусть требуется найти $\{AE\}^+$ над S. На первом проходе тела цикла DO Z[1] равно AE. В теле цикла FOR EACH будут найдены FD A→D и E→C, и в конце цикла Z[1] станет равным ACDE. На втором проходе тела цикла DO при Z[2], равном ACDE, в теле цикла FOR EACH будет найдена FD CD→F, и в конце цикла Z[2] станет равным ACDEF. Следующий проход тела цикла DO не изменит Z[3], и Z^+ ($\{AE\}^+$) будет равно ACDEF.

Алгоритм построения замыкания множества атрибутов Z над заданным множеством FD S помогает легко установить, входит ли заданная FD Z→B в замыкание S^+. Очевидно, что необходимым и достаточным условием для этого является B⊂Z^+, т. е. вхождение составного атрибута B в замыкание Z.*

Определение 6.6. Суперключ отношения

Суперключом отношения r называется любое подмножество K заголовка r, включающее, по меньшей мере, хотя бы один возможный ключ r. **Конец определения.**

Одно из следствий этого определения состоит в том, что подмножество K заголовка отношения r является суперключом тогда и только тогда, когда для любого атрибута A (возможно, составного) заголовка отношения r выполняется FD K→A. В терминах замыкания множества атрибутов K является суперключом тогда и только тогда, когда K^+ совпадает с заголовком r.

Минимальное покрытие множества функциональных зависимостей

Определение 6.7. Покрытие множества FD

Множество FD S2 называется *покрытием* множества FD S1, если любая FD, выводимая из S1, выводится также из S2. **Конец определения.**

Легко заметить, что S2 является покрытием S1 тогда и только тогда, когда $S1^+ \subseteq S2^+$. Два множества FD S1 и S2 называются *эквивалентными*, если каждое из них является покрытием другого, т. е. $S1^+ = S2^+$.

* Мы используем здесь знаки операций проверки включения множеств, что не совсем корректно, поскольку если, например, множество *B* состоит из одного элемента, то для его обозначения используется имя соответствующего атрибута, и в этом случае правильнее было бы использовать знак «∈» (проверка вхождения элемента во множество).

Определение 6.8. Минимальное множество FD

Множество FD S называется *минимальным* в том и только в том случае, когда удовлетворяет следующим свойствам:

a) правая часть любой FD из S является множеством из одного атрибута (простым атрибутом);

b) детерминант каждой FD из S обладает свойством *минимальности*; это означает, что удаление любого атрибута из детерминанта приводит к изменению замыкания S^+, т. е. порождению множества FD, не эквивалентного S;*

c) удаление любой FD из S приводит к изменению S^+, т. е. порождению множества FD, не эквивалентного S. **Конец определения**.

Чтобы продемонстрировать минимальные и неминимальные множества FD, вернемся к примеру отношения СЛУЖАЩИЕ_ПРОЕКТЫ {СЛУ_НОМ, СЛУ_ИМЯ, СЛУ_ЗАРП, ПРО_НОМ, ПРОЕКТ_РУК} с рис. 6.1. Если считать, что единственным возможным ключом этого отношения является атрибут СЛУ_НОМ, то множество FD {СЛУ_НОМ→СЛУ_ИМЯ, СЛУ_НОМ→СЛУ_ЗАРП, СЛУ_НОМ→ПРО_НОМ, ПРО_НОМ→ПРОЕКТ_РУК} будет минимальным. Действительно, в правых частях FD этого множества находятся множества, состоящие ровно из одного атрибута; каждый из детерминантов тоже является множеством из одного атрибута, удаление которого, очевидно, недопустимо; удаление каждой FD явно приводит к изменению замыкания множества FD, поскольку утрачиваемая информация не выводится с помощью аксиом Армстронга.

С другой стороны, множества FD

(*) {СЛУ_НОМ→{СЛУ_ИМЯ, СЛУ_ЗАРП}, СЛУ_НОМ→ПРО_НОМ, СЛУ_НОМ→ПРОЕКТ_РУК, ПРО_НОМ→ПРОЕКТ_РУК},

(**) {СЛУ_НОМ→СЛУ_ИМЯ, {СЛУ_НОМ, СЛУ_ИМЯ}→СЛУ_ЗАРП, СЛУ_НОМ→ПРО_НОМ, СЛУ_НОМ→ПРОЕКТ_РУК, ПРО_НОМ→ПРОЕКТ_РУК} и

(***) {СЛУ_НОМ→СЛУ_НОМ, СЛУ_НОМ→СЛУ_ИМЯ, СЛУ_НОМ→СЛУ_ЗАРП, СЛУ_НОМ→ПРО_НОМ, СЛУ_НОМ→ПРОЕКТ_РУК, ПРО_НОМ→ПРОЕКТ_РУК}

не являются минимальными. Для множества (*) в правой части первой FD присутствует множество из двух элементов. Для множества (**) удаление атрибута СЛУ_ИМЯ из детерминанта второй FD не меняет замыкание множества FD. Для множества (***) удаление первой FD не приводит к изменению замыкания. Эти примеры показывают, что для определения минимальности множества FD не всегда требуется явное построение замыкания данного множества.

Интересным и важным является тот факт, что *для любого множества FD S существует (и даже может быть построено) эквивалентное ему минимальное множество S^-*.

* FD с минимальным детерминантом называется *минимальной слева*.

Приведем общую схему построения S^- по заданному множеству FD S. Во-первых, используя правило (5) (декомпозиции), мы можем привести множество S к эквивалентному множеству FD $S1$, правые части FD которого содержат только одноэлементные множества (простые атрибуты). Далее, для каждой FD из $S1$, детерминант D $\{D_1, D_2, \ldots, D_n\}$ которой содержит более одного атрибута, будем пытаться удалять атрибуты D_i, получая множество FD $S2$. Если после удаления атрибута D_i $S2$ эквивалентно $S1$, то этот атрибут удаляется, и пробуется следующий атрибут. Назовем $S3$ множество FD, полученное путем допустимого удаления атрибутов из всех детерминантов FD множества $S1$. Наконец, для каждой FD f из множества $S3$ будем проверять эквивалентность множеств $S3$ и $S3$ MINUS $\{f\}$. Если эти множества эквивалентны, удалим f из множества $S3$, и в заключение получим множество $S4$, которое минимально и эквивалентно исходному множеству FD S.

Пусть, например, имеется отношение r $\{A, B, C, D\}$ и задано множество FD $S = \{A \rightarrow B, A \rightarrow BC, AB \rightarrow C, AC \rightarrow D, B \rightarrow C\}$. По правилу декомпозиции S эквивалентно множеству $S1$ $\{A \rightarrow B, A \rightarrow C, AB \rightarrow C, AC \rightarrow D, B \rightarrow C\}$. В детерминанте FD $AC \rightarrow D$ можно удалить атрибут C, поскольку по правилу дополнения из FD $A \rightarrow C$ следует $A \rightarrow AC$; по правилу транзитивности выводится FD $A \rightarrow D$, поэтому атрибут C в детерминанте FD $AC \rightarrow D$ является избыточным. FD $AB \rightarrow C$ может быть удалена, поскольку может быть выведена из FD $A \rightarrow C$ (по правилу пополнения из этой FD выводится $AB \rightarrow BC$, а по правилу декомпозиции далее выводится $AB \rightarrow C$). Наконец, FD $A \rightarrow C$ тоже выводится по правилу транзитивности из FD $A \rightarrow B$ и $B \rightarrow C$. Таким образом, мы получаем множество зависимостей $\{A \rightarrow B, A \rightarrow D, B \rightarrow C\}$, которое является минимальным и эквивалентно S по построению.

Определение 6.9. Минимальное покрытие множества FD

Минимальным покрытием множества FD S называется любое минимальное множество FD $S1$, эквивалентное S. **Конец определения**.

Поскольку для каждого множества FD существует эквивалентное минимальное множество FD, у каждого множества FD имеется хотя бы одно минимальное покрытие, причем для его нахождения не обязательно находить замыкание исходного множества.

Декомпозиция без потерь и функциональные зависимости

Как уже отмечалось, в следующей лекции мы будем обсуждать подход к проектированию реляционных баз данных на основе нормализации, т. е. декомпозиции (разбиения путем проецирования) отношения, находящегося в предыдущей нормальной форме, на два или более отношений, удовлетворяющих требованиям следующей нормальной формы.

Считаются правильными такие декомпозиции отношения, которые обратимы, т. е. имеется возможность собрать исходное отношение из декомпозированных отношений без потери информации. Такие декомпозиции называются *декомпозициями без потерь*.

Корректные и некорректные декомпозиции отношений. Теорема Хита

На рис. 6.3 приведены две возможные декомпозиции отношения СЛУЖАЩИЕ_ПРОЕКТЫ (для экономии места мы сократили и слегка изменили тело отношения из рис. 6.1).

СЛУЖАЩИЕ_ПРОЕКТЫ

СЛУ_НОМЕР	СЛУ_ИМЯ	СЛУ_ЗАРП	ПРО_НОМ	ПРОЕКТ_РУК
2934	Иванов	22000.00	1	Иванов
2941	Иваненко	22000.00	2	Иваненко

Декомпозиция (1). Отношения **СЛУЖ** и **СЛУ_ПРО**

СЛУ_НОМ	СЛУ_ИМЯ	СЛУ_ЗАРП
2934	Иванов	22000.00
2941	Иваненко	22000.00

СЛУ_НОМ	ПРО_НОМ	ПРОЕКТ_РУК
2934	1	Иванов
2941	2	Иваненко

Декомпозиция (2). Отношения **СЛУЖ** и **ЗАРП_ПРО**

СЛУ_НОМ	СЛУ_ИМЯ	СЛУ_ЗАРП
2934	Иванов	22000.00
2941	Иваненко	22000.00

СЛУ_ЗАРП	ПРО_НОМ	ПРОЕКТ_РУК
22000.00	1	Иванов
22000.00	2	Иваненко

Рис. 6.3. Две возможные декомпозиции отношения СЛУЖАЩИЕ_ПРОЕКТЫ

Анализ рис. 6.3 показывает, что в случае декомпозиции (1) мы не потеряли информацию о служащих – про каждого из них можно узнать имя, размер зарплаты, номер выполняемого проекта и имя руководителя проекта. Вторая декомпозиция не дает возможности получить данные о проекте служащего, поскольку Иванов и Иваненко получают одинаковую зарплату; следовательно, эта декомпозиция приводит к потере информации. Что же привело к тому, что одна декомпозиция является декомпозицией без потерь, а вторая – нет?

Заметим, что при проведении декомпозиции мы использовали операцию взятия проекции. Каждое из отношений СЛУЖ, СЛУ_ПРО и ЗАРП_ПРО является проекцией исходного отношения СЛУЖАЩИЕ_ПРОЕКТЫ. В случае декомпозиции (1) отсутствие потери информации означает, что в результате естественного соединения отношений СЛУЖ и СЛУ_ПРО мы гарантированно получим отношение, заголовок и тело которого совпадают с заголовком и телом отношения СЛУЖАЩИЕ_ПРОЕКТЫ. Следует отметить, что это произойдет для любых допустимых (и согласованных) значений переменных отношений СЛУЖАЩИЕ_ПРОЕКТЫ, СЛУЖ и СЛУ_ПРО, поскольку у всех этих переменных атрибут СЛУ_НОМ является возможным ключом. Однако если выполнить естественное соединение отношений СЛУ и ЗАРП_ПРО, то будет получено отношение, показанное на рис. 6.4.

Схема этого отношения, естественно (поскольку соединение – естественное), совпадает со схемой отношения СЛУЖАЩИЕ_ПРОЕКТЫ, но в теле появились лишние кортежи, наличие которых и приводит к утрате исходной информации. Интуитивно понятно, что это происходит потому, что в отношении ЗАРП_ПРО отсутствуют функциональные зависимости СЛУ_ЗАРП→ПРО_НОМ и СЛУ_ЗАРП→ПРОЕКТ_РУК, но точнее причину потери информации в данном случае мы объясним несколько позже.

Корректность же декомпозиции 1 следует из теоремы Хита:

Теорема Хита.

Пусть задано отношение r {A, B, C} (A, B и C, в общем случае, являются составными атрибутами) и выполняется FD $A \rightarrow B$.

СЛУ_НОМ	СЛУ_ИМЯ	СЛУ_ЗАРП	ПРО_НОМ	ПРОЕКТ_РУК
2934	Иванов	22000.00	1	Иванов
2941	Иваненко	22000.00	2	Иваненко
2934	Иванов	22000.00	2	Иваненко
2941	Иваненко	22000.00	1	Иванов

Рис. 6.4. Результат естественного соединения отношений СЛУЖ и ЗАРП_ПРО

Тогда *r* = (*r* PROJECT {*A*, *B*}) NATURAL JOIN (*r* PROJECT {*A*, *C*}).

Доказательство. Прежде всего, докажем, что в теле результата естественного соединения (обозначим этот результат через *r1*) содержатся все кортежи тела отношения *r*. Действительно, пусть кортеж {*a*, *b*, *c*} ∈ *r*. Тогда по определению операции взятия проекции {*a*, *b*} ∈ (*r* PROJECT {*A*, *B*}) и {*a*, *c*} ∈ (*r* PROJECT {*A*, *C*}). Следовательно, {*a*, *b*, *c*} ∈ *r1*. Теперь докажем, что в теле результата естественного соединения нет лишних кортежей, т. е. что если кортеж {*a*, *b*, *c*} ∈ *r1*, то {*a*, *b*, *c*} ∈ *r*. Если {*a*, *b*, *c*} ∈ *r1*, то существуют {*a*, *b*} ∈ (*r* PROJECT {*A*, *B*}) и {*a*, *c*} ∈ (*r* PROJECT {*A*, *C*}). Последнее условие может выполняться в том и только в том случае, когда существует кортеж {*a*, *b**, *c*} ∈ *r*. Но поскольку выполняется FD *A*→*B*, то *b* = *b** и, следовательно, {*a*, *b*, *c*} = {*a*, *b**, *c*}. **Конец доказательства.**

Для иллюстрации общего случая применения теоремы Хита рассмотрим отношение СЛУЖАЩИЕ_ОТДЕЛЫ_ПРОЕКТЫ {СЛУ_НОМ, СЛУ_ОТД, ПРО_НОМ} (рис. 6.5). Атрибут СЛУ_ОТД содержит номера отделов, в которых работают служащие, а ПРО_НОМ – номера проектов, в которых служащие

СЛУЖАЩИЕ_ОТДЕЛЫ_ПРОЕКТЫ

СЛУ_НОМ	СЛУ_ОТД	ПРО_НОМ
2934	630	1
2941	631	1
2934	630	2
2941	631	2

Декомпозиция.
Отношения **СЛУЖ_ОТДЕЛЫ** и **СЛУЖ_ПРОЕКТЫ**

СЛУ_НОМ	СЛУ_ОТД
2934	630
2941	631

СЛУ_НОМ	ПРО_НОМ
2934	1
2941	1
2934	2
2941	2

Рис. 6.5. Декомпозиция без потерь по теореме Хита

принимают участие. Каждый служащий работает только в одном отделе, т. е. имеется FD СЛУ_НОМ→СЛУ_ОТД, но один служащий может участвовать в нескольких проектах.

В отношении СЛУЖАЩИЕ_ОТДЕЛЫ_ПРОЕКТЫ атрибут СЛУ_НОМ не является возможным ключом, но, как показано на рис. 6.5, наличия FD СЛУ_НОМ→СЛУ_ОТД оказывается достаточно для декомпозиции этого отношения без потерь.

Для дальнейшего изложения нам потребуется ввести еще одно определение и сделать пару замечаний.

Определение 6.10. Минимально зависимые атрибуты

Атрибут *B* минимально зависит от атрибута *A*, если выполняется минимальная слева FD *A*→*B*. **Конец определения**.

Например, в отношении СЛУЖАЩИЕ_ПРОЕКТЫ выполняются FD СЛУ_НОМ→СЛУ_ЗАРП и {СЛУ_НОМ, СЛУ_ИМЯ}→СЛУ_ЗАРП. Первая FD является минимальной слева, а вторая – нет. Поэтому СЛУ_ЗАРП минимально зависит от СЛУ_НОМ, а для {СЛУ_НОМ, СЛУ_ИМЯ} свойство минимальной зависимости не выполняется.

Диаграммы функциональных зависимостей

Далее, для иллюстраций в следующей лекции нам пригодятся диаграммы FD, с помощью которых можно наглядно представлять минимальные множества FD. Например, на рис. 6.6 приведена диаграмма минимального множества FD отношения СЛУЖАЩИЕ_ПРОЕКТЫ.

В левой части диаграммы все стрелки начинаются с атрибута СЛУ_НОМ, который является единственным возможным (и, следовательно, первичным) ключом отношения СЛУЖАЩИЕ_ПРОЕКТЫ. Обратите внимание на отсутствие стрелки от СЛУ_НОМ к ПРОЕКТ_РУК. Конечно, поскольку СЛУ_НОМ является возможным ключом, должна выполняться и FD СЛУ_НОМ→ПРОЕКТ_РУК. Но эта FD является транзитивной (через ПРО_НОМ)

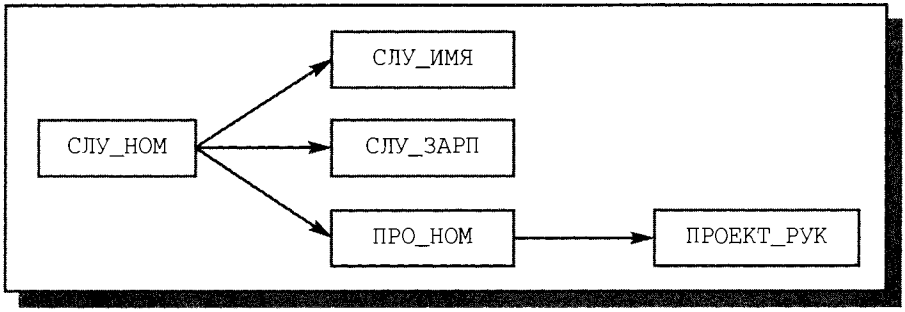


Рис. 6.6. Диаграмма минимального множества FD отношения СЛУЖАЩИЕ_ПРОЕКТЫ

и поэтому не входит в минимальное множество FD. Заметим, что в процессе нормализации, к рассмотрению которого мы приступим в следующей лекции, из диаграмм множества FD удаляются стрелки, начинающиеся не от возможных ключей.

Заключение

В этой лекции было введено понятие функциональной зависимости и исследовались важные свойства функциональных зависимостей. Одна из целей состояла в том, чтобы на основе некоторого множества функциональных зависимостей суметь построить минимальное эквивалентное множество функциональных зависимостей. Мы начали обсуждение с понятия замыканий множества функциональных зависимостей и аксиом Амстронга, теоретически позволяющих построить такое замыкание. Замыкание множества функциональных зависимостей содержит все функциональные зависимости, выводимые из функциональных зависимостей заданного множества. Рассмотренный далее алгоритм построения замыкания множества атрибутов над заданным множеством функциональных зависимостей упрощает задачу, позволяя определить принадлежность заданной функциональной зависимости к замыканию заданного множества функциональных зависимостей без потребности в реальном построении замыкания.

Далее мы занялись покрытиями множеств функциональных зависимостей и минимальными множествами функциональных зависимостей. Наиболее важным результатом этой части лекции является доказательство существования и наметки алгоритма построения минимального покрытия заданного множества функциональных зависимостей – минимального множества функциональных зависимостей, эквивалентного исходному множеству.

Наконец, последний раздел лекции был посвящен критерию декомпозиции отношения без потерь, т. е. такому способу проецирования заданного отношения на два отношения, при котором результат естественного соединения проекций в точности совпадает с исходным отношением. Достаточное (и очень естественное) условие декомпозиции без потерь обеспечивает теорема Хита.

Лекция 7. Проектирование реляционных баз данных на основе принципов нормализации: первые шаги нормализации

Эта лекция открывает серию из четырех лекций, посвященных проектированию реляционных баз данных. В данной лекции речь пойдет о нормализации схем отношений с учетом только функциональных зависимостей между атрибутами отношений. Эти «первые шаги» нормализации позволяют получить схему базы данных, в которых все переменные отношений находятся в нормальной форме Бойса-Кодда, обычно расцениваемой удовлетворительной для большей части приложений.

Ключевые слова: первая нормальная форма, процесс нормализации, аномалии обновления, вторая нормальная форма, третья нормальная форма, независимые проекции отношений, теорема Риссанена, атомарное отношение, перекрывающиеся возможные ключи, нормальная форма Бойса-Кодда.

Введение

При проектировании базы данных решаются две основные проблемы.

- Каким образом отобразить объекты предметной области в абстрактные объекты модели данных, чтобы это отображение не противоречило семантике предметной области и было, по возможности, лучшим (эффективным, удобным и т. д.)? Часто эту проблему называют проблемой логического проектирования баз данных.
- Как обеспечить эффективность выполнения запросов к базе данных, т. е. каким образом, имея в виду особенности конкретной СУБД, расположить данные во внешней памяти, создания каких дополнительных структур (например, индексов) потребовать и т. д.? Эту проблему обычно называют проблемой физического проектирования баз данных.

В случае реляционных баз данных трудно предложить какие-либо общие рецепты по части физического проектирования. Здесь слишком многое зависит от используемой СУБД. Поэтому мы ограничимся вопросами логического проектирования реляционных баз данных, которые существенны при использовании любой реляционной СУБД.

Более того, мы не будем касаться очень важного аспекта проектирования – определения ограничений целостности общего вида (за исключением ограничений, задаваемых функциональными и многозначными зависимостями, а также зависимостями проекции/соединения). Дело в том, что при использовании СУБД с развитыми механизмами ограниче-

ний целостности (например, SQL-ориентированных систем) трудно предложить какой-либо универсальный подход к определению ограничений целостности. Эти ограничения могут иметь произвольно сложную форму, и их формулировка пока относится скорее к области искусства, чем инженерного мастерства. Самое большее, что предлагается по этому поводу в литературе, это автоматическая проверка непротиворечивости набора ограничений целостности.

Так что в этой и следующей лекциях мы будем считать, что проблема проектирования реляционной базы данных состоит в обоснованном принятии решений о том, из каких отношений должна состоять БД и какие атрибуты должны быть у этих отношений.

В этой и следующей лекциях будет рассмотрен классический подход, при котором весь процесс проектирования базы данных осуществляется в терминах реляционной модели данных методом последовательных приближений к удовлетворительному набору схем отношений. Исходной точкой является представление предметной области в виде одного или нескольких отношений, и на каждом шаге проектирования производится некоторый набор схем отношений, обладающих «улучшенными» свойствами. Процесс проектирования представляет собой процесс нормализации схем отношений, причем каждая следующая нормальная форма обладает свойствами, в некотором смысле, лучшими, чем предыдущая.

Каждой нормальной форме соответствует определенный набор ограничений, и отношение находится в некоторой нормальной форме, если удовлетворяет свойственному ей набору ограничений. Примером может служить ограничение первой нормальной формы – значения всех атрибутов отношения атомарны*. Поскольку требование первой нормальной формы является базовым требованием классической реляционной модели данных, мы будем считать, что исходный набор отношений уже соответствует этому требованию.

В теории реляционных баз данных обычно выделяется следующая последовательность нормальных форм:

- первая нормальная форма (1NF);
- вторая нормальная форма (2NF);
- третья нормальная форма (3NF);
- нормальная форма Бойса-Кодда (BCNF);
- четвертая нормальная форма (4NF);
- пятая нормальная форма, или нормальная форма проекции-соединения (5NF или PJ/NF).

Основные свойства нормальных форм состоят в следующем:

* Напомним из лекции 2, что *атомарность* значения трактуется в том смысле, что значение типизировано, и с этим значением можно работать только с помощью операций соответствующего типа данных.

- каждая следующая нормальная форма в некотором смысле лучше предыдущей нормальной формы;
- при переходе к следующей нормальной форме свойства предыдущих нормальных форм сохраняются.

В основе процесса проектирования лежит метод нормализации, т. е. декомпозиции отношения, находящегося в предыдущей нормальной форме, на два или более отношений, которые удовлетворяют требованиям следующей нормальной формы.

В этой лекции мы обсудим первые шаги процесса нормализации, в которых учитываются функциональные зависимости между атрибутами отношений. Хотя мы и называем эти шаги первыми, именно они имеют основную практическую важность, поскольку позволяют получить схему реляционной базы данных, в большинстве случаев удовлетворяющую потребности приложений.

Минимальные функциональные зависимости и вторая нормальная форма

Пусть имеется переменная отношения СЛУЖАЩИЕ_ПРОЕКТЫ_ЗАДАНИЯ {СЛУ_НОМ, СЛУ_УРОВ, СЛУ_ЗАРП, ПРО_НОМ, СЛУ_ЗАДАН}. Новые атрибуты СЛУ_УРОВ и СЛУ_ЗАДАН содержат, соответственно, данные о разряде служащего и о задании, которое выполняет служащий в данном проекте. Будем считать, что разряд служащего определяет размер его заработной платы, и что каждый служащий может участвовать в нескольких проектах, но в каждом проекте он выполняет только одно задание. Тогда очевидно, что единственно возможным ключом отношения СЛУЖАЩИЕ_ПРОЕКТЫ_ЗАДАНИЯ является составной атрибут {СЛУ_НОМ, ПРО_НОМ}. Диаграмма минимального множества FD показана на рис. 7.1, а возможное тело значения отношения – на рис. 7.2.

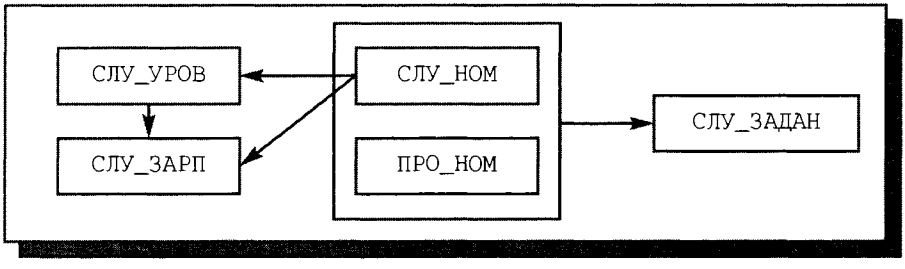


Рис. 7.1. Диаграмма множества FD отношения СЛУЖАЩИЕ_ПРОЕКТЫ_ЗАДАНИЯ

СЛУ_НОМ	СЛУ_УРОВ	СЛУ_ЗАРП	ПРО_НОМ	СЛУ_ЗАДАН
2934	2	22400.00	1	A
2935	3	29600.00	1	B
2936	1	20000.00	1	C
2937	1	20000.00	1	D
2934	2	22400.00	2	D
2935	3	29600.00	2	C
2936	1	20000.00	2	B
2937	1	20000.00	2	A

Рис. 7.2. Возможное значение переменной отношения СЛУЖАЩИЕ_ПРОЕКТЫ_ЗАДАНИЯ

Аномалии обновления, возникающие из-за наличия неминимальных функциональных зависимостей

Во множество FD отношения СЛУЖАЩИЕ_ПРОЕКТЫ_ЗАДАНИЯ входит много FD, в которых детерминантом является не возможный ключ отношения (соответствующие стрелки в диаграмме начинаются не с {СЛУ_НОМ, ПРО_НОМ}, т. е. некоторые функциональные зависимости атрибутов от возможного ключа не являются минимальными). Это приводит к так называемым *аномалиям обновления*. Под аномалиями обновления понимаются трудности, с которыми приходится сталкиваться при выполнении операций добавления кортежей в отношение (INSERT), удаления кортежей (DELETE) и модификации кортежей (UPDATE). Обсудим сначала аномалии обновления, вызываемые наличием FD СЛУ_НОМ→СЛУ_УРОВ (эти аномалии связаны с избыточностью хранения значений атрибутов СЛУ_УРОВ и СЛУ_ЗАРП в каждом кортеже, описывающем задание служащего в некотором проекте).

- *Добавление кортежей*. Мы не можем дополнить отношение СЛУЖАЩИЕ_ПРОЕКТЫ_ЗАДАНИЯ данными о служащем, который в данное время еще не участвует ни в одном проекте (ПРО_НОМ является частью первичного ключа и не может содержать неопределенных значений). Между тем часто бывает, что сначала служащего принимают на работу, устанавливают его разряд и размер зарплаты, а лишь потом назначают для него проект.
- *Удаление кортежей*. Мы не можем сохранить в отношении СЛУЖАЩИЕ_ПРОЕКТЫ_ЗАДАНИЯ данные о служащем, завершившем учас-

тие в своем последнем проекте (по той причине, что значение атрибута ПРО_НОМ для этого служащего становится неопределенным). Между тем характерна ситуация, когда между проектами возникают перерывы, не приводящие к увольнению служащих.

- *Модификация кортежей.* Чтобы изменить разряд служащего, мы будем вынуждены модифицировать все кортежи с соответствующим значением атрибута СЛУ_НОМ. В противном случае будет нарушена естественная FD СЛУ_НОМ→СЛУ_УРОВ (у одного служащего имеется только один разряд).

Возможная декомпозиция

Для преодоления этих трудностей можно произвести декомпозицию переменной отношения СЛУЖАЩИЕ_ПРОЕКТЫ_ЗАДАНИЯ на две переменных отношений – СЛУЖ {СЛУ_НОМ, СЛУ_УРОВ, СЛУ_ЗАРП} и СЛУЖ_ПРО_ЗАДАН {СЛУ_НОМ, ПРО_НОМ, СЛУ_ЗАДАН}. На основании теоремы Хита эта декомпозиция является декомпозицией без потерь, поскольку в исходном отношении имелась FD {СЛУ_НОМ, ПРО_НОМ}→СЛУ_ЗАДАН. На рис. 7.3 показаны диаграммы множеств FD этих отношений, а на рис. 7.4 – их значения.

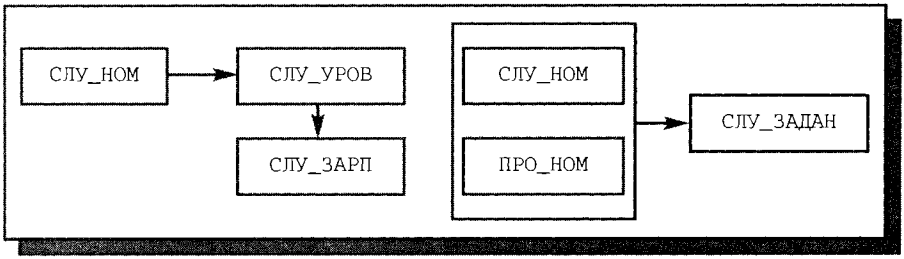


Рис. 7.3. Диаграммы FD в переменных отношений СЛУЖ и СЛУЖ_ПРО_ЗАДАН

Теперь мы можем легко справиться с операциями обновления.

- *Добавление кортежей.* Чтобы сохранить данные о принятом на работу служащем, который еще не участвует ни в каком проекте, достаточно добавить соответствующий кортеж в отношение СЛУЖ.
- *Удаление кортежей.* Если кто-то из служащих прекращает работу над проектом, достаточно удалить соответствующий кортеж из отношения СЛУЖ_ПРО_ЗАДАН. При увольнении служащего нужно удалить кортежи с соответствующим значением атрибута СЛУ_НОМ из отношений СЛУЖ и СЛУЖ_ПРО_ЗАДАН.
- *Модификация кортежей.* Если у служащего меняется разряд (и, следовательно, размер зарплаты), достаточно модифицировать один кортеж в отношении СЛУЖ.

Значение переменной отношения СЛУЖ

СЛУ_НОМ	СЛУ_УРОВ	СЛУ_ЗАРП
2934	2	22400.00
2935	3	29600.00
2936	1	20000.00
2937	1	20000.00

Значение переменной отношения СЛУЖ_ПРО_ЗАДАН

СЛУ_НОМ	ПРО_НОМ	СЛУ_ЗАДАН
2934	1	A
2935	1	B
2936	1	C
2937	1	D
2934	2	D
2935	2	C
2936	2	B
937	2	A

Рис. 7.4. Значения переменных отношений

Вторая нормальная форма

Как видно, на рис. 7.3 отсутствуют FD, не являющиеся *минимальными*. Наличие таких FD на рис. 7.1 вызывало аномалии обновления. Проблема заключалась в том, что атрибут СЛУЖ_УРОВ относился к сущности *служащий*, в то время как первичный ключ идентифицировал сущность *задание_служащего_в_проекте*.

Определение 7.1. Вторая нормальная форма

Переменная отношения находится во второй нормальной форме (2NF) тогда и только тогда, когда она находится в первой нормальной форме, и каждый неключевой атрибут* минимально функционально зависит от первичного ключа.** **Конец определения.**

Переменные отношений СЛУЖ и СЛУЖ_ПРО_ЗАДАН находятся в 2NF (все неключевые атрибуты отношений минимально зависят от первичных

* Неключевым атрибутом называется атрибут, не входящий ни в один возможный ключ.

** В определении предполагается, что у отношения имеется только один возможный ключ.

ключей СЛУ_НОМ и {СЛУ_НОМ, ПРО_НОМ} соответственно). Переменная отношения СЛУЖАЩИЕ_ПРОЕКТЫ_ЗАДАНИЯ не находится в 2NF (например, FD {СЛУ_НОМ, ПРО_НОМ}→СЛУ_УРОВ не является минимальной). Любая переменная отношения, находящаяся в 1NF, но не находящаяся в 2NF, может быть приведена к набору переменных отношений, находящихся в 2NF. В результате декомпозиции мы получаем набор проекций исходной переменной отношения, естественное соединение значений которых воспроизводит значение исходной переменной отношения (т. е. это декомпозиция без потерь). Для переменных отношений СЛУЖ и СЛУЖ_ПРО_ЗАДАН исходное отношение СЛУЖАЩИЕ_ПРОЕКТЫ_ЗАДАНИЯ воспроизводится их естественным соединением по общему атрибуту СЛУ_НОМ.

Заметим, что допустимое значение переменной отношения СЛУЖ может содержать кортежи, информационное наполнение которых выходит за пределы допустимых значений переменной отношения СЛУЖАЩИЕ_ПРОЕКТЫ_ЗАДАНИЯ. Например, в теле отношения СЛУЖ может находиться кортеж с данными о служащем с номером 2938, который еще не участвует ни в одном проекте. Наличие такого кортежа не влияет на результат естественного соединения, тело которого все равно будет совпадать с телом допустимого значения переменной отношения СЛУЖАЩИЕ_ПРОЕКТЫ_ЗАДАНИЯ.

Нетранзитивные функциональные зависимости и третья нормальная форма

В произведенной декомпозиции переменной отношения СЛУЖАЩИЕ_ПРОЕКТЫ_ЗАДАНИЯ множество FD переменной отношения СЛУЖ_ПРО_ЗАДАН предельно просто – в единственной нетривиальной функциональной зависимости детерминантом является возможный ключ. При использовании этой переменной отношения какие-либо аномалии обновления не возникают. Однако переменная отношения СЛУЖ не является такой же совершенной.

Аномалии обновлений, возникающие из-за наличия транзитивных функциональных зависимостей

Функциональные зависимости переменной отношения СЛУЖ по-прежнему порождают некоторые аномалии обновления. Они вызываются наличием транзитивной FD СЛУ_НОМ→СЛУ_ЗАРП (через FD СЛУ_НОМ→СЛУ_УРОВ и СЛУ_УРОВ→СЛУ_ЗАРП). Эти аномалии связаны с избыточностью хранения значения атрибута СЛУ_ЗАРП в каждом кортеже, характеризующем служащих с одним и тем же разрядом.

- *Добавление кортежей.* Невозможно сохранить данные о новом разряде (и соответствующем ему размере зарплаты), пока не появится служа-

Значение переменной отношения **СЛУЖ1**

СЛУ_НОМ	СЛУ_УРОВ
2934	2
2935	3
2936	1
2937	1

Значение переменной отношения **УРОВ**

СЛУ_УРОВ	СЛУ_ЗАРП
2	22400.00
3	29600.00
1	20000.00

Рис. 7.6. Тела отношений СЛУЖ1 и УРОВ

означало, что атрибут СЛУ_ЗАРП характеризовал не сущность *служащий*, сущность *разряд*.

Определение 7.2. Третья нормальная форма

Переменная отношения находится в третьей нормальной форме (3NF в том и только в том случае, когда она находится во второй нормальной форме, и каждый неключевой атрибут нетранзитивно* функционально зависит от первичного ключа.** **Конец определения.**

Отношения СЛУЖ1 и УРОВ оба находятся в 3NF (все неключевые атрибуты нетранзитивно зависят от первичных ключей СЛУ_НОМ и СЛУ_УРОВ). Отношение СЛУЖ не находится в 3NF (FD СЛУ_НОМ→СЛУ_ЗАР является транзитивной). Любое отношение, находящееся в 2NF, но не находящееся в 3NF, может быть приведено к набору отношений, находящихся в 3NF. Мы получаем набор проекций исходного отношения, естественное соединение которых воспроизводит исходное отношение (т. е это декомпозиция без потерь). Для отношений СЛУЖ1 и УРОВ исходное отношение СЛУЖ воспроизводится их естественным соединением по общему атрибуту СЛУ_УРОВ.

Заметим, что допустимые значения отношения УРОВ могут содержать кортежи, информационное наполнение которых выходит за пределы тел

* Очевидно, что FD называется нетранзитивной тогда и только тогда, когда она не являетс транзитивной.

** В этом определении опять предполагается, что у отношения имеется только один возмож ный ключ.

щий с новым разрядом. (Первичный ключ не может содержать неопределенные значения.)

- *Удаление кортежей.* При увольнении последнего служащего с данным разрядом мы утратим информацию о наличии такого разряда и соответствующем размере зарплаты.
- *Модификация кортежей.* При изменении размера зарплаты, соответствующей некоторому разряду, мы будем вынуждены изменить значение атрибута СЛУ_ЗАРП в кортежах всех служащих, которым назначен этот разряд (иначе не будет выполняться FD СЛУ_УРОВ→ СЛУ_ЗАРП).

Возможная декомпозиция

Для преодоления этих трудностей произведем декомпозицию переменной отношения СЛУЖ на две переменных отношений – СЛУЖ1 {СЛУ_НОМ, СЛУ_УРОВ} и УРОВ {СЛУ_УРОВ, СЛУ_ЗАРП}. По теореме Хита, это снова декомпозиция без потерь по причине наличия, например, FD СЛУ_НОМ→СЛУ_УРОВ. На рис. 7.5 показаны диаграммы FD этих переменных отношений, а на рис. 7.6 – их возможные значения.

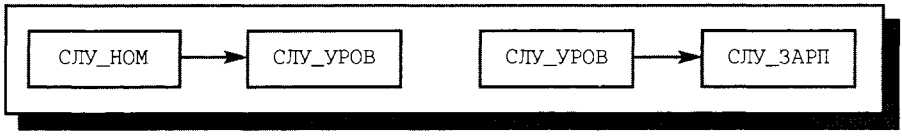


Рис. 7.5. Диаграммы FD в отношениях СЛУЖ1 и УРОВ

Как видно из рис. 7.6, это преобразование обратимо, т. е. любое допустимое значение исходной переменной отношения СЛУЖ является естественным соединением значений отношений СЛУЖ1 и УРОВ. Также можно заметить, что мы избавились от трудностей при выполнении операций обновления.

- *Добавление кортежей.* Чтобы сохранить данные о новом разряде, достаточно добавить соответствующий кортеж к отношению УРОВ.
- *Удаление кортежей.* При увольнении последнего служащего, обладающего данным разрядом, удаляется соответствующий кортеж из отношения СЛУЖ1, и данные о разряде сохраняются в отношении УРОВ.
- *Модификация кортежей.* При изменении размера зарплаты, соответствующей некоторому разряду, изменяется значение атрибута СЛУ_ЗАРП ровно в одном кортеже отношения УРОВ.

Третья нормальная форма

Трудности, которые мы испытывали, были связаны с наличием транзитивной FD СЛУ_НОМ→СЛУ_ЗАРП. Наличие этой FD на самом деле

отношения СЛУЖ. Например, в теле отношения УРОВ может находиться кортеж с данными о разряде 4, который еще не присвоен ни одному служащему. Наличие такого кортежа не влияет на результат естественного соединения, который все равно будет являться допустимым значением отношения СЛУЖ.

Независимые проекции отношений. Теорема Риссанена

Обратите внимание, что для переменной отношения СЛУЖ {СЛУ_НОМ, СЛУ_УРОВ, СЛУ_ЗАРП}, кроме декомпозиции на отношения СЛУЖ1 {СЛУ_НОМ, СЛУ_УРОВ} и УРОВ {СЛУ_УРОВ, СЛУ_ЗАРП}, возможна и декомпозиция на отношения СЛУЖ1 {СЛУ_НОМ, СЛУ_УРОВ} и СЛУЖ_ЗАРП {СЛУ_НОМ, СЛУ_ЗАРП}.* Оба отношения, полученные путем второй декомпозиции, находятся в 3NF, и эта декомпозиция также является декомпозицией без потерь. Тем не менее вторая декомпозиция, в отличие от первой, не устраняет проблемы, связанные с обновлением отношения СЛУЖ. Например, по-прежнему невозможно сохранить данные о разряде, которым не обладает ни один служащий. Посмотрим, с чем это связано.

Отношения СЛУЖ1 и УРОВ могут обновляться независимо (являются *независимыми проекциями*), и при этом результат их естественного соединения всегда будет таким, как если бы обновлялось исходное отношение СЛУЖ. Это происходит потому, что FD отношения СЛУЖ трансформировались в индивидуальные ограничения первичного ключа отношений СЛУЖ1 и УРОВ. При второй декомпозиции FD СЛУ_УРОВ→СЛУ_ЗАРП трансформируется в ограничение целостности сразу для двух отношений (такого рода ограничения целостности называются ограничениями базы данных, и их поддержка гораздо более накладна с технической точки зрения). Понятно, что в процессе нормализации декомпозиция отношения на независимые проекции является предпочтительной. Необходимые и достаточные условия независимости проекций отношения обеспечивает теорема Риссанена.

Теорема Риссанена

Проекции *r1* и *r2* отношения *r* являются независимыми тогда и только тогда, когда:

- каждая FD в отношении *r* логически следует** из FD в *r1* и *r2*;
- общие атрибуты *r1* и *r2* образуют возможный ключ хотя бы для одного из этих отношений.

Мы не будем приводить доказательство этой теоремы, но продемонстрируем ее верность на примере двух показанных выше декомпозиций

* Теоретически возможная третья декомпозиция отношения СЛУЖ на отношения СЛУЖ2 {СЛУ_НОМ, СЛУ_ЗАРП} и УРОВ {СЛУ_УРОВ, СЛУ_ЗАРП} не является декомпозицией без потерь. Чтобы убедиться в этом, рассмотрите случай, когда для двух разных разрядов служащих назначен один и тот же размер зарплаты. Покажите также, что для этой декомпозиции не выполняются условия теоремы Хита.

** Т.е. выводится на основе аксиом Армстронга.

отношения СЛУЖ. В первой декомпозиции (на проекции СЛУЖ1 и УРОВ) общий атрибут СЛУ_УРОВ является возможным (и первичным) ключом отношения УРОВ, а единственная дополнительная FD отношения СЛУЖ (СЛУ_НОМ→СЛУ_ЗАРП) логически следует из FD СЛУ_НОМ→СЛУ_УРОВ и СЛУ_УРОВ→СЛУ_ЗАРП, выполняемых для отношений СЛУЖ1 и УРОВ соответственно. Вторая декомпозиция удовлетворяет второму условию теоремы Риссанена (СЛУ_НОМ является первичным ключом в каждом из отношений СЛУЖ1 и СЛУ_ЗАРП), но FD СЛУ_УРОВ→СЛУ_ЗАРП не выводится из FD СЛУ_НОМ→СЛУ_УРОВ и СЛУ_НОМ→СЛУ_ЗАРП.

Атомарным отношением называется отношение, которое невозможно декомпозировать на независимые проекции. Далеко не всегда для неатомарных (не являющихся атомарными) отношений требуется декомпозиция на атомарные проекции. Например, отношение СЛУЖ2 {СЛУ_НОМ, СЛУ_ЗАРП, ПРО_НОМ} с множеством FD {СЛУ_НОМ→СЛУ_ЗАРП, СЛУ_НОМ→ ПРО_НОМ} не является атомарным (возможна декомпозиция на независимые проекции СЛУЖ3 {СЛУ_НОМ, СЛУ_ЗАРП} и СЛУЖ4 {СЛУ_НОМ, ПРО_НОМ}). Но эта декомпозиция не улучшает свойства отношения СЛУЖ2 и поэтому не является осмысленной. Другими словами, при выборе способа декомпозиции нужно стремиться к получению независимых проекций, но не обязательно атомарных.

Перекрывающиеся возможные ключи и нормальная форма Бойса-Кодда

До сих пор в определениях нормальных форм мы предполагали, что у декомпозируемого отношения имеется только один возможный ключ. На практике чаще всего бывает именно так. Но имеется один частный случай, который (почти) удовлетворяет требованиям 2NF и 3NF, но, тем не менее, порождает аномалии обновления. Это тот случай, когда у отношения имеется несколько возможных ключей, и некоторые из этих возможных ключей «перекрываются», т. е. содержат общие атрибуты.

Аномалии обновлений, связанные с наличием перекрывающихся возможных ключей

Например, пусть имеется переменная отношения СЛУЖ_ПРО_ЗАДАН1 {СЛУ_НОМ, СЛУ_ИМЯ, ПРО_НОМ, СЛУ_ЗАДАН} с множеством FD, показанным на рис. 7.7.

В отношении СЛУЖ_ПРО_ЗАДАН1 служащие уникально идентифицируются как по номерам удостоверений, так и по именам. Следовательно, существуют FD СЛУ_НОМ→СЛУ_ИМЯ и СЛУ_ИМЯ→СЛУ_НОМ. Но один служащий может участвовать в нескольких проектах, поэтому возможными ключами

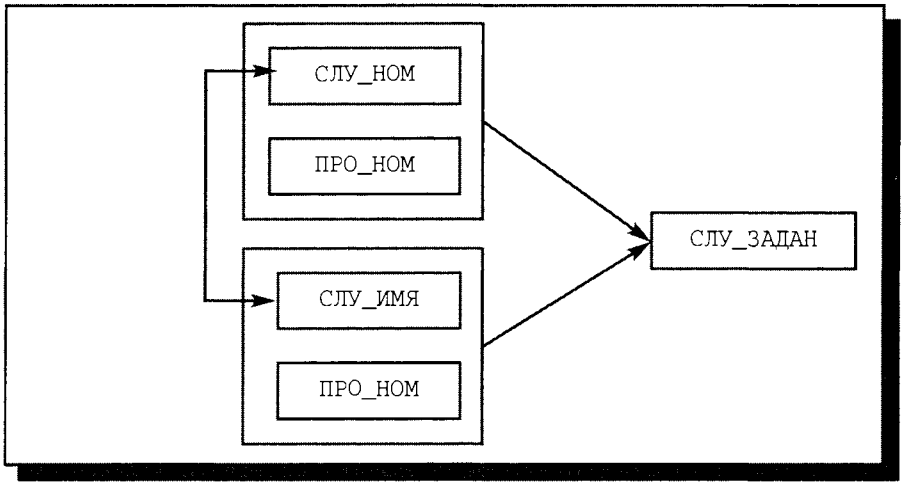


Рис. 7.7. Диаграмма FD отношения **СЛУЖ_ПРО_ЗАДАН1**

являются {СЛУ_НОМ, ПРО_НОМ} и {СЛУ_ИМЯ, ПРО_НОМ}. На рис. 7.8 показано возможное значение переменной отношения СЛУЖ_ПРО_ЗАДАН1.

СЛУ_НОМ	СЛУ_ИМЯ	ПРО_НОМ	ПРО_ЗАДАН
2934	Иванов	1	A
2941	Иваненко	2	B
2934	Иванов	2	B
941	Иваненко	1	A

Рис. 7.8. Возможное значение переменной отношения СЛУЖ_ПРО_ЗАДАН1

Очевидно, что, хотя в отношении СЛУЖ_ПРО_ЗАДАН1 все FD неключевых атрибутов от возможных ключей являются минимальными и транзитивные FD отсутствуют, этому отношению свойственны аномалии обновления. Например, в случае изменения имени служащего требуется обновить атрибут СЛУ_ИМЯ во всех кортежах отношения СЛУЖ_ПРО_ЗАДАН1, соответствующих данному служащему. Иначе будет нарушена FD СЛУ_НОМ→СЛУ_ИМЯ, и база данных окажется в несогласованном состоянии.

Нормальная форма Бойса-Кодда

Причиной отмеченных аномалий является то, что в требованиях 2NF и 3NF не требовалась минимальная функциональная зависимость от первичного ключа атрибутов, являющихся компонентами других возможных ключей. Проблему решает нормальная форма, которую исторически принято называть нормальной формой Бойса-Кодда и которая является уточнением 3NF в случае наличия нескольких перекрывающихся возможных ключей.

Определение 7.3. Нормальная форма Бойса-Кодда

Переменная отношения находится в нормальной форме Бойса-Кодда (BCNF) в том и только в том случае, когда любая выполняемая для этой переменной отношения нетривиальная и минимальная FD имеет в качестве детерминанта некоторый возможный ключ данного отношения. **Конец определения**.

Переменная отношения СЛУЖ_ПРО_ЗАДАН1 может быть приведена к BCNF путем одной из двух декомпозиций: СЛУЖ_НОМ_ИМЯ {СЛУ_НОМ, СЛУ_ИМЯ} и СЛУЖ_НОМ_ПРО_ЗАДАН {СЛУ_НОМ, ПРО_НОМ, СЛУ_ЗАДАН} с множеством FD и значениями, показанными на рис. 7.9, и СЛУЖ_НОМ_ИМЯ {СЛУ_НОМ, СЛУ_ИМЯ} и СЛУЖ_ИМЯ_ПРО_ЗАДАН {СЛУ_ИМЯ, ПРО_НОМ, СЛУ_ЗАДАН} (FD и значения результирующих переменных отношений выглядят аналогично).

Очевидно, что каждая из декомпозиций устраняет трудности, связанные с обновлением отношения СЛУЖ_ПРО_ЗАДАН1.

Всегда ли следует стремиться к BCNF?

Предположим теперь, что в организации все проекты включают разные задания, и по-прежнему каждый служащий может участвовать в нескольких проектах, но может выполнять в каждом проекте только одно задание. Одно задание в каждом проекте могут выполнять несколько служащих. Тогда переменная отношения СЛУЖ_ПРО_ЗАДАН имеет множество FD, показанное на рис. 7.10, и может содержать значение, представленное на том же рисунке.

В этом отношении существуют два возможных ключа: {СЛУ_НОМ, ПРО_НОМ} и {СЛУ_НОМ, СЛУ_ЗАДАН}. Отношение удовлетворяет требованиям 3NF: отсутствуют неминимальные FD неключевых атрибутов от возможных ключей (поскольку нет неключевых атрибутов) и отсутствуют транзитивные FD. Однако из-за наличия FD СЛУ_ЗАДАН→ПРО_НОМ это отношение не находится в BCNF. Поэтому отношению СЛУ_ПРО_ЗАДАН снова свойственны аномалии обновления. Например (поскольку СЛУ_НОМ является компонентом обоих возможных ключей), невозможно удалить

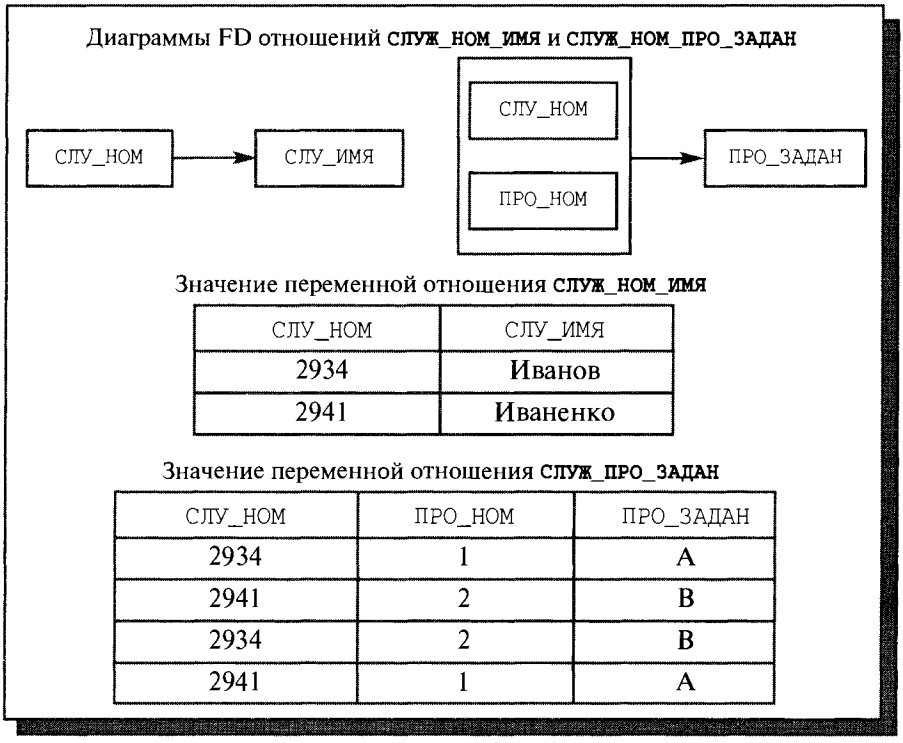


Рис. 7.9. Диаграммы FD и значения переменных отношений СЛУЖ_НОМ_ИМЯ и СЛУЖ_НОМ_ПРО_ЗАДАН

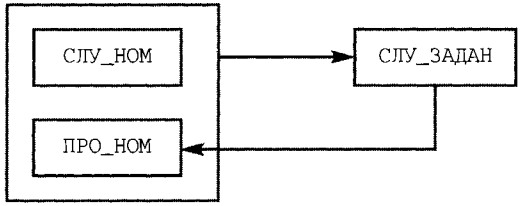
данные о единственном служащем, выполняющем задание в некотором проекте, не утратив информацию об этом задании.

Можно привести отношение СЛУЖ_ПРО_ЗАДАН к BCNF, выполнив его декомпозицию на отношения СЛУЖ_НОМ_ЗАДАН {СЛУ_НОМ, СЛУ_ЗАДАН}* и ПРО_НОМ_ЗАДАН {СЛУ_ЗАДАН, ПРО_НОМ}, и эта декомпозиция решает обозначенные проблемы (теперь можно хранить данные о задании проекта, не выполняемом ни одним служащим). Значения переменных отношений СЛУЖ_НОМ_ЗАДАН и ПРО_НОМ_ЗАДАН показаны на рис. 7.11.

Однако возникают новые трудности. Например, система должна запретить добавление в отношение СЛУЖ_НОМ_ЗАДАН кортежа <2934, D>, поскольку задание D относится к проекту 1, а служащий с номером 2934 уже выполняет задание в этом проекте. Так происходит, потому что исходная FD {СЛУ_НОМ, ПРО_НОМ}→СЛУ_ЗАДАН не выводится из единственной (нетривиальной) действующей для этих проекций FD СЛУ_ЗАДАН→

* Единственным возможным ключом отношения СЛУЖ_НОМ_ЗАДАН является {СЛУ_НОМ, СЛУ_ЗАДАН}, и в этом отношении отсутствуют нетривиальные FD.

Диаграмма FD отношения **СЛУЖ_ПРО_ЗАДАН** (новый вариант)



Значение переменной отношения **СЛУЖ_ПРО_ЗАДАН** (новый вариант)

СЛУ_НОМ	ПРО_НОМ	ПРО_ЗАДАН
2934	1	A
2935	1	A
2941	2	B
2934	2	C
2941	1	D

Рис. 7.10. Новый вариант переменной отношения СЛУЖ_ПРО_ЗАДАН

ПРО_НОМ, и соответствующее ограничение целостности становится ограничением базы данных.

Тем самым, проекции СЛУЖ_НОМ_ЗАДАН и ПРО_НОМ_ЗАДАН не являются независимыми, а отношение СЛУЖ_ПРО_ЗАДАН атомарно, хотя и не находится в BCNF. Из этого следует, что при проектировании реляционной базы данных приведение отношения к BCNF не должно быть самоцелью. Нужно внимательно оценивать положительные и отрицательные последствия нормализации.

Наконец, приведем пример, когда наличие двух перекрывающихся возможных ключей не мешает отношению находиться в BCNF. Предположим, что в организации проекты включают одни и те же задания, каждый служащий может участвовать в нескольких проектах, но может выполнять в каждом проекте только одно задание. Тогда переменная отношения СЛУЖ_НОМ_ЗАДАН имеет множество FD, показанное на рис. 7.12, и может содержать значение, показанное на том же рисунке.

В третьем варианте отношения СЛУЖ_НОМ_ЗАДАН имеются перекрывающиеся возможные ключи ({СЛУ_НОМ, ПРО_НОМ} и {ПРО_НОМ, СЛУ_ЗАДАН}), однако оно находится в BCNF, поскольку эти ключи являются единственными детерминантами. Легко убедиться, что отношению СЛУЖ_НОМ_ЗАДАН аномалии обновления не свойственны.

Значение переменной отношения **СЛУЖ_НОМ_ЗАДАН**

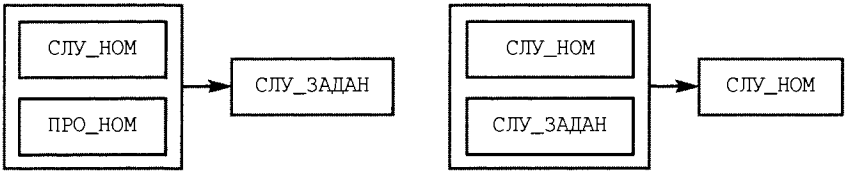
СЛУ_НОМ	ПРО_ЗАДАН
2934	A
2935	A
2941	B
2934	C
2941	D

Значение переменной отношения **ПРО_НОМ_ЗАДАН**

ПРО_НОМ	ПРО_ЗАДАН
1	A
2	B
2	C
1	D

Рис. 7.11. Значения переменных отношений СЛУЖ_НОМ_ЗАДАН и ПРО_НОМ_ЗАДАН

Диаграмма FD отношения **СЛУЖ_НОМ_ЗАДАН** (третий вариант)



Возможное значение переменной отношения **СЛУЖ_НОМ_ЗАДАН** (третий вариант)

СЛУ_НОМ	ПРО_НОМ	СЛУ_ЗАДАН
2934	1	A
2935	1	B
2941	2	B
2934	2	A
2941	1	D

Рис. 7.12. Третий вариант отношения СЛУЖ_НОМ_ЗАДАН

Заключение

В этой лекции мы обсудили три начальные нормальные формы отношений – вторую и третью нормальные формы и нормальную форму Бойса-Кодда, – которые производятся путем декомпозиции без потерь исходного отношения на две проекции, где отсутствуют аномалии изменений, существовавшие в исходном отношении по причине наличия функциональных зависимостей с нежелательными свойствами.

Нормализация схемы базы данных способствует более эффективному выполнению системой управления базами данных операций обновления базы данных, поскольку сокращается число проверок и вспомогательных действий, поддерживающих целостность базы данных. При проектировании реляционной базы данных почти всегда добиваются второй нормальной формы всех входящих в базу данных отношений. В часто обновляемых базах данных обычно стараются обеспечить третью нормальную форму отношений. На нормальную форму Бойса-Кодда внимание обращают гораздо реже, поскольку на практике ситуации, в которых у отношения имеется несколько составных перекрывающихся возможных ключей, встречаются нечасто.

Лекция 8. Проектирование реляционных баз данных на основе принципов нормализации: дальнейшая нормализация

В этой лекции мы обсудим два более сложных вида зависимостей между атрибутами отношений – многозначные зависимости и зависимости проекции/соединения, а также основанные на учете таких зависимостей нормальные формы отношений. Основанная на учете зависимости проекции/соединения пятая нормальная форма отношения является «окончательной» нормальной формой, которую можно получить путем декомпозиции отношений на основе их проецирования.

Ключевые слова: многозначная зависимость, лемма Фейджина, теорема Фейджина, четвертая нормальная форма, *n*-декомпозируемое отношение, тривиальная многозначная зависимость, зависимость проекции/соединения, зависимость проекции/соединения, подразумеваемая возможными ключами, тривиальная зависимость проекции/соединения, пятая нормальная форма, нормальная форма проекции/соединения, окончательность пятой нормальной формы.

Введение

Функциональные зависимости, о которых мы говорили в предыдущих двух лекциях, и нормальные формы, основанные на учете «аномальных» функциональных зависимостей, являются естественными и легко понимаемыми, поскольку в их основе лежит понятие функционального отображения, интуитивно понятного даже людям, далеким от математики. Конечно, было бы замечательно, если бы ликвидация в ходе нормализации аномальных функциональных зависимостей гарантировала отсутствие аномалий обновления отношений.

К сожалению, эта гарантия в общем случае не обеспечивается. Иногда в переменных отношений требуется поддержка более сложных ограничений целостности, для выражения которых понятие функции оказывается недостаточным. Класс зависимостей, опирающихся на понятие *функционала* – обобщение понятия функции, обнаружил в 1970-е гг. Рональд Фейджин. Он назвал такие зависимости *многозначными*, поскольку в них одному значению детерминанта соответствует множество значений зависимого атрибута. Наличие в переменной отношения многозначных зависимостей, не являющихся функциональными зависимостями от возможного ключа, приводит к аномалиям обновления таких отношений. Фейджин показал, что в этом случае возможна декомпозиция данных отношений на две проекции, для

которых подобные аномалии обновления не проявляются. Такие проекции находятся в *четвертой нормальной форме*.

Позже было установлено, что при наличии некоторых естественных ограничений, являющихся обобщением ограничений многозначных зависимостей, и в отношениях, которые находятся в четвертой нормальной форме, проявляются аномалии обновления. Более того, эти аномалии невозможно устранить путем проецирования отношения на две проекции, требуется декомпозиция на три или большее число отношений. Такие ограничения получили название зависимостей *проекции/соединения*. Отношение, в котором существует нетривиальная зависимость проекции/соединения, может быть декомпозировано на три или большее число проекций, в которых зависимости проекции/соединения следуют из возможного ключа. Такие проекции находятся в *пятой нормальной форме*, или *нормальной форме проекции/соединения*. В отношениях, находящихся в пятой нормальной форме, отсутствуют аномалии обновления, которые можно было бы устранить путем декомпозиции, и поэтому при достижении пятой нормальной формы процесс проектирования реляционной базы данных на основе нормализации естественным образом завершается.

Многозначные зависимости и четвертая нормальная форма

Чтобы перейти к вопросам дальнейшей нормализации, рассмотрим еще одну возможную (четвертую) интерпретацию переменной отношения СЛУЖ_ПРО_ЗАДАН. Предположим, что каждый служащий может участвовать в нескольких проектах, но в каждом проекте, в котором он участвует, им должны выполняться одни и те же задания. Возможное значение четвертого варианта переменной отношения СЛУЖ_ПРО_ЗАДАН показано на рис. 8.1.

СЛУ_НОМ	ПРО_НОМ	СЛУ_ЗАДАН
2934	1	A
2934	1	B
2934	2	A
2934	2	B
....	....	....
2941	1	A
2941	1	D

Рис. 8.1. Возможное значение переменной отношения СЛУЖ_ПРО_ЗАДАН (четвертый вариант)

Аномалии обновлений при наличии многозначных зависимостей и возможная декомпозиция

В новом варианте переменной отношения единственным возможным ключом является заголовок отношения {СЛУ_НОМ, ПРО_НОМ, СЛУ_ЗАДАН}. Кортеж <*сн*, *пн*, *сз*> входит в тело отношения в том и только в том случае, когда служащий с номером *сн* выполняет в проекте *пн* задание *сз*. Поскольку для каждого служащего указываются все проекты, в которых он участвует, и все задания, которые он должен выполнять в этих проектах, для каждого допустимого значения переменной отношения СЛУЖ_ПРО_ЗАДАН должно выполняться следующее ограничение (ТСПЗ обозначает тело отношения):

```
IF (<сн, пн1, сз1> ТСПЗ AND <сн, пн2, сз2> ТСПЗ)
THEN (<сн, пн1, сз2> ТСПЗ AND <сн, пн2, сз1> ТСПЗ)
```

Наличие такого ограничения (как мы скоро увидим, это ограничение порождается наличием многозначной зависимости) приводит к тому, что при работе с отношением СЛУЖ_ПРО_ЗАДАН проявляются аномалии обновления.

- *Добавление кортежа.* Если уже участвующий в проектах служащий присоединяется к новому проекту, то к телу значения переменной отношения СЛУЖ_ПРО_ЗАДАН требуется добавить столько кортежей, сколько заданий выполняет этот служащий.
- *Удаление кортежей.* Если служащий прекращает участие в проектах, то отсутствует возможность сохранить данные о заданиях, которые он может выполнять.
- *Модификация кортежей.* При изменении одного из заданий служащего необходимо изменить значение атрибута СЛУ_ЗАДАН в стольких кортежах, в скольких проектах участвует служащий.

Трудности, связанные с обновлением переменной отношения СЛУЖ_ПРО_ЗАДАН, решаются путем его декомпозиции на две переменных отношений: СЛУЖ_ПРО_НОМ {СЛУ_НОМ, ПРО_НОМ} и СЛУЖ_ЗАДАНИЕ {СЛУ_НОМ, СЛУ_ЗАДАН}. Значения этих переменных отношений, соответствующие значению переменной отношения СЛУЖ_ПРО_ЗАДАН с рис. 8.1, показаны на рис. 8.2.

Легко видеть, что декомпозиция, представленная на рис. 8.2, является декомпозицией без потерь и что эта декомпозиция решает перечисленные выше проблемы с обновлением переменной отношения СЛУЖ_ПРО_ЗАДАН.

- *Добавление кортежа.* Если некоторый уже участвующий в проектах служащий присоединяется к новому проекту, то к телу значения пере-

Значение переменной отношения **СЛУЖ_ПРО_НОМ**

СЛУ_НОМ	ПРО_НОМ
2934	1
2934	2
....	....
2941	1

Значение переменной отношения **СЛУЖ_ЗАДАНИЕ**

СЛУ_НОМ	СЛУ_ЗАДАН
2934	A
2934	B
....	....
2941	A
2941	D

Рис. 8.2. Значения переменных отношений СЛУЖ_ПРО_НОМ и СЛУЖ_ЗАДАНИЕ

менной отношения СЛУЖ_ПРО_НОМ требуется добавить один кортеж, соответствующий новому проекту.

- *Удаление кортежей.* Если служащий прекращает участие в проектах, то данные о заданиях, которые он может выполнять, остаются в отношении СЛУЖ_ЗАДАНИЕ.
- *Модификация кортежей.* При изменении одного из заданий служащего необходимо изменить значение атрибута СЛУ_ЗАДАН в одном кортеже отношения СЛУЖ_ЗАДАНИЕ.

Многозначные зависимости. Теорема Фейджина. Четвертая нормальная форма

Заметим, что последний вариант переменной отношения СЛУЖ_ПРО_ЗАДАН находится в BCNF, поскольку все атрибуты заголовка отношения входят в состав единственно возможного ключа. В этом отношении вообще отсутствуют нетривиальные FD. Поэтому ранее обсуждавшиеся принципы нормализации здесь неприменимы, но, тем не менее, мы получили полезную декомпозицию. Все дело в том, что в случае четвертого варианта отношения СЛУЖ_ПРО_ЗАДАН мы имеем дело с новым видом зависимости, впервые обнаруженным Роном Фейджином в 1971 г. Фейджин назвал зависимости этого вида *многозначными* (multi-valued dependency – MVD). Как мы увидим немного позже, MVD является обобщением понятия FD.

В отношении СЛУЖ_ПРО_ЗАДАН выполняются две MVD: СЛУ_НОМ →→ПРО_НОМ и СЛУ_НОМ→→СЛУ_ЗАДАН. Первая MVD означает, что каждому значению атрибута СЛУ_НОМ соответствует определяемое только этим значением множество значений атрибута ПРО_НОМ. Другими словами, в результате вычисления алгебраического выражения

```
(СЛУЖ_ПРО_НОМ WHERE (СЛУ_НОМ = сн AND СЛУ_ЗАДАН = сз))
  PROJECT {ПРО_НОМ}
```

для фиксированного допустимого значения *сн* и любого допустимого значения *сз* мы всегда получим одно и то же множество значений атрибута ПРО_НОМ. Аналогично трактуется вторая MVD.

Определение 8.1. Многозначная зависимость

В переменной отношения r с атрибутами A, B, C (в общем случае, составными) имеется многозначная зависимость B от A (A→→B) в том и только в том случае, когда множество значений атрибута B, соответствующее паре значений атрибутов A и C, зависит от значения A и не зависит от значения C. **Конец определения.**

Многозначные зависимости обладают интересным свойством «двойственности», которое демонстрирует следующая лемма.

Лемма Фейджина

В отношении r {A, B, C} выполняется MVD A→→B в том и только в том случае, когда выполняется MVD A→→C.

Доказательство достаточности условия леммы. Пусть выполняется MVD A→→B. Пусть имеется некоторое удовлетворяющее этой зависимости значение V_r переменной отношения r, a обозначает значение атрибута A в некотором кортеже тела V_r, а {b} – множество значений атрибута B, взятых из всех кортежей тела V_r, в которых значением атрибута A является a. Предположим, что для этого значения a MVD A→→C не выполняется. Это означает, что существуют такое допустимое значение c атрибута C и такое значение b∈{b}, что кортеж <a, b, c>∉V_r. Но это противоречит наличию MVD A→→B. Следовательно, если выполняется MVD A→→B, то выполняется и MVD A→→C. Аналогично можно доказать необходимость условия леммы. **Конец доказательства.**

Таким образом, MVD A→→B и A→→C всегда составляют пару. Поэтому обычно их представляют вместе в форме A →→ B | C.

FD является частным случаем MVD, когда множество значений зависимого атрибута обязательно состоит из одного элемента. Таким образом, если выполняется FD A→B, то выполняется и MVD A→→B. *

Мы видим, что отношения СЛУЖ_ПРО_НОМ и СЛУЖ_ЗАДАНИЕ не содер-

* Упражнение по ходу лекции. Пусть имеется отношение r с атрибутами A, B, C (в общем случае, составными), в котором существует FD A → B. Что в этом случае можно сказать про зависимость атрибутов A и C?

жат MVD, отличных от FD, и именно в этом выигрывает декомпозиция из рис. 8.2. Правомочность этой декомпозиции доказывается приведенной ниже теоремой Фейджина, которая является уточнением и обобщением теоремы Хита.

Теорема Фейджина

Пусть имеется переменная отношения r с атрибутами A, B, C (в общем случае, составными). Отношение r декомпозируется без потерь на проекции $\{A, B\}$ и $\{A, C\}$ тогда и только тогда, когда для него выполняется MVD $A \rightarrow\!\!\!\rightarrow B \mid C$.

Докажем достаточность условия теоремы. Пусть V_r является некоторым допустимым значением переменной отношений r. Пусть a есть значение атрибута A в некотором кортеже тела V_r, $\{b\}$ — множество значений атрибута B, взятых из всех кортежей тела V_r, в которых значением атрибута A является a, и $\{c\}$ — множество значений атрибута C, взятых из всех кортежей тела V_r, в которых значением атрибута A является a. Тогда очевидно, что в тело значения переменной отношения r PROJECT $\{A, B\}$ будут входить все кортежи вида $\{a, b_i\}$, где $b_i \in \{b\}$, и если некоторый кортеж $\{a, b_j\}$ входит в тело значения переменной отношения r PROJECT $\{A, B\}$, то $b_j \in \{b\}$. Аналогичные рассуждения применимы к r PROJECT $\{A, C\}$. Очевидно, что из этого следует, что при наличии многозначной зависимости $A \rightarrow\!\!\!\rightarrow B \mid C$ в переменной отношения $r\{A, B, C\}$ декомпозиция r на проекции r PROJECT $\{A, B\}$ и r PROJECT $\{A, C\}$ является декомпозицией без потерь.

Для доказательства необходимости условия теоремы предположим, что декомпозиция переменной отношения r $\{A, B, C\}$ на проекции r PROJECT $\{A, B\}$ и r PROJECT $\{A, C\}$ является декомпозицией без потерь для любого допустимого значения V_r переменной отношения r. Мы должны показать, что в теле B_r значения-отношения V_r поддерживается ограничение

```
IF (<a, b1, c1> ∈ Br AND <a, b2, c2> ∈ Br)
THEN (<a, b1, c2> ∈ Br AND <a, b2, c1> ∈ Br)
```

Действительно, пусть в B_r входят кортежи $<a, b_1, c_1>$ и $<a, b_2, c_2>$. Предположим, что $<a, b_1, c_2> \notin B_r$ OR $<a, b_2, c_1> \notin B_r$. Но в тело значения переменной отношения r PROJECT $\{A, B\}$ входят кортежи $<a, b_1>$ и $<a, b_2>$, а в тело значения переменной отношения r PROJECT $\{A, C\}$ — $<a, c_1>$ и $<a, c_2>$. Очевидно, что в тело значения естественного соединения r PROJECT $\{A, B\}$ NATURAL JOIN r PROJECT $\{A, C\}$ войдут кортежи $<a, b_1, c_2>$ и $<a, b_2, c_1>$, и наше предположение об отсутствии по крайней мере одного из этих кортежей в B_r противоречит исходному предположению о том, что декомпозиция r на проекции r PROJECT $\{A, B\}$ и r PROJECT $\{A, C\}$ является декомпозицией без потерь. Тем самым, теорема Фейджина полностью доказана. **Конец доказательства**.

Теорема Фейджина обеспечивает основу для декомпозиции отношений, удаляющей «аномальные» многозначные зависимости, с приведением отношений в *четвертую нормальную форму*.

Определение 8.2. Четвертая нормальная форма

Переменная отношения r находится в четвертой нормальной форме (4NF) в том и только в том случае, когда она находится в BCNF, и все MVD r являются FD с детерминантами – возможными ключами отношения r. **Конец определения.**

В сущности, 4NF является BCNF, в которой многозначные зависимости вырождаются в функциональные (позволим себе на один момент отказаться от сокращений). Понятно, что отношение СЛУЖ_ПРО_ЗАДАН не находится в 4NF, поскольку детерминант MVD СЛУ_НОМ→→ПРО_НОМ и СЛУ_НОМ→→СЛУ_ЗАДАН не является возможным ключом, и эти MVD не являются функциональными. С другой стороны, отношения СЛУЖ_ПРО_НОМ и СЛУЖ_ЗАДАНИЕ находятся в BCNF и не содержат MVD, отличных от FD с детерминантом – возможным ключом. Поэтому они находятся в 4NF.

Зависимости проекции/соединения и пятая нормальная форма

Приведение отношения к 4NF предполагает его декомпозицию без потерь на две проекции (как и в случае 2NF, 3NF и BCNF). Однако бывают (хотя и нечасто) случаи, когда декомпозиция без потерь на две проекции невозможна, но можно произвести декомпозицию без потерь на большее число проекций. Будем называть *n*-декомпозируемым отношением отношение, которое может быть декомпозировано без потерь на *n* проекций. До сих пор мы имели дело с 2-декомпозируемыми отношениями.

N-декомпозируемые отношения

Начнем с еще одного определения.

Определение 8.3. Тривиальная многозначная зависимость

В переменной отношения r с атрибутами (возможно, составными) A и B MVD A→→B называется *тривиальной*, если либо $A \subseteq B$, либо A UNION B совпадает с заголовком отношения r. **Конец определения.**

Тривиальная MVD всегда удовлетворяется. При $A \subseteq B$ она вырождается в тривиальную FD. В случае A UNION B = Hr требования многозначной зависимости соблюдаются очевидным образом.

Для примера *n*-декомпозируемого отношения при $n > 2$ рассмотрим пятый вариант переменной отношения СЛУЖ_ПРО_ЗАДАН, в которой имеется единственно возможный ключ {СЛУ_НОМ, ПРО_НОМ, СЛУ_ЗАДАН} и отсутствуют нетривиальные MVD. Пример значения переменной отношения приведен на рис. 8.3.

Как показано на рис. 8.3, результат естественного соединения проекций СЛУЖ_ПРО_НОМ и ПРО_НОМ_ЗАДАН почти совпадает с телом исходного отношения СЛУЖ_ПРО_ЗАДАН, но в нем присутствует один лишний кортеж, который исчезнет после выполнения заключительного естественного соединения с проекцией СЛУЖ_ЗАДАНИЕ. Читателям предлагается убедиться, что исходное отношение будет восстановлено при любом порядке естественного соединения трех проекций.

Зависимость проекции/соединения

Утверждение о том, что тело отношения СЛУЖ_ПРО_ЗАДАН восстанавливается без потерь путем естественного соединения его проекций СЛУЖ_ПРО_НОМ, ПРО_НОМ_ЗАДАН и СЛУЖ_ЗАДАНИЕ эквивалентно следующему утверждению (ТСПЗ, ТСПН, ТПНЗ и ТСЗ обозначают тела значений переменных отношений СЛУЖ_ПРО_ЗАДАН, СЛУЖ_ПРО_НОМ, ПРО_НОМ_ЗАДАН и СЛУЖ_ЗАДАНИЕ соответственно):

```
IF (<СН, ПН> ∈ ТСПН AND <ПН, СЗ> ∈ ТПНЗ AND <СН, СЗ> ∈ ТСЗ)
THEN <СН, ПН, СЗ> ∈ ТСПЗ
```

Чтобы возможность восстановления без потерь отношения СЛУЖ_ПРО_ЗАДАН путем естественного соединения его проекций СЛУЖ_ПРО_НОМ, ПРО_НОМ_ЗАДАН и СЛУЖ_ЗАДАНИЕ существовала при любом допустимом значении переменной отношения СЛУЖ_ПРО_ЗАДАН, должно поддерживаться следующее ограничение:

```
IF (<СН1, ПН1, СЗ2> ∈ ТСПЗ  AND <СН2, ПН1, СЗ1> ∈ ТСПЗ
                             AND <СН1, ПН2, СЗ1> ∈ ТСПЗ)
THEN <СН1, ПН1, СЗ1> ∈ ТСПЗ
```

Это обычное ограничение реального мира, которое для отношения СЛУЖ_ПРО_ЗАДАН может быть сформулировано на естественном языке следующим образом:

***Если** служащий с номером СН участвует в проекте ПН, и в проекте ПН выполняется задание СЗ, и служащий с номером СН выполняет задание СЗ, **то** служащий с номером СН выполняет задание СЗ в проекте ПН.*

В общем виде такое ограничение называется *зависимостью проекции/соединения*. Вот формальное определение.

Определение 8.4. Зависимость проекции/соединения

Пусть задана переменная отношения r, и A, B, ..., Z являются произвольными подмножествами заголовка r (составными, перекрывающимися атрибутами). В переменной отношения r удовлетворяется зависимость

Возможное значение переменной отношения **СЛУЖ_ПРО_ЗАДАН** (пятый вариант)

СЛУ_НОМ	ПРО_НОМ	СЛУ_ЗАДАН
2934	1	A
2934	1	B
2934	2	A
2941	1	A

Значение переменной отношения **СЛУЖ_ПРО_НОМ** = (**СЛУЖ_ПРО_ЗАДАН** PROJECT {**СЛУ_НОМ**, **ПРО_НОМ**})

СЛУ_НОМ	ПРО_НОМ
2934	1
2934	2
2941	1

Значение переменной отношения **СЛУЖ_ЗАДАНИЕ** = (**СЛУЖ_ПРО_ЗАДАН** PROJECT {**ПРО_НОМ**, **СЛУ_ЗАДАН**})

ПРО_НОМ	СЛУ_ЗАДАН
1	A
1	B
2	A

Значение переменной отношения **СЛУЖ_ПРО_НОМ** = (**СЛУЖ_ПРО_ЗАДАН** PROJECT {**СЛУ_НОМ**, **СЛУ_ЗАДАН**})

СЛУ_НОМ	СЛУ_ЗАДАН
2934	A
2934	B
2941	A

Результат естественного соединения значений **СЛУЖ_ПРО_НОМ** NATURAL JOIN **ПРО_НОМ_ЗАДАН**

СЛУ_НОМ	ПРО_НОМ	СЛУ_ЗАДАН	
2934	1	A	
2934	1	B	
2934	2	A	
2941	1	A	Лишний кортеж
2941	1	B	←

Рис. 8.3. Возможное значение переменной отношения СЛУЖ_ПРО_ЗАДАН (пятый вариант), результаты проекций и результат частичного естественного соединения

проекции/соединения (Project-Join Dependency – PJD) *(A, B, ..., Z) тогда и только тогда, когда любое допустимое значение r можно получить путем естественного соединения проекций этого значения на атрибуты A, B, ..., Z. **Конец определения.**

Аномалии, вызываемые наличием зависимости проекции/соединения

В переменной отношения СЛУЖ_ПРО_ЗАДАН выполняется PJD *({СЛУ_НОМ, ПРО_НОМ}, {ПРО_НОМ, СЛУ_ЗАДАН}, {СЛУ_НОМ, СЛУ_ЗАДАН}). Наличие такой PJD обеспечивает возможность декомпозиции отношения на три проекции, но возникает вопрос, зачем это нужно? Чем плохо исходное отношение СЛУЖ_ПРО_ЗАДАН? Ответ обычный: этому отношению свойственны аномалии обновления. Для примера предположим, что значением СЛУЖ_ПРО_ЗАДАН является отношение, показанное на рис. 8.4.

- *Добавление кортежей.* Если к ТСП31 (рис. 8.4) добавляется кортеж <2941, 1, A>, то должен быть добавлен и кортеж <2934, 1, A>. Действительно, в теле отношения появятся кортежи <2934, 1, B>, <2941, 1, A> и <2934, 2, A>. Ограничение целостности требует включения и кортежа <2934, 1, A>. Интересно, что добавление кортежа <2934, 1, A> не нарушает ограничение целостности и, тем самым, не требует добавления кортежа <2941, 1, A>.

Возможное значение переменной отношения **СЛУЖ_ПРО_ЗАДАН** (ТСП31)

СЛУ_НОМ	ПРО_НОМ	СЛУ_ЗАДАН
2934	1	B
2934	2	A

Результат добавления к ТСП31 кортежа <2941, 1, A> (ТСП32)

СЛУ_НОМ	ПРО_НОМ	СЛУ_ЗАДАН
2934	1	B
2934	2	A
2941	1	A
2934	1	A

Рис. 8.4. Иллюстрации аномалий обновления в отношении СЛУЖ_ПРО_ЗАДАН при наличии зависимости соединения

- *Удаление кортежа.* Если из ТСП32 удаляется кортеж <2934, 1, A>, то должен быть удален и кортеж <2941, 1, A>, поскольку в соответствии с ограничением целостности наличие второго кортежа означает наличие первого. Интересно, что удаление кортежа <2941, 1, A> не нарушает ограничения целостности и не требует дополнительных удалений.

Устранение аномалий обновления в 3-декомпозиции

После выполнения декомпозиции трудности с обновлением автоматически снимаются. Действительно, декомпозируем отношение СЛУЖ_ПРО_ЗАДАН на три отношения: СЛУЖ_ПРО_НОМ {СЛУ_НОМ, ПРО_НОМ}, СЛУЖ_ЗАДАНИЕ {СЛУ_НОМ, СЛУ_ЗАДАН} и ПРО_НОМ_ЗАДАН {ПРО_НОМ, СЛУ_ЗАДАН}. Результат декомпозиции значения переменной отношения СЛУЖ_ПРО_ЗАДАН с телом ТСП31 показан в верхней части рис. 8.5.

Теперь если мы хотим добавить данные о служащем с номером 2941, выполняющем задание A в проекте 1, то, естественно, вставим кортеж <2941, 1> в отношение СОТР-ПРО_НОМ, кортеж <2941, A> в отношение СОТР-ЗАДАНИЕ и кортеж <1, A> в отношение ПРО_НОМ-ЗАДАН. Результат этих операций показан в средней части рис. 8.5.

Но если выполнить естественное соединение декомпозированных отношений с телами, полученными после добавления данных о служащем с номером 2941, выполняющем задание A в проекте 1, то будет получено значение-отношение с заголовком отношения СЛУЖ_ПРО_ЗАДАН и телом ТСП32 (нижняя часть рис. 8.5). Тем самым, проведенная декомпозиция позволила избежать сложностей при выполнении добавления кортежей с получением корректных результатов.

Аналогично можно проиллюстрировать простоту и корректность операций удаления кортежей.

Пятая нормальная форма

Отношения СЛУЖ_ПРО_НОМ, СЛУЖ_ЗАДАНИЕ и ПРО_НОМ_ЗАДАН находятся в пятой нормальной форме, но, прежде чем привести ее определение, нам требуется ввести еще два важных понятия.

Определение 8.5. Зависимость проекции/соединения, подразумеваемая возможными ключами

В переменной отношения *r* PJD *(*A*, *B*, ..., *Z*) называется подразумеваемой возможными ключами в том и только в том случае, когда каждый составной атрибут *A*, *B*, ..., *Z* является суперключом *r*, т. е. включает хотя бы один возможный ключ *r*. **Конец определения.**

Определение 8.6. Тривиальная зависимость проекции/соединения

В переменной отношения *r* зависимость проекции/соединения *(*A*, *B*, ..., *Z*) называется тривиальной, если хотя бы один из составных атрибутов *A*, *B*, ..., *Z* совпадает с заголовком *r*. **Конец определения.**

Результат декомпозиции переменной отношения СЛУЖ_ПРО_ЗАДАН с телом значения ТСПЗ1

СЛУЖ_ПРО_НОМ

СЛУ_НОМ	ПРО_НОМ
2934	1
2934	2

СЛУЖ_ЗАДАНИЕ

СЛУ_НОМ	СЛУ_ЗАДАН
2934	B
2934	A

ПРО_НОМ_ЗАДАН

ПРО_НОМ	СЛУ_ЗАДАН
1	B
2	A

Добавление данных о служащем с номером 2941, выполняющем задание A в проекте 1

СЛУЖ_ПРО_НОМ

СЛУ_НОМ	ПРО_НОМ
2934	1
2934	2
2941	1

СЛУЖ_ЗАДАНИЕ

СЛУ_НОМ	СЛУ_ЗАДАН
2934	B
2934	A
2941	A

ПРО_НОМ_ЗАДАН

ПРО_НОМ	СЛУ_ЗАДАН
1	B
2	A
1	A

Результат естественного соединения отношений после добавления данных

СЛУ_НОМ	ПРО_НОМ	СЛУ_ЗАДАН
2934	1	B
2934	2	A
2941	1	A
2934	1	A

Рис. 8.5. Иллюстрация декомпозиции отношения с зависимостью соединения

Легко убедиться, что нетривиальные PJD, подразумеваемые возможными ключами, существуют во всех отношениях с арностью, большей двух, первичный ключ которых не совпадает с заголовком отношения. Например, если в отношении СЛУЖ_ПРО_ЗАДАН атрибут СЛУ_НОМ является первичным ключом, то, очевидно, имеется PJD *({СЛУ_НОМ, ПРО_НОМ}, {СЛУ_НОМ, СЛУ_ЗАДАН}) (это следует из теоремы Хита). Но такие зависимости проекции/соединения неинтересны с точки зрения проектирования базы данных, поскольку не порождают аномалии обновления. Поэтому общепринятое определение пятой нормальной формы выглядит следующим образом.

Определение 8.7. Пятая нормальная форма

Переменная отношения *r* находится в пятой нормальной форме, или в нормальной форме проекции/соединения (5NF, или PJ/NF – Project-Join Normal Form) в том и только в том случае, когда каждая нетривиальная PJD в *r* подразумевается возможными ключами *r*. **Конец определения**.

Таким образом, чтобы распознать, что данная переменная отношения r находится в 5NF, необходимо знать все возможные ключи r и все PJD этой переменной отношения. Обнаружение всех зависимостей соединения является нетривиальной задачей, и для ее решения нет общих методов. Поэтому на практике проектирование реляционных баз методом нормализации обычно завершается после достижения 4NF, и отношения, находящиеся в 4NF, как правило, находятся и в 5NF. Зачем же тогда была введена эта туманная и труднодостижимая пятая нормальная форма?

Ответ на этот естественный вопрос состоит в том, что 5NF является «окончательной» нормальной формой, которой можно достичь в процессе нормализации на основе проекций. «Окончательность» понимается в том смысле, что у отношения, находящегося в 5NF, отсутствуют аномалии обновлений, которые можно было бы устранить путем его декомпозиции. Другими словами, такие отношения далее нормализовать бессмысленно.

Заключение

Процесс проектирования реляционной базы на основе метода нормализации преследует две основных цели:

- избежать избыточности хранения данных;
- устранить аномалии обновления отношений.

Рассмотрим, насколько эти цели актуальны в современных условиях, когда объемы доступных носителей внешней памяти непрерывно возрастают, стоимость их падает, а современные серверы реляционных баз данных способны автоматически поддерживать целостность баз данных. Здесь следует отметить два важных обстоятельства.

Во-первых, теория реляционных баз данных и методы их проектирования активно развивались уже более 25 лет тому назад. Ситуация в области технологии аппаратуры и программного обеспечения тогда была совсем иной, чем сегодня, и хорошо нормализованные реляционные базы данных в значительной степени способствовали росту эффективности приложений.

Во-вторых, в то время реляционные базы преимущественно использовались в информационных системах оперативной обработки транзакций (On-Line Transaction Processing – OLTP). Характерные примеры таких систем мы отмечали в лекции 1 – банковские системы, системы резервирования билетов и мест в гостиницах. Системам категории OLTP свойственны частые обновления базы данных, поэтому аномалии обновлений, даже если их корректировка производится СУБД автоматически, могут заметно снижать эффективность приложения.

Сегодня на переднем крае приложений баз данных находятся системы категории оперативной аналитической обработки (On-Line Analytical Processing – OLAP). В подобных системах, в частности, системах поддерж-

ки принятия решений, базы данных в основном используются для выборки данных, поэтому аномалиями обновлений можно пренебречь, а объем этих баз настолько огромен, что можно пренебречь и избыточностью хранения.

Значит ли это, что подход к проектированию реляционных баз данных методом нормализации утратил свою роль? Нет!

Мир приложений баз данных в настоящее время огромен. Сегодня любое мало-мальски приличное предприятие использует хотя бы одно приложение баз данных – бухгалтерские, складские, кадровые системы. Это системы категории OLTP с частым обновлением данных и умеренными запросами к базе данных, не вызывающими соединений многих отношений. Для небольших компаний равно важны как эффективность информационных систем, так и стоимость используемых аппаратно-программных средств. Правильно спроектированные, хорошо нормализованные реляционные базы данных помогают решению корпоративных проблем.

Да, любое правильно развивающееся предприятие рано или поздно приходит к использованию систем категории OLAP, например, некоторой разновидности систем поддержки принятия решений (Decision Support System – DSS). В базах данных таких систем обновления очень редки, а запросы могут иметь произвольную сложность, включая соединения многих отношений. Но, во-первых, технологически правильно для системы OLAP поддерживать отдельную базу данных (обычно подобные базы данных называют *хранилищами данных – DataWarehouse*), а во-вторых, основными источниками данных для построения таких хранилищ данных являются базы данных систем OLTP. Так что актуальность правильно спроектированных баз данных OLTP-систем не уменьшается, а постоянно возрастает.

Следует ли из этого, что принципы нормализации непригодны для проектирования баз данных OLAP-приложений? И снова в ответ категорическое НЕТ! Возможно, окончательная схема такой базы данных должна быть денормализована из соображений повышения эффективности выполнения запросов. Но чтобы получить правильную денормализованную схему, нужно сначала понять, как выглядит нормализованная схема.

Основной вывод этой и предыдущей лекций можно сформулировать следующим образом. Пока мы остаемся в мире реляционных баз данных, для правильного проектирования базы данных необходимо понимать принципы нормализации, воспринимая их не как догму, а как руководство к действию. Кстати, это относится не только к «ручному» проектированию реляционных баз данных, но и к их проектированию с применением семантических моделей данных и CASE-средств, которые мы обсудим в следующих двух лекциях.

Лекция 9. Проектирование реляционных баз данных с использованием семантических моделей: ER-диаграммы

В этой и следующей лекциях обсуждаются подходы к проектированию реляционных баз данных на основе использования семантических моделей данных. Лекции обеспечивают начальный уровень знаний в этой области и не заменяют книг, целиком посвященных данной теме. Настоящая лекция посвящена общему введению в семантические модели данных и краткому рассмотрению диаграммной семантической модели «Сущность-Связь». Анализируются стандартные приемы преобразования концептуальной схемы базы данных, представленной в терминах ER-модели, в реляционную схему.

Ключевые слова: семантическая модель данных, концептуальная схема базы, системы автоматизации проектирования баз данных, семантическая модель Entity-Relationship (сущность-связь), сущность, тип сущности, экземпляр типа сущности, связь, тип связи, экземпляр типа связи, бинарная связь, рекурсивная связь, конец связи, роль связи, степень конца связи, обязательность связи, атрибут сущности, уникальный идентификатор типа сущности, нормальные формы ER-диаграмм, первая нормальная форма ER-диаграммы, вторая нормальная форма ER-диаграммы, третья нормальная форма ER-диаграммы, подтипы и супертипы сущностей, уточняемые степени связи, взаимно исключающие связи, каскадные удаления экземпляров сущностей, получение реляционной схемы из ER-диаграммы, представление в реляционной схеме супертипов и подтипов сущности, представление в реляционной схеме взаимно исключающих связей.

Введение

Широкое распространение реляционных СУБД и их использование в самых разнообразных приложениях показывает, что реляционная модель данных достаточна для моделирования разнообразных предметных областей. Однако проектирование реляционной базы данных в терминах отношений на основе кратко рассмотренного нами в двух предыдущих лекциях механизма нормализации часто представляет собой очень сложный и неудобный для проектировщика процесс.

Ограниченность реляционной модели при проектировании баз данных

При использовании в проектировании ограниченность реляционной модели проявляется в следующих аспектах.

- Модель не обеспечивает достаточных средств для представления смысла данных. Семантика реальной предметной области должна независимым от модели способом представляться в голове проектировщика. В частности, это относится к отмечавшейся нами ранее проблеме представления ограничений целостности, выходящих за пределы ограничений первичного и внешнего ключа.
- Во многих прикладных областях трудно моделировать предметную область на основе плоских таблиц*. В ряде случаев на самой начальной стадии проектирования дизайнеру приходится нелегко, поскольку от него требуется описать предметную область в виде одной (возможно, даже ненормализованной) таблицы.
- Хотя весь процесс проектирования происходит на основе учета зависимостей, реляционная модель не предоставляет какие-либо формализованные средства для представления этих зависимостей.
- Несмотря на то, что процесс проектирования начинается с выделения некоторых существенных для приложения объектов предметной области («сущностей») и выявления связей между этими сущностями, реляционная модель данных не предлагает какого-либо механизма для разделения сущностей и связей**.

Семантические модели данных

Потребность проектировщиков баз данных в более удобных и мощных средствах моделирования предметной области вызвала к жизни направление семантических моделей данных. Хотя любая развитая семантическая модель данных, как и реляционная модель, включает структурную, манипуляционную и целостную части, главным назначением семантических моделей является обеспечение возможности выражения семантики данных.

---

* Начиная с этой лекции, мы переходим к использованию терминов *таблица*, *строка* и *столбец* вместо строгих реляционных терминов *отношение*, *атрибут* и *таблица*, поскольку здесь под «реляционными» базами данных понимаются, главным образом, SQL-ориентированные базы данных, для которых эта упрощенная терминология более естественна.

** Многие сторонники реляционного подхода считают отсутствие раздельного представления сущностей и связей преимуществом реляционной модели данных, мотивируя это тем, что зачастую то, что вчера считалось сущностью, сегодня разумнее принять за связь, и наоборот. Это, безусловно, верно с точки зрения поддержки и модификации существующих реляционных баз данных, но отнюдь не так с точки зрения проектирования базы данных.

Прежде чем мы коротко рассмотрим особенности двух распространенных семантических моделей, остановимся на возможных областях их применения. Чаще всего на практике семантическое моделирование используется на первой стадии проектирования базы данных. При этом в терминах семантической модели производится концептуальная схема базы данных, которая затем вручную преобразуется к реляционной (или какой-либо другой) схеме. Этот процесс выполняется под управлением методик, в которых достаточно четко оговорены все этапы такого преобразования. Основным достоинством данного подхода является отсутствие потребности в дополнительных программных средствах, поддерживающих семантическое моделирование. Требуется только знание основ выбранной семантической модели и правил преобразования концептуальной схемы в реляционную схему.

Следует заметить, что многие начинающие проектировщики баз данных недооценивают важность семантического моделирования вручную. Зачастую это воспринимается как дополнительная и излишняя работа. Эта точка зрения абсолютно неверна. Во-первых, построение мощной и наглядной концептуальной схемы БД позволяет более полно оценить специфику моделируемой предметной области и избежать возможных ошибок на стадии проектирования схемы реляционной БД. Во-вторых, на этапе семантического моделирования производится важная документация (хотя бы в виде вручную нарисованных диаграмм и комментариев к ним), которая может оказаться очень полезной не только при проектировании схемы реляционной БД, но и при эксплуатации, сопровождении и развитии уже заполненной БД.

Неоднократно приходилось и приходится наблюдать ситуации, в которых отсутствие такого рода документации существенно затрудняет внесение даже небольших изменений в схему существующей реляционной БД. Конечно, это относится к случаям, когда проектируемая БД содержит не слишком малое число таблиц. Скорее всего, без семантического моделирования можно обойтись, если число таблиц не превышает десяти, но оно совершенно необходимо, если БД включает более сотни таблиц. Для справедливости заметим, что процедура создания концептуальной схемы вручную с ее последующим преобразованием в реляционную схему БД затруднительна в случае больших БД (содержащих несколько сотен таблиц). Причины, по всей видимости, не требуют пояснений.

История систем автоматизации проектирования баз данных (CASE-средств*) началась с автоматизации процесса рисования диа-

* Позволю себе одно терминологическое замечание, которое может показаться несколько наивным для специалистов в области инженерии программного обеспечения (software engineering), к числу которых я не принадлежу. Издавна существует отдельный класс программных систем, предназначенных для автоматизации проектирования новых продуктов

грамм, проверки их формальной корректности, обеспечения средств долговременного хранения диаграмм и другой проектной документации. Конечно, компьютерная поддержка работы с диаграммами для проектировщика БД очень полезна. Наличие электронного архива проектной документации помогает при эксплуатации, администрировании и сопровождении базы данных. Но система, которая ограничивается поддержкой рисования диаграмм, проверкой их корректности и хранением, напоминает текстовый редактор, поддерживающий ввод, редактирование и проверку синтаксической корректности конструкций некоторого языка программирования, но существующий отдельно от компилятора. Кажется естественным желание расширить такой редактор функциями компилятора, и это действительно возможно, поскольку известна техника компиляции конструкций языка программирования в коды целевого компьютера. Но коль скоро имеется четкая методика преобразования концептуальной схемы БД в реляционную схему, то почему бы не выполнить программную реализацию соответствующего «компилятора» и не включить ее в состав системы проектирования баз данных?

Эта идея, естественно, показалась разумной производителям CASE-средств проектирования БД. Подавляющее большинство подобных систем, представленных на рынке, обеспечивает автоматизированное преобразование диаграммных концептуальных схем баз данных, представленных в той или иной семантической модели данных, в реляционные схемы, специфицированные чаще всего на языке SQL. У читателя может возникнуть вопрос, почему в предыдущем предложении говорится про «автоматизированное», а не про «автоматическое» преобразование? Все дело в том, что в типичной схеме SQL-ориентированной БД могут содержаться определения многих объектов (ограничений целостности общего

в разных областях промышленности – автомобилестроении, аэрокосмической промышленности, электронной промышленности и т.д. Очевидно, что процесс проектирования автомобиля принципиально отличается от процесса проектирования микропроцессора, но, тем не менее, для обозначения любой Системы Автоматизации ПРоектирования используется собирательный термин САПР (CAD – Computer Aided Design). Это оправдывается тем, что разные подклассы САПР имеют гораздо больше общих черт, чем различий. Так вот, по моему мнению, система автоматизации проектирования БД по своему назначению и строению в большей степени является системой класса САПР, чем системой класса CASE (Computer Aided Software Engineering). По всей видимости, средства автоматизированной поддержки проектирования баз данных стали в свое время называть CASE-средствами, поскольку они обычно включали не только инструменты для поддержки проектирования, но и инструменты, поддерживающие проектирование и разработку приложений баз данных. В последние годы такие инструменты все реже производятся в виде одного пакета, и сам термин «CASE-средство» почти вышел из употребления. Тем не менее, поскольку не появилось какое-либо другое собирательное название средств поддержки проектирования баз данных, мы будем продолжать использовать именно этот термин.

вида, триггеров и хранимых процедур и т. д.), которые невозможно сгенерировать автоматически на основе концептуальной схемы. Поэтому на завершающем этапе проектирования реляционной схемы снова требуется ручная работа проектировщика.

Еще раз обратите внимание на то, какой ход рассуждений привел нас к выводу о возможности автоматизации процесса преобразования концептуальной схемы БД в реляционную схему. Если создатели семантической модели данных предоставляют методику преобразования концептуальных схем в реляционные, то почему бы не реализовать программу, которая производит те же преобразования, следуя той же методике? Зададимся теперь другим, но, по существу, схожим вопросом. Если создатели семантической модели данных предоставляют язык (например, диаграммный), используя который проектировщики БД на основе исходной информации о предметной области могут сформировать концептуальную схему БД, то почему бы не реализовать программу, которая сама генерирует концептуальную схему БД в соответствующей семантической модели, используя исходную информацию о предметной области? Хотя нам не известны коммерческие CASE-средства проектирования БД, поддерживающие такой подход, экспериментальные системы успешно существовали. Они представляли собой интегрированные системы проектирования с автоматизированным созданием концептуальной схемы на основе интервью с экспертами предметной области и последующим преобразованием концептуальной схемы в реляционную.

Как правило, CASE-средства, автоматизирующие преобразование концептуальной схемы БД в реляционную, производят реляционную схему базы данных в третьей нормальной форме. Нормализация более высокого уровня усложняет программную реализацию и редко требуется на практике.

Наконец, третья возможность, которую следует упомянуть, хотя она еще не вышла (или только выходит, а может быть, так никогда и не выйдет) за пределы исследовательских и экспериментальных проектов, – это работа с базой данных в семантической модели, т. е. СУБД, основанные на семантических моделях данных. При этом снова рассматриваются два варианта: обеспечение пользовательского интерфейса на основе семантической модели данных с автоматическим отображением конструкций этого интерфейса в реляционную модель данных (это задача примерно того же уровня сложности, что и автоматическая компиляция концептуальной схемы базы данных в реляционную схему) и прямая реализация СУБД, основанная на какой-либо семантической модели данных. Многие авторитетные специалисты полагают, что ближе всего ко второму подходу объектно-ориентированные СУБД, чьи модели данных по многим параметрам близки к семантическим моделям (хотя в некоторых аспектах они более мощны, а в некоторых – более слабы).

Семантическая модель Entity-Relationship (Сущность-Связь)

В этой лекции мы кратко рассмотрим некоторые черты одной из наиболее популярных семантических моделей данных – модели «Сущность-Связь» (часто ее называют кратко ER-моделью от *Entity-Relationship*).

Здесь следует сделать два замечания, касающиеся, главным образом, терминологии. Оба термина *relation* и *relationship* могут быть переведены на русский язык как *отношение*. Поэтому в русскоязычной литературе ER-модель иногда называют моделью *сущность-отношение*, а иногда и реляционной семантической моделью. Наверное, в этом нет ничего страшного, если говорить о ER-модели в отрыве от тематики проектирования реляционных баз данных.

Но если требуется одновременно использовать термины ER-модели и реляционной модели данных, то, безусловно, требуется применять для терминов *relation* и *relationship* разные русские эквиваленты. За этими терминами стоят весьма различные понятия. В реляционной модели отношение (relation) – это единственная родовая структура данных. С помощью этого же механизма представляются «связанные» сущности (вспомните, например, про внешние ключи). Как мы увидим немного позже, в ER-модели для представления схемы базы данных используются два равноправных понятия – *сущность* и *связь*. Связи в ER-модели играют роль, отличную от той, какую играют отношения в реляционной модели данных.

Кроме того, в русскоязычную терминологию вошла и чистая транслитерация термина *relation* именно в смысле *отношение*. Мы говорим, например, про реляционную модель данных, реляционную алгебру и т. д., понимая модель данных, основанную на отношениях, алгебру отношений и т. п. По этому поводу, по крайней мере, в контексте баз данных, разумно окончательно зарезервировать термины *relation* и *отношение* для обозначения понятий реляционной модели данных, а для термина *relationship* использовать другой допустимый русскоязычный эквивалент – *связь*.

На использовании разных вариантов ER-модели основано большинство современных подходов к проектированию баз данных (главным образом, реляционных). Модель была предложена Питером Ченом (Peter Chen) в 1976 г. Моделирование предметной области базируется на использовании графических диаграмм, включающих небольшое число разнородных компонентов. Простота и наглядность представления концептуальных схем баз данных в ER-модели привели к ее широкому распространению в CASE-системах, поддерживающих автоматизированное проектирование реляционных баз данных. Среди множества разновидностей ER-моделей одна из наиболее популярных и развитых применялась в системе CASE компании

Oracle. Мы обсудим некоторый упрощенный вариант этой модели. Если говорить более точно, сосредоточимся на ее структурной и целостной частях.

Основные понятия ER-модели

Основными понятиями ER-модели являются *сущность*, *связь* и *атрибут*. *Сущность* – это реальный или представляемый объект, информация о котором должна сохраняться и быть доступной.* В диаграммах ER-модели сущность представляется в виде прямоугольника, содержащего имя сущности. При этом имя сущности – это имя типа, а не некоторого конкретного экземпляра этого типа.** Для большей выразительности и лучшего понимания имя сущности может сопровождаться примерами конкретных экземпляров этого типа.

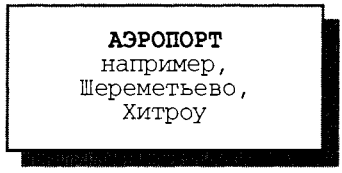


Рис. 9.1. Пример типа сущности

На рис. 9.1 изображена сущность АЭРОПОРТ с примерными экземплярами «Шереметьево» и «Хитроу». Эта примитивная диаграмма тем не менее несет важную информацию. Во-первых, она показывает, что в базе данных будут содержаться однотипные структуры данных (экземпляры сущности), описывающие аэропорты. Во-вторых, поскольку в жизни существует несколько точек зрения на аэропорты (например, точка зрения пилота, точка зрения пассажира, точка зрения администратора) и этим точкам зрения соответствуют разные структуры данных, то приведенные примеры аэропортов позволяют несколько сузить допустимый набор точек зрения. В нашем случае приведены примеры международных аэропортов, так что, скорее всего, имеется точка зрения пассажира или пилота международных авиарейсов.

* Понятно, что это «определение» на самом деле является тавтологией, поскольку, во-первых, мы пытаемся определить термин *сущность* через не определенный термин *объект*, а во-вторых, попытки определения термина *объект* настолько же безнадежны. Обычно авторы пытаются оправдываться тем, что в подобном контексте они имеют в виду «житейское», а не сколько-нибудь формализованное понятие *объекта*. Конечно, от этого не становится легче, поскольку понятие *сущности* должно пониматься в достаточно точном смысле. Но эта тавтология не изобретена автором этого курса; она традиционна для области семантического моделирования. В этой области стремятся максимально избегать формальностей.

** Хотя было бы правильнее всегда использовать термины *тип сущности* и *экземпляр типа сущности*, для избежания многословности (и следуя традиции) в тех случаях, где это не приводит к двусмысленности, мы будем использовать термин *сущность* в значении *типа сущности*.

При определении типа сущности необходимо гарантировать, что каждый экземпляр сущности может быть отличим от любого другого экземпляра той же сущности. Это требование в некотором роде аналогично требованию отсутствия кортежей-дубликатов в реляционных таблицах.

Связь – это графически изображаемая ассоциация, устанавливаемая между двумя типами сущностей. Как и сущность, связь – это типовое понятие, все экземпляры обоих связываемых типов сущностей подчиняются устанавливаемым правилам связывания. Поэтому правильнее говорить о типе связи, устанавливаемой между типами сущности, и об экземплярах типа связи, устанавливаемых между экземплярами типа сущности.* В обсуждаемом здесь варианте ER-модели эта ассоциация всегда является бинарной и может существовать между двумя разными типами сущностей или между типом сущности и им же самим (рекурсивная связь). В любой связи выделяются два конца (в соответствии с существующей парой связываемых сущностей), на каждом из которых указываются имя конца связи, степень конца связи (сколько экземпляров данного типа сущности должно присутствовать в каждом экземпляре данного типа связи), обязательность связи (т. е. любой ли экземпляр данного типа сущности должен участвовать в некотором экземпляре данного типа связи).**

Связь представляется в виде ненаправленной линии, соединяющей две сущности или ведущей от сущности к ней же самой. При этом в месте «стыковки» связи с сущностью используются:

- трехточечный вход в прямоугольник сущности, если для этой сущности в связи могут (или должны) использоваться много (*many*) экземпляров сущности;
- одноточечный вход, если в связи может (или должен) участвовать только один экземпляр сущности.

Обязательный конец связи изображается сплошной линией, а необязательный – прерывистой линией.

Связь между сущностями БИЛЕТ и ПАССАЖИР, показанная на рис. 9.2, связывает билеты и пассажиров. Конец связи с именем «для» позволяет связывать с одним пассажиром более одного билета, причем каждый би-

Рис. 9.2. Пример типа связи

* Тем не менее, как и в случае типа сущности, мы будем часто использовать термин *связь* в значении *типа связи*.

** В некоторых вариантах ER-модели конец связи называют ролью связи в данной сущности. Тогда можно говорить об *имени роли*, *степени роли* и *обязательности роли* связи в данной сущности.

лет должен быть связан с каким-либо пассажиром. Конец связи с именем «имеет» показывает, что каждый билет может принадлежать только одному пассажиру, причем пассажир не обязан иметь хотя бы один билет.

Лаконичная устная трактовка изображенной диаграммы состоит в следующем:

- каждый БИЛЕТ предназначен для одного и только одного ПАССАЖИРА;
- каждый ПАССАЖИР может иметь один или более БИЛЕТОВ.

На следующем примере (рис. 9.3) изображена рекурсивная связь, связывающая сущность МУЖЧИНА с ней же самой. Конец связи с именем «сын» определяет тот факт, что несколько людей могут быть сыновьями одного отца. Конец связи с именем «отец» означает, что не у каждого мужчины должны быть сыновья.

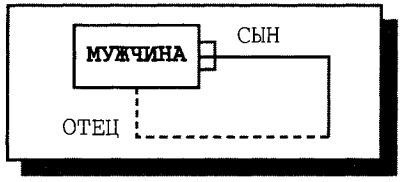


Рис. 9.3. Пример рекурсивного типа связи

Лаконичная устная трактовка изображенной диаграммы состоит в следующем:

- каждый МУЖЧИНА является сыном одного и только одного МУЖЧИНЫ;
- каждый МУЖЧИНА может являться отцом одного или более МУЖЧИН.

Атрибутом сущности является любая деталь, которая служит для уточнения, идентификации, классификации, числовой характеристики или выражения состояния сущности. Имена атрибутов заносятся в прямоугольник, изображающий сущность, под именем сущности и изображаются малыми буквами, возможно, с примерами.

Пример типа сущности ЧЕЛОВЕК с указанными атрибутами показан на рис. 9.4. С технической точки зрения атрибуты типа сущности в ER-модели похожи на атрибуты отношения в реляционной модели данных. И в том, и в другом случаях введение именованных атрибутов вводит некоторую типо-

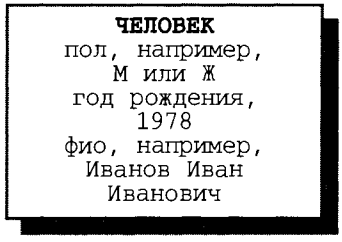


Рис. 9.4. Пример типа сущности с атрибутами

вую структуру данных, имя которой совпадает с именем типа сущности в случае ER-модели или с именем переменной отношения в случае реляционной модели. Этой типовой структуре должны следовать все экземпляры типа сущности или все кортежи отношения. Но имеется и важное отличие. Напомним, что в реляционной модели данных атрибут определяется как упорядоченная пара <имя_атрибута, имя_домена> (или <имя_атрибута, имя_базового_типа_данных>, если понятие домена не поддерживается). Заголовок отношения, определяемый как множество таких пар, представляет собой полный аналог структурного типа данных в языках программирования.

При определении атрибутов типа сущности в ER-модели указание домена атрибута не является обязательным, хотя это и возможно (см. ниже). Обсудим, чем вызвана эта возможность «ослабленного» определения атрибутов. Прежде всего, как отмечалось в разделе «Введение», семантические модели данных используются для построения концептуальных схем БД, и эти схемы преобразуются в реляционные схемы БД, которые поддерживаются той или иной СУБД. Несмотря на то что в настоящее время типовые возможности РСУБД в основном стандартизованы (на основе стандарта языка SQL), детали базового набора типов данных и средств определения доменов в разных системах могут различаться. Поскольку производители CASE-средств проектирования реляционных БД стремятся не связывать обеспечиваемые ими возможности семантического моделирования с конкретной реализацией СУБД, они стимулируют откладывание строгого определения типов атрибутов до стадии полного определения реляционной схемы.

Кроме того, напомним, что при определении атрибута отношения допускается использование имен атрибутов, совпадающих с именами своих доменов (это два разных пространства имен, и наличие одинаковых имен у атрибутов и доменов не вызывает коллизий). Поэтому при определении атрибутов типов сущности можно так подбирать их имена, что они в дальнейшем будут подсказывать, какие домены у этих атрибутов имеются в виду. Пониманию предполагаемой сути доменов способствует и возможность указания примеров значений атрибутов. Например, на рис. 9.4 имеется атрибут год рождения, в качестве примерного значения которого указано «1976». Это подсказывает, что в реляционной схеме при определении соответствующего атрибута наиболее естественным базовым типом данных будет темпоральный тип «ДАТА», значения которого задают дату с точностью до года.

Уникальные идентификаторы типов сущности

Как отмечалось выше, при определении типа сущности необходимо гарантировать, что каждый экземпляр сущности является отличимым от любого другого экземпляра той же сущности. Поскольку сущность являет-

ся абстракцией реального или представляемого объекта внешнего мира, это требование нужно иметь в виду уже при выборе кандидата в типы сущности. Например, предположим, что проектируется база данных для поддержки работы книжного склада. На складе могут храниться произвольные части тиража любого издания любой книги. Может ли в этом случае индивидуальная книга являться прообразом типа сущности? Утверждается, что нет, поскольку отсутствует возможность различения книг одного издания. Для книжного склада прообразом типа сущности будет набор одноименных книг одного автора, вышедших в одном издании. Одним из атрибутов этого типа сущности будет `число книг в наборе`. Но когда книга поступает в библиотеку и ей присваивается уникальный библиотечный номер, она становится разумным прообразом типа сущности. Плохо устроены библиотеки, в которых не различаются индивидуальные книги (даже одноименные книги одного автора, вышедшие в одном издании).

Но при проектировании базы данных мало того, чтобы проектировщик убедился в правильном выборе типов сущности, гарантирующем различие экземпляров каждого типа сущности. Необходимо сообщить системе автоматизации проектирования БД, каким образом будут различаться эти экземпляры, т. е. сообщить, как конструируются уникальные идентификаторы экземпляров каждого типа сущности. В ER-модели у экземпляра типа сущности не может быть назначаемого пользователем имени или назначаемого системой внешнего уникального идентификатора. Экземпляр типа сущности может идентифицироваться только своими индивидуальными характеристиками, а они представляются значениями атрибутов и экземплярами типов связи, связывающими данный экземпляр типа сущности с экземплярами других типов сущности или этого же типа сущности. Поэтому уникальным идентификатором сущности может быть атрибут, комбинация атрибутов, связь, комбинация связей или комбинация связей и атрибутов, уникально отличающая любой экземпляр сущности от других экземпляров сущности того же типа.

Приведем несколько примеров. На рис. 9.5 показан тип сущности `КНИГА`, пригодный для использования в базе данных книжного склада.

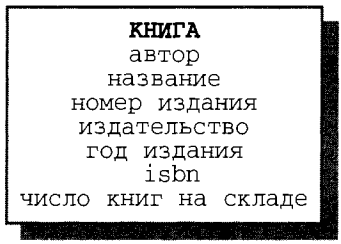


Рис. 9.5. Тип сущности, экземпляры которого идентифицируются атрибутами

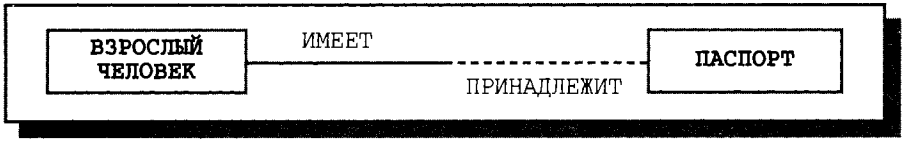


Рис. 9.6. Тип сущности, экземпляры которой идентифицируются связью

При издании любой книги в любом издательстве (кроме пиратских, которыми мы для простоты пренебрежем) ей присваивается уникальный номер – ISBN. Понятно, что значение атрибута `isbn` будет уникально идентифицировать партию книг на складе. Кроме того, конечно, в качестве уникального идентификатора годится и комбинация атрибутов `<автор, название, номер издания, издательство, год издания>`.

На рис. 9.6 диаграмма включает два связанных типа сущности. У каждого взрослого человека имеется один и только один паспорт (мы снова не берем в расчет особый случай, когда у одного человека имеется несколько паспортов), и каждый паспорт может принадлежать только одному взрослому человеку (некоторые уже готовые паспорта могут быть еще никому не выданы). Тогда связь человека с его паспортом (конец связи ИМЕЕТ) уникально идентифицирует взрослого человека, т. е., грубо говоря, паспорт определяет взрослого человека. Поскольку могут существовать паспорта, еще не выданные какому-либо человеку, эта связь не является уникальным идентификатором сущности ПАСПОРТ.

На рис. 9.7 диаграмма включает три связанных типа сущности. Профессора обладают знаниями в нескольких учебных дисциплинах. Преподавание каждой дисциплины доступно нескольким профессорам. Другими словами, между сущностями ПРОФЕССОР и ДИСЦИПЛИНА определена связь «многие ко многим». Каждый профессор может готовить курсы по любой доступной ему дисциплине. Каждой дисциплине может быть по-

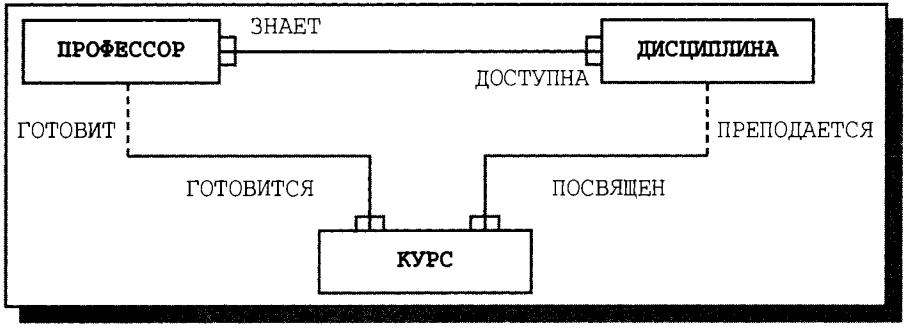


Рис. 9.7. Тип сущности, экземпляры которой идентифицируются комбинацией связей

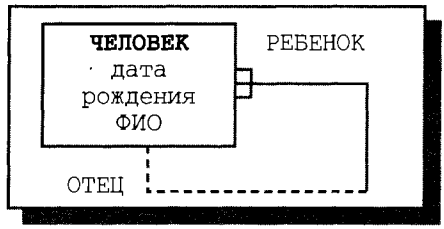


Рис. 9.8. Тип сущности, экземпляры которой идентифицируются комбинацией атрибутов и связей

священо несколько учебных курсов. Но каждый профессор может готовить только один курс по любой доступной ему дисциплине, и каждый курс может быть посвящен только одной дисциплине. Тем самым, каждый экземпляр типа сущности КУРС уникально идентифицируется экземпляром сущности ПРОФЕССОР и экземпляром сущности ДИСЦИПЛИНА, т. е. парой связей с именами концов ГОТОВИТСЯ и ПОСВЯЩЕН на стороне сущности КУРС. Заметим, что сущности ПРОФЕССОР и ДИСЦИПЛИНА связями не идентифицируются.

Наконец, на рис. 9.8 приведен пример типа сущности, уникальный идентификатор которого является комбинацией атрибутов и связей. Это несколько уточненный вариант сущности с рекурсивной связью с рис. 9.3. У каждого человека могут быть дети, и у каждого человека имеется отец. Тогда, если предположить, что близнецам, появившимся на свет одновременно, не дают одинаковых имен, то уникальным идентификатором типа сущности ЧЕЛОВЕК может быть комбинация атрибутов <дата рождения, ФИО> и связь с именем конца РЕБЕНОК.

Нормальные формы ER-диаграмм

Как и в случае схем реляционных баз данных, для ER-диаграмм вводится понятие нормальных форм, причем их смысл очень близко соответствует смыслу нормальных форм отношений. Заметим, что определения нормальных форм ER-диаграмм делают более понятным смысл нормализации схем отношений. Мы приведем только очень краткие и неформальные определения трех первых нормальных форм. Конечно, можно было бы ввести дальнейшие нормальные формы ER-диаграмм, аналогичные нормальной форме Бойса-Кодда, 4NF и 5NF, но на практике к такой нормализации обычно не прибегают, а общие идеи после ознакомления с лекцией 8 должны быть понятны и так.

му, могут использоваться разные транспортные предприятия, и каждое транспортное предприятие может обслуживать несколько аэродромов.

Чем плоха эта ситуация? Прежде всего, тем, что скрывается тот факт, что авиаремонтное предприятие ремонтирует самолеты, а не аэродромы. Наша же связь на самом деле означает, что любой аэродром из группы аэродромов обслуживается любым авиаремонтным предприятием из группы таких предприятий. Проблема состоит именно в том, что значением атрибута «самолеты» является множество экземпляров типа сущности САМОЛЕТ, и этот тип сущности сам обладает атрибутами и связями.

Ситуацию исправляет ER-диаграмма, показанная на рис. 9.9 (b). Здесь мы выделили тип сущности САМОЛЕТ. Связь между сущностями АЭРОДРОМ и САМОЛЕТ показывает, что к одному аэродрому приписывается несколько самолетов. Связь между сущностями САМОЛЕТ и АВИАРЕМОНТНОЕ ПРЕДПРИЯТИЕ означает, что каждый самолет из группы самолетов (группу самолетов могут составлять, например, все самолеты одного типа) обслуживается любым транспортным предприятием из некоторой группы таких предприятий. ER-диаграмма на рис. 9.9 (b) находится в первой нормальной форме и, как мы видим, лучше отображает реальную ситуацию.

Вторая нормальная форма ER-диаграммы

Во второй нормальной форме устраняются атрибуты, зависящие только от части уникального идентификатора. Эта часть уникального идентификатора определяет отдельную сущность.

На рис. 9.10 (a) показана диаграмма, на которой тип сущности ЭЛЕМЕНТ РАСПИСАНИЯ не удовлетворяет требованиям второй нормальной формы. На этой диаграмме у сущности ЭЛЕМЕНТ РАСПИСАНИЯ имеются следующие свойства. Элементы расписания предназначены для сохранения данных о рейсах самолетов, вылетающих в течение дня. Некоторыми важными характеристиками рейса являются номер рейса, аэропорт вылета, аэропорт назначения, дата и время вылета, бортовой номер самолета, тип самолета. Если говорить про российские авиационные компании, то (1) у каждого рейса имеется заранее приписанный ему номер (уникальный среди всех других имеющихся номеров рейсов), (2) не все рейсы совершаются каждый день, поэтому характеристикой конкретного рейса является дата и время его совершения, (3) бортовой номер самолета определяется парой <номер рейса, дата-время вылета>. Имеется связь «многие к одному» между сущностями ЭЛЕМЕНТ РАСПИСАНИЯ и ГОРОД. Экземпляры типа сущности ГОРОД характеризуют город, в который прибывает данный рейс.

Уникальным идентификатором типа сущности ЭЛЕМЕНТ РАСПИСАНИЯ является пара атрибутов <номер рейса, дата-время вылета>. Если вер-

Первая нормальная форма ER-диаграммы

В первой нормальной форме ER-диаграммы устраняются атрибуты, содержащие множественные значения, т. е. производится выявление неявных сущностей, «замаскированных» под атрибуты.

На рис. 9.9 (а) показана диаграмма, в которой тип сущности АЭРОДРОМ не удовлетворяет требованию первой нормальной формы. Здесь для нас несущественны атрибуты сущности АВИАРЕМОНТНОЕ ПРЕДПРИЯТИЕ, но сущность АЭРОДРОМ помимо атрибутов, отражающих собственные характеристики аэродромов (длина взлетно-посадочной полосы, число ангаров и т.д.) содержит атрибут, множественное значение которого характеризует самолеты, приписанные к этому аэродрому. Очевидно, что самолеты нуждаются в ремонте, т. е. должны обслуживаться некоторым авиаремонтным предприятием. Но поскольку самолеты являются частью сущности АЭРОДРОМ, единственным способом фиксации этого факта на диаграмме является проведение связи «многие ко многим» между типами сущности АЭРОДРОМ и АВИАРЕМОНТНОЕ ПРЕДПРИЯТИЕ. Таким образом выражается то соображение, что для ремонта разных самолетов, приписанных к одному аэродро-

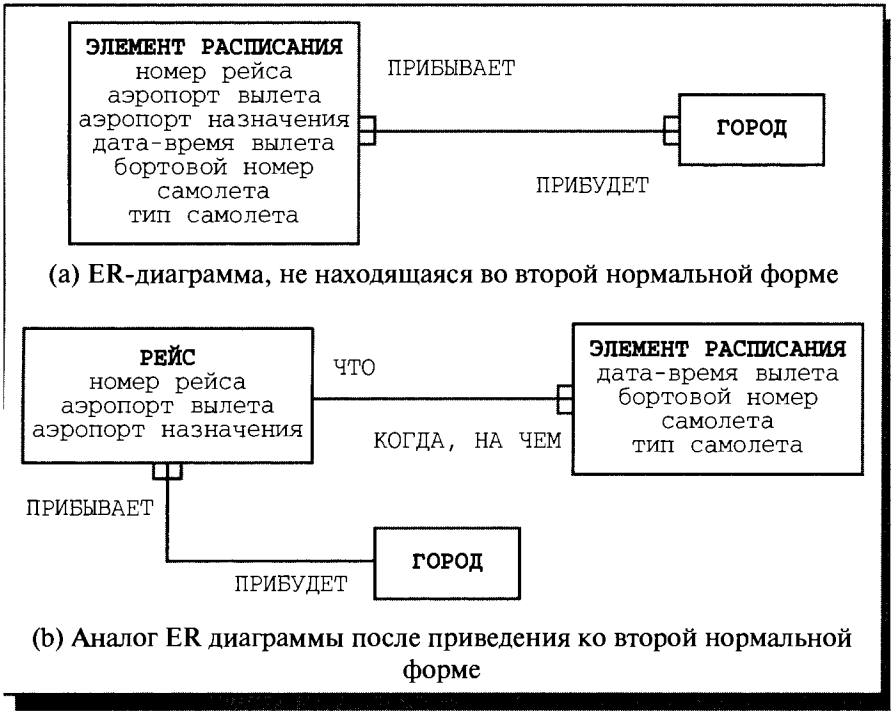
Рис. 9.9. Пример приведения ER-диаграммы к первой нормальной форме

нуться к терминам функциональных зависимостей, то между атрибутами этой сущности имеются следующие FD:

- {номер рейса, дата-время вылета}→бортовой номер самолета;
- номер рейса → аэропорт вылета;
- номер рейса → аэропорт назначения;
- бортовой номер самолета → тип самолета.

Кроме того, очевидно, что каждый экземпляр связи с сущностью ГОРОД также определяется значением атрибута номер рейса. Налицо нарушение требования второй нормальной формы. Мы получаем не только избыточное хранение значений атрибутов аэропорт вылета и аэропорт назначения в каждом экземпляре типа сущности ЭЛЕМЕНТ РАСПИСАНИЯ с одним и тем же значением номера рейса. Искажается и затемняется смысл связи с сущностью ГОРОД. Можно подумать, что в разные дни один и тот же рейс прибывает в разные города.

На рис. 9.10 (b) показан нормализованный вариант диаграммы, в котором все сущности находятся во второй нормальной форме. Теперь имеются три типа сущности: РЕЙС с атрибутами номер рейса, аэропорт



(a) ER-диаграмма, не находящаяся во второй нормальной форме

(b) Аналог ER диаграммы после приведения ко второй нормальной форме

Рис. 9.10. Пример приведения ER-диаграммы ко второй нормальной форме

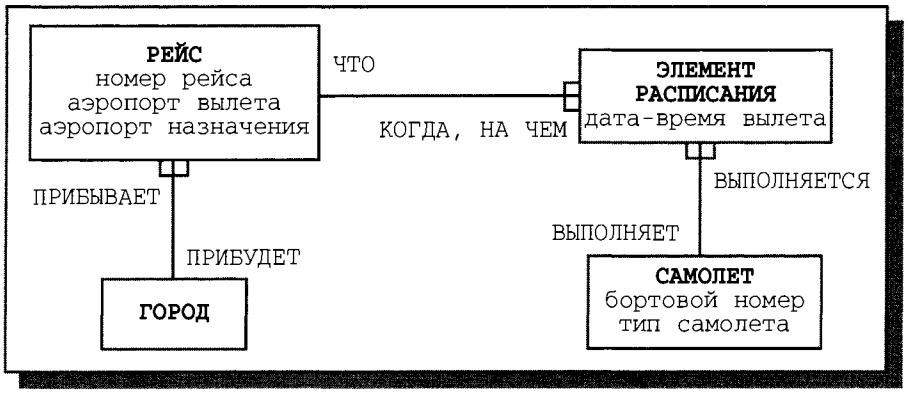


Рис. 9.11. Пример приведения ER-диаграммы к третьей нормальной форме

ках объектно-ориентированного программирования), в ER-модели поддерживается возможность определения нового типа сущности путем наследования некоторого супертипа сущности. Механизм наследования в ER-модели обладает несколькими особенностями: в частности, интересные нюансы связаны с необходимостью графического изображения этого механизма (см. ниже).

- *Уточняемые степени связи.* Иногда бывает полезно определить возможное количество экземпляров сущности, участвующих в данной связи (например, ввести ограничение, связанное с тем, что служащему разрешается участвовать не более чем в трех проектах одновременно). Для выражения этого семантического ограничения разрешается указывать на конце связи ее максимально допустимую или обязательную степень.
- *Взаимно исключающие связи.* Для заданного типа сущности можно определить такой набор типов связи с другими типами сущности, что для каждого экземпляра заданного типа сущности может (если набор связей является необязательным) или должен (если набор связей обязателен) существовать экземпляр только одной связи из этого набора.
- *Каскадные удаления экземпляров сущностей.* Некоторые связи бывают настолько сильными (конечно, в случае связи «один ко многим»), что при удалении опорного экземпляра сущности (соответствующего концу связи «один») нужно удалить и все экземпляры сущности, соответствующие концу связи «многие». Соответствующее требование каскадного удаления можно специфицировать при определении связи.
- *Домены.* Как и в случае реляционной модели данных, в некоторых случаях полезна возможность определения потенциально допустимого множества значений атрибута сущности (домена).

вылета, аэропорт назначения, ЭЛЕМЕНТ РАСПИСАНИЯ с атрибутами дата-время вылета, бортовой номер самолета, тип самолета и ГОРОД. Уникальным идентификатором сущности РЕЙС является атрибут номер рейса, уникальный идентификатор ЭЛЕМЕНТ РАСПИСАНИЯ состоит из атрибута дата вылета и конца связи КОГДА, НА ЧЕМ. Мы видим, что ни в одном типе сущности больше нет атрибутов, определяемых частью уникального идентификатора. Свойства второй нормальной формы удовлетворяются, и мы имеем более качественную диаграмму.

Третья нормальная форма ER-диаграммы

В третьей нормальной форме устраняются атрибуты, зависящие от атрибутов, не входящих в уникальный идентификатор. Эти атрибуты являются основой отдельной сущности.

Взглянем еще раз на тип сущности ЭЛЕМЕНТ РАСПИСАНИЯ на рис. 9.10 (b). Конечно, каждый день каждый рейс выполняется только одним самолетом, поэтому бортовой номер самолета полностью зависит от уникального идентификатора. Но бортовой номер является уникальной характеристикой каждого самолета, и от этой характеристики зависят все остальные характеристики, в частности тип самолета. Другими словами, между уникальным идентификатором и другими атрибутами типа сущности ЭЛЕМЕНТ РАСПИСАНИЯ имеются следующие функциональные зависимости:

```
{КОГДА, НА ЧЕМ, дата-время вылета}→бортовой номер самолета
{КОГДА, НА ЧЕМ, дата-время вылета}→тип самолета
бортовой номер самолета→тип самолета
```

Как видно, имеется транзитивная FD {КОГДА, НА ЧЕМ, дата вылета} → тип самолета, и наличие этой FD вызывает нарушение требования третьей нормальной формы. На самом деле, тип сущности ЭЛЕМЕНТ РАСПИСАНИЯ на рис. 9.10 (b) включает в себя (по крайней мере, частично) тип сущности САМОЛЕТ. Это вызывает избыточность хранения и затуманивает смысл диаграммы. На рис. 9.11 показан нормализованный вариант диаграммы, в котором все сущности находятся в третьей нормальной форме.

Более сложные элементы ER-модели

До сих пор мы рассматривали только самые основные и наиболее очевидные понятия ER-модели данных. К числу некоторых более сложных элементов модели относятся следующие.

- *Подтипы и супертипы сущностей.* Подобно тому как это делается в языках программирования с развитыми типовыми системами (например, в язы-

Эти и другие усложненные элементы модели данных «Сущность-Связь» делают ее более мощной, но одновременно несколько затрудняют ее использование. Конечно, при реальном применении ER-диаграмм для проектирования баз данных необходимо ознакомиться со всеми возможностями. Ниже мы подробнее обсудим два элемента из числа упомянутых выше – супертипы и подтипы сущности, а также приведем пример сущности с взаимно исключающими связями.

Наследование типов сущности и типов связи

Сущность может быть расщеплена на два или большее число взаимно исключающих подтипов, каждый из которых включает общие атрибуты и/или связи. Эти общие атрибуты и/или связи явно определяются один раз на более высоком уровне. В подтипах могут определяться собственные атрибуты и/или связи. В принципе, подтипизация может продолжаться на более низких уровнях, но опыт использования ER-модели при проектировании баз данных показывает, что в большинстве случаев оказывается достаточно двух-трех уровней.

Если у типа сущности A имеются подтипы B_1, B_2,..., B_n, то:

(a) любой экземпляр типа сущности B_1, B_2,..., B_n является экземпляром типа сущности A (включение);

(b) если a является экземпляром типа сущности A, то a является экземпляром некоторого подтипа сущности B_i $(i = 1, 2, \ldots, n)$ (отсутствие собственных экземпляров у супертипа сущности);

(c) ни для каких подтипов B_i и B_j $(i, j = 1, 2, \ldots, n)$ не существует экземпляра, типом которого одновременно являются типы сущности B_i и B_j (разъединенность подтипов).

Тип сущности, на основе которого определяются подтипы, называется супертипом. Как мы видели выше, подтипы должны образовывать полное множество, т. е. любой экземпляр супертипа должен относиться к некоторому подтипу. Иногда для обеспечения такой полноты приходится определять дополнительный подтип ПРОЧИЕ.

Пример супертипа ЛЕТАТЕЛЬНЫЙ АППАРАТ и его подтипов АЭРОПЛАН, ВЕРТОЛЕТ, ПТИЦЕЛЕТ и ПРОЧИЕ показан на рис. 9.12. У подтипа АЭРОПЛАН имеются два собственных подтипа – ПЛАНЕР и МОТОРНЫЙ САМОЛЕТ. Для супертипа сущности ЛЕТАТЕЛЬНЫЙ АППАРАТ определен атрибут максимальная дальность полета и необязательная связь «многие ко многим» с типом сущности ПИЛОТ. Эти атрибут и связь наследуется всеми подтипами этого супертипа сущности. У непосредственного подтипа сущности АЭРОПЛАН определяется один дополнительный атрибут, так что в совокупности у данного типа сущности имеются два атрибута максимальная дальность полета и размах крыльев и одна унаследованная связь с типом сущно-

сти ПИЛОТ. У подтипа второго уровня МОТОРНЫЙ САМОЛЕТ супертипа АЭРОПЛАН определяется один дополнительный атрибут мощность мотора и одна дополнительная (обязательная) связь с типом сущности АЭРОДРОМ. Тем самым, у типа сущности МОТОРНЫЙ САМОЛЕТ имеются три атрибута: два унаследованных – максимальная дальность полета и размах крыльев и один собственный – мощность мотора, а также две связи: одна унаследованная – с типом сущности ПИЛОТ и одна собственная – с типом сущности АЭРОДРОМ. И так далее. Понятно, что для типа сущности ПРОЧИЕ, скорее всего, бессмысленно определять собственные атрибуты и связи, так что свойства этого типа будут совпадать со свойствами его супертипа.

Как же следует понимать диаграмму, представленную на рис. 9.12? Если начинать от супертипа, то диаграмма изображает ЛЕТАТЕЛЬНЫЙ АППАРАТ, который должен быть АЭРОПЛАНОМ, ВЕРТОЛЕТОМ, ПТИЦЕЛЕТОМ или ДРУГИМ ЛЕТАТЕЛЬНЫМ АППАРАТОМ. Если начинать от подтипа (например, сущности ВЕРТОЛЕТ), то это ВЕРТОЛЕТ, который относится к типу ЛЕТАТЕЛЬНОГО АППАРАТА. Если начинать от подтипа, который является одновременно супертипом, то это АЭРОПЛАН, который относится к типу ЛЕТАТЕЛЬНОГО АППАРАТА и должен быть ПЛАНЕРОМ или МОТОРНЫМ САМОЛЕТОМ.

В механизме наследования ER-модели допускается наличие двух или более разбиений сущности на подтипы. Например, тип сущности ЧЕЛОВЕК может быть расщеплен на подтипы по профессиональному признаку (ПРОГРАММИСТ, ДОЯРКА и т. д.), а может быть расщеплен и по половому признаку (МУЖЧИНА, ЖЕНЩИНА).

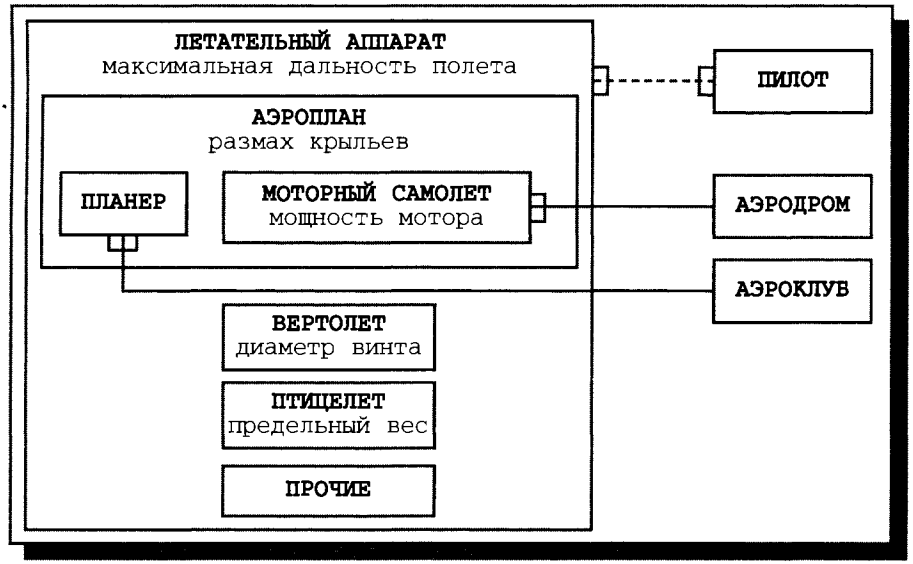


Рис. 9.12. Супертипы и подтипы сущности

Взаимно исключающие связи

Пример диаграммы из двух сущностей с взаимно исключающими связями показан на рис. 9.13(a). Самолет может находиться в рабочем состоянии, и тогда у него имеется один и только один пилот. Или же самолет может находиться на ремонте на одном из нескольких возможных авиаремонтных предприятий (каждое предприятие может производить ремонт нескольких самолетов).

В данном случае для каждого экземпляра типа сущности САМОЛЕТ должен существовать экземпляр одной из указанных связей. Для экземпляров типа сущности САМОЛЕТ, соответствующих исправным самолетам, должен существовать экземпляр связи «один к одному» с экземпляром типа сущности ПИЛОТ, а экземпляры, соответствующие неисправным самолетам, должны участвовать в экземпляре типа связи «многие к одному» с экземпляром типа сущности АВИАРЕМОНТНОЕ ПРЕДПРИЯТИЕ.

Как показано на рис. 9.13(b), диаграмма со взаимно исключающими связями из рис. 9.13(a) может быть преобразована к диаграмме без взаимно исключающих связей путем введения подтипов. Поскольку любой самолет может быть либо исправным, либо неисправным, можно корректным образом ввести два подтипа супертипа САМОЛЕТ — ИСПРАВНЫЙ САМОЛЕТ и НЕИСПРАВНЫЙ САМОЛЕТ. На уровне супертипа сущности связи не определяются. Для подтипа ИСПРАВНЫЙ САМОЛЕТ определяется обязательная связь «один к одному» с типом сущности ПИЛОТ, а для подтипа НЕИСПРАВНЫЙ СА-

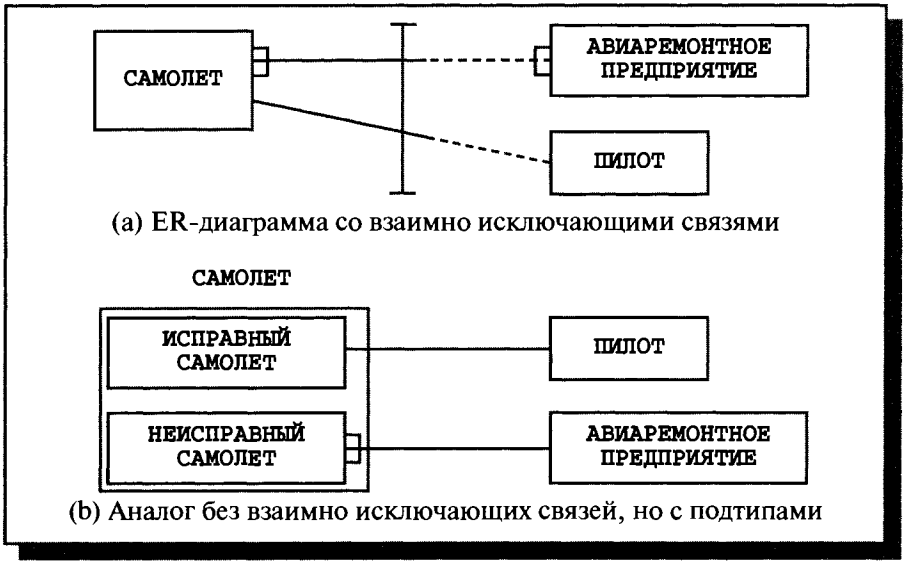


Рис. 9.13. Пример ER-диаграммы со взаимно исключающими связями

МОЛЕТ определяется обязательная связь «многие к одному» с типом сущности АВИАРЕМОНТНОЕ ПРЕДПРИЯТИЕ.

Заметим, что для того чтобы описанная схема реализации механизма взаимно исключающих связей на основе механизма наследования действительно могла работать, в средствах манипулирования данными ER-модели должна быть предусмотрена возможность динамического изменения типа сущности у экземпляра. Конкретно для нашего случая требуется возможность изменения типа экземпляра сущности ИСПРАВНЫЙ САМОЛЕТ на тип сущности НЕИСПРАВНЫЙ САМОЛЕТ, и наоборот (исправный самолет может ломаться, неисправный самолет – приводиться в рабочее состояние). Конечно, при такой смене типа должен изменяться и экземпляр связи. Заметим, что в рассматриваемом случае мы имеем дело с ограниченным динамическим изменением типа экземпляра, поскольку и исправные, и неисправные самолеты являются экземплярами супертипа САМОЛЕТ.

Получение реляционной схемы из ER-диаграммы

Опишем типовую многошаговую процедуру преобразования ER-диаграммы в реляционную (более точно, в SQL-ориентированную) схему базы данных.

Базовые приемы

Каждый *простой тип сущности* превращается в таблицу. (Простым типом сущности называется тип сущности, не являющийся подтипом и не имеющий подтипов.) *Имя сущности* становится именем таблицы. *Экземплярам* типа сущности соответствуют *строки* соответствующей таблицы.

Каждый *атрибут* становится столбцом таблицы с тем же именем; может выбираться более точный формат представления данных. Столбцы, соответствующие необязательным атрибутам, могут содержать неопределенные значения; столбцы, соответствующие обязательным атрибутам, – не могут.

Компоненты уникального идентификатора сущности превращаются в *первичный ключ таблицы*. Если имеется несколько возможных уникальных идентификаторов, для первичного ключа выбирается наиболее характерный. Если в состав уникального идентификатора входят связи, к числу столбцов первичного ключа добавляется копия уникального идентификатора сущности, находящейся на дальнем конце связи (этот процесс может продолжаться рекурсивно, и в общем случае может привести к зацикливанию). Для именования этих столбцов используются имена концов связей и/или имена парных типов сущностей.

Связи «многие к одному» (и «один к одному») становятся внешними ключами, т. е. образуется копия уникального идентификатора сущности

на конце связи «один», и соответствующие столбцы составляют внешний ключ таблицы, соответствующей типу сущности на конце связи «многие». Необязательные связи соответствуют столбцам внешнего ключа, допускающим наличие неопределенных значений; обязательные связи – столбцам, не допускающим неопределенных значений. Если между двумя типами сущности *A* и *B* имеется связь «один к одному», то соответствующий внешний ключ по желанию проектировщика может быть объявлен как в таблице *A*, так и в таблице *B*. Чтобы отразить в определении таблицы ограничение, которое заключается в том, что степень конца связи должна равняться единице, соответствующий (возможно, составной) столбец должен быть дополнительно специфицирован как возможный ключ таблицы (в случае использования языка SQL для этого служит спецификация `UNIQUE` – см. лекцию 12).

Для поддержки связи «многие ко многим» между типами сущности *A* и *B* создается дополнительная таблица *AB* с двумя столбцами, один из которых содержит уникальные идентификаторы экземпляров сущности *A*, а другой – уникальные идентификаторы экземпляров сущности *B*. Обозначим через `УИД(c)` уникальный идентификатор экземпляра *c* некоторого типа сущности *C*. Тогда, если в экземпляре связи «многие ко многим» участвуют экземпляры a_1, a_2, ..., a_n типа сущности *A* и экземпляры b_1, b_2, ..., b_m типа сущности *B*, то в таблице *AB* должны присутствовать все строки вида <`УИД(`a_i`)`, `УИД(`b_j`)`> для $i = 1, 2, \ldots, n$, $j = 1, 2, \ldots, m$. Понятно, что, используя таблицы *A*, *B* и *AB*, с помощью стандартных реляционных операций можно найти все пары экземпляров типов сущности, участвующих в данной связи.

Индексы создаются для первичного ключа (уникальный индекс), внешних ключей и тех атрибутов, на которых предполагается в основном базировать запросы.*

Представление в реляционной схеме супертипов и подтипов сущности

В этом подразделе мы предполагаем, что реляционная схема базы данных проектируется в расчете на использование обычной SQL-ориентированной СУБД, не поддерживающей объектно-реляционные расширения. Кстати, заметим, что поддержка таких расширений не слишком помогает при переходе от концептуальной схемы базы данных в модели

* Как отмечалось в начале лекции 6, вопросы определения индексов и других вспомогательных структур данных относятся к этапу физического, а не логического проектирования данных. Конечно, на практике эти этапы часто перекрываются во времени. Заметим, кстати, что в SQL-ориентированных СУБД индексы для всех возможных и внешних ключей, как правило, создаются системой автоматически.

«Сущность-Связь» к объектно-реляционной схеме, соответствующей последним стандартам языка SQL.

Если в концептуальной схеме (ER-диаграмме) присутствуют подтипы, то возможны два способа их представления в реляционной схеме:

(a) собрать все подтипы в одной таблице;

(b) для каждого подтипа образовать отдельную таблицу.

При применении способа (a) таблица создается для максимального супертипа (типа сущности, не являющегося подтипом), а для подтипов могут создаваться представления (см. лекции про SQL). Таблица содержит столбцы, соответствующие каждому атрибуту (и связям) каждого подтипа. В таблицу добавляется, по крайней мере, один столбец, содержащий код ТИПА; он становится частью первичного ключа. Для каждой строки таблицы значение этого столбца определяет тип сущности, экземпляру которого соответствует строка. Столбцы этой строки, которые соответствуют атрибутам и связям, отсутствующим в данном типе сущности, должны содержать неопределенные значения.

При использовании метода (b) для каждого подтипа первого уровня (для более глубоких уровней применяется метод (a)) супертип воссоздается с помощью представления UNION (из всех таблиц подтипов выбираются общие столбцы – столбцы супертипа).

У каждого способа есть свои достоинства и недостатки. К достоинствам первого способа (одна таблица для супертипа и всех его подтипов) можно отнести следующее:

- соответствие логике супертипов и подтипов; поскольку любой экземпляр любого подтипа является экземпляром супертипа, логично хранить вместе все строки, соответствующие экземплярам супертипа;
- обеспечение простого доступа к экземплярам супертипа и не слишком сложный доступ к экземплярам подтипов;
- возможность обойтись небольшим числом таблиц.

Недостатки метода (a):

- прикладная программа, имеющая дело с одной таблицей супертипа, должна включать дополнительную логику работы с разными наборами столбцов (в зависимости от значения столбца ТИП) и разными ограничениями целостности (в зависимости от особенностей связей, определенных для подтипа);
- общая для всех подтипов таблица потенциально может стать узким местом при многопользовательском доступе по причине возможности блокировки таблицы целиком*;
- для индивидуальных столбцов подтипов должна допускаться возможность содержать неопределенные значения; таким образом, потенци-

* Этот аспект тоже относится к этапу физического проектирования, поскольку связан с особенностями реализации конкретной СУБД.

ально в общей таблице будет содержаться много неопределенных значений, что при использовании некоторых РСУБД может потребовать значительного объема внешней памяти*.

Достоинства метода (b) состоят в следующем:

- действуют более понятные правила работы с подтипами (каждому подтипу соответствует одноименная таблица);
- упрощается логика приложений; каждая программа работает только с нужной таблицей.

Недостатки метода (b):

- в общем случае требуется слишком много отдельных таблиц;
- работа с экземплярами супертипа на основе представления, объединяющего таблицы супертипов, может оказаться недостаточно эффективной;
- поскольку множество экземпляров супертипа является объединением множеств экземпляров подтипов, не все РСУБД могут обеспечить выполнение операций модификации экземпляров супертипа.

Представление в реляционной схеме взаимно исключающих связей

Существуют два способа формирования схемы реляционной БД при наличии взаимно исключающих связей (имеются в виду связи «один ко многим», причем конец связи «многие» находится на стороне сущности, для которой связи являются взаимно исключающими):

(a) общее хранение внешних ключей;

(b) раздельное хранение внешних ключей.

Понятно, что если имеются взаимно исключающие связи упомянутой категории, то в таблице, соответствующей сущности, для которой связи являются взаимно исключающими, необходимо хранить внешние ключи. Если внешние ключи всех потенциально связанных таблиц имеют общий формат, то можно применить способ (a), т. е. создать два столбца: идентификатор связи и идентификатор сущности (возможно, составной). Столбец идентификатора связи используется для различения связей, покрываемых дугой исключения. В столбце (столбцах) идентификатора сущности хранятся значения уникального идентификатора сущности на дальнем конце соответствующей связи.

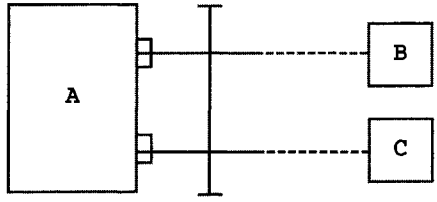
Если результирующие внешние ключи не относятся к одному домену, то приходится прибегать к использованию способа (b), т. е. создавать для каждой связи, покрываемой дугой исключения, явные столбцы внешних ключей; все эти столбцы могут содержать неопределенные значения.

* Хотя в большинстве SQL-ориентированных СУБД хранение неопределенных значений вызывает минимальные накладные расходы; это снова аспект физического проектирования.

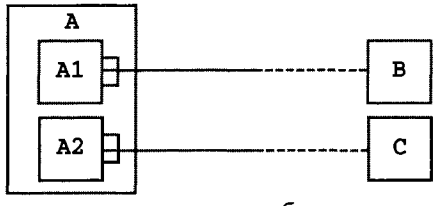
Преимущество подхода (a) состоит в том, что в таблице, соответствующей сущности, появляется всего два дополнительных столбца. Очевидным недостатком является усложнение выполнения операции соединения: чтобы воспользоваться для соединения одной из альтернативных связей, нужно сначала произвести ограничение таблицы в соответствии с нужным значением столбца, содержащего идентификаторы связей.

При использовании подхода (b) соединения являются явными (и естественными). Недостаток состоит в том, что требуется иметь столько столбцов, сколько имеется альтернативных связей. Кроме того, в каждом из таких столбцов будет содержаться много неопределенных значений, хранение которых потенциально может привести к серьезным накладным расходам внешней памяти.

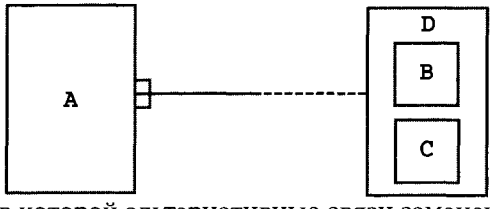
Модификация, показанная на рис. 9.14 (b), основана на том наблюдении, что коль скоро связи являются альтернативными, то они разделяют



(a) ER-диаграмма с взаимно исключающими связями



(b) ER-диаграмма с подтипами сущности без альтернативных связей



(c) ER-диаграмма, в которой альтернативные связи заменены связью с супертипом сущности

Рис. 9.14. Возможные модификации ER-диаграмм, позволяющие избежать взаимно исключающих связей

множество экземпляров сущности *A* на два или более непересекающихся подмножества, которые могут лежать в основе определения подтипов *A1* и *A2*. Это хороший вариант, если такие подтипы могут пригодиться еще для чего-нибудь. Например, в случае взаимно исключающей связи, представленной на рис. 9.12, у исправных и неисправных самолетов могут иметься несовпадающие множества атрибутов (скажем, у типа сущности ИСПРАВНЫЕ САМОЛЕТЫ может иметься атрибут дата завершения гарантийного срока, а у типа сущности НЕИСПРАВНЫЕ САМОЛЕТЫ – атрибут тип неисправности). С другой стороны, как отмечалось в предыдущем разделе, для использования этого подхода требуется возможность динамического изменения типа существующего экземпляра.

Модификация, показанная на рис. 9.14 (c), основана на том наблюдении, что коль скоро типы сущности *B* и *C* участвуют в альтернативной связи, то, по всей видимости, у этих сущностей имеется что-то общее. Возможно, их было бы правильнее определять как подтипы некоторого общего типа сущности. Заметим, что пример с рис. 9.12 явно демонстрирует, что далеко не всегда типы сущности, участвующие в альтернативной связи, обладают общими чертами. Создание общего супертипа для типов сущности ПИЛОТ и АВИАРЕМОНТНОЕ ПРЕДПРИЯТИЕ представляется весьма странной идеей.

На этом мы заканчиваем краткую экскурсию в семантическое моделирование с использованием ER-диаграмм.

Заключение

Основной целью данной лекции было ознакомление с семантическими моделями данных на примере упрощенного варианта ER-модели. Представленный вариант ER-модели, с одной стороны, является достаточно развитым, чтобы можно было почувствовать общую специфику семантических моделей данных, а с другой стороны, не перегружен деталями и излишними понятиями, затрудняющими общее понимание подхода.

С практической точки зрения наибольшую пользу могут принести рассмотренные приемы перехода от ER-диаграмм к схеме реляционной базы данных. Особенно могут пригодиться рекомендации по представлению в реляционной схеме связей «многие ко многим», подтипов и супертипов сущности и взаимно исключающих связей.

Но, помимо прочего, язык UML активно применяется для проектирования реляционных БД. Для этого используется небольшая часть языка (диаграммы классов), да и то не в полном объеме. С точки зрения проектирования реляционных БД модельные возможности не слишком отличаются от возможностей ER-диаграмм. Но все же мы кратко опишем диаграммы классов UML, поскольку их использование при проектировании реляционных БД позволяет оставаться в общем контексте UML и применять другие виды диаграмм для проектирования приложений баз данных.

Основные понятия диаграмм классов UML

Диаграммой классов в терминологии UML называется диаграмма, на которой показан набор классов (и некоторых других сущностей*, не имеющих явного отношения к проектированию БД), а также связей между этими классами.** Кроме того, диаграмма классов может включать комментарии и ограничения. Ограничения могут неформально задаваться на естественном языке или же могут формулироваться на языке объектных ограничений OCL (Object Constraints Language).*** Чуть позже мы обсудим эту тему более подробно.

Классы, атрибуты, операции

Классом называется именованное описание совокупности объектов с общими атрибутами, операциями, связями и семантикой. Графически класс изображается в виде прямоугольника. У каждого класса должно быть имя (текстовая строка), уникально отличающее его от всех других классов. При формировании имен классов в UML допускается использование произвольной комбинации букв, цифр и даже знаков препинания. Однако на практике рекомендуется использовать в качестве имен классов короткие и осмысленные прилагательные и существительные, каждое из которых начинается с заглавной буквы. Примеры описания классов показаны на рис. 10.1.

---

* В этой лекции мы используем термин *сущность* настолько же неформально, как в предыдущей лекции использовали термин *объект*. UML претендует на обеспечение более точного и формального понятия *объекта* (UML обычно называют языком *объектно-ориентированного моделирования*). В спецификации языка UML даже присутствует определение понятия объекта средствами самого UML. Однако, по нашему глубокому убеждению, несмотря на эти попытки, понятие *объекта* в UML остается таким же нечетким, как и понятие *сущности* в ER-модели. По-прежнему приходится опираться в основном на интуицию и здравый смысл.

** В UML, как и в модели ER-диаграмм, для родового обозначения связей используется термин *relationship*. Во многих переводах книг про UML на русский язык вместо термина *связь* применяется термин *отношение*. Как и в предыдущей лекции, мы используем термин *связь*.

*** Язык OCL является частью общей спецификации UML, но, в отличие от других частей языка, имеет не графическую, а линейную нотацию.

Лекция 10. Проектирование реляционных баз данных с использованием семантических моделей: диаграммы классов языка UML

В этой лекции мы обсудим основные понятия диаграмм классов языка UML и возможности применения этой диаграммной модели для проектирования реляционных баз данных. Кроме того, в лекции будет кратко рассмотрен язык объектных ограничений OCL и приведены примеры формулировок на языке OCL ограничений целостности в терминах концептуальной схемы базы данных.

Ключевые слова: язык объектно-ориентированного моделирования UML (Unified Modeling Language), диаграмма классов, язык объектных ограничений OCL (Object Constraints Language), класс, атрибут класса, свойство класса, операция класса, сигнатура операции, связь в диаграмме классов, связь-зависимость (dependency), связь-обобщение (generalization), связь-ассоциация (association), суперкласс, подкласс, связь «is a», полиморфизм по включению, множественное наследование классов в UML, *n*-арные ассоциации в UML, имя ассоциации, роль класса в ассоциации, кратность (multiplicity) роли ассоциации, экземпляр ассоциации (соединение – link), агрегатная ассоциация, композитная ассоциация (композиция), навигация по ассоциации, уникальный идентификатор объекта, типы коллекций в OCL, пред- и постусловия операций классов в OCL, инварианты классов в OCL, операции над объектами в OCL, операции над значениями коллекционных типов данных в OCL.

Введение

Языку объектно-ориентированного моделирования UML (Unified Modeling Language) посвящено великое множество книг, многие из которых переведены на русский язык (а некоторые и написаны российскими авторами). Язык UML разработан и развивается консорциумом OMG (Object Management Group) и имеет много общего с объектными моделями, на которых основана технология распределенных объектных систем CORBA, и объектной моделью ODMG (Object Data Management Group).

UML позволяет моделировать разные виды систем: чисто программные, чисто аппаратные, программно-аппаратные, смешанные, явно включающие деятельность людей и т. д. Даже если бы мы ограничились возможностями использования UML для проектирования программных информационных систем, это слишком далеко увело бы нас от основной темы курса.

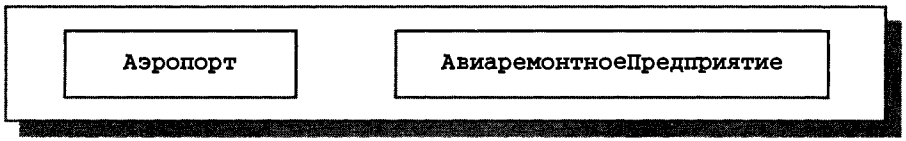


Рис. 10.1. Примеры описания классов

Атрибутом класса называется именованное *свойство* класса, описывающее множество значений, которые могут принимать экземпляры этого свойства. Класс может иметь любое число атрибутов (в частности, не иметь ни одного атрибута). Свойство, выражаемое атрибутом, является свойством моделируемой сущности, общим для всех объектов данного класса. Так что атрибут является абстракцией состояния объекта. Любой атрибут любого объекта класса должен иметь некоторое значение.

Имена атрибутов представляются в разделе класса, расположенном под именем класса. Хотя UML не накладывает ограничений на способы создания имен атрибутов (имя атрибута может быть произвольной текстовой строкой), на практике рекомендуется использовать короткие прилагательные и существительные, отражающие смысл соответствующего свойства класса. Первое слово в имени атрибута рекомендуется писать с прописной буквы, а все остальные слова – с заглавной. Пример описания класса с указанными атрибутами показан на рис. 10.2.

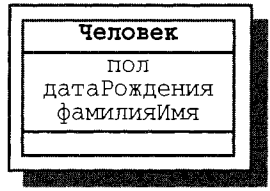


Рис. 10.2. Класс `Человек` с указанными именами атрибутов

Операцией класса называется именованная услуга, которую можно запросить у любого объекта этого класса. Операция – это абстракция того, что можно делать с объектом. Класс может содержать любое число операций (в частности, не содержать ни одной операции). Набор операций класса является общим для всех объектов данного класса.

Операции класса определяются в разделе, расположенном ниже раздела с атрибутами. При этом можно ограничиться только указанием имен операций, оставив детальную спецификацию выполнения операций на более поздние этапы моделирования. Для именования операций рекомендуется использовать глагольные формы, соответствующие ожидаемому поведению объектов данного класса. Описание операции может также содержать ее *сигнатуру*, т. е. имена и типы всех параметров, а если опера-

ция является функцией, то и тип ее значения. Класс Человек с определенными операциями показан на рис. 10.3.

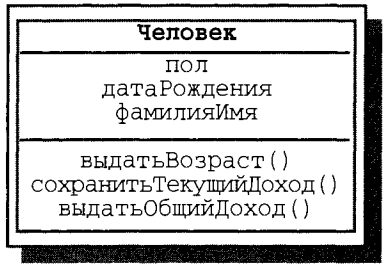


Рис. 10.3. Класс Человек с операциями

Для класса Человек мы определили три операции: выдатьВозраст, сохранитьТекущийДоход, выдатьОбщийДоход. В операции выдатьВозраст используются значение атрибута датаРождения и значение текущей даты. Операция сохранитьТекущийДоход позволяет зафиксировать в состоянии объекта сумму и дату поступления дохода данного человека. Операция выдатьОбщийДоход выдает суммарный доход данного человека за указанное время. Заметим, что состояние объекта меняется при выполнении только второй операции. Результаты первой и третьей операций формируются на основе текущего состояния объекта.

Категории связей. Связь-зависимость

В диаграмме классов могут участвовать связи трех разных категорий: *зависимость* (dependency), *обобщение* (generalization) и *ассоциация* (association). При проектировании реляционных БД наиболее важны вторая и третья категории связей, поэтому о связях-зависимостях будет сказано только самое основное.

Зависимостью называют связь по применению, когда изменение в спецификации одного класса может повлиять на поведение другого класса, использующего первый класс. Чаще всего зависимости применяют в диаграммах классов, чтобы отразить в сигнатуре операции одного класса тот факт, что параметром этой операции могут быть объекты другого класса. Понятно, что если интерфейс второго класса изменяется, это влияет на поведение объектов первого класса. Простой пример диаграммы классов со связью-зависимостью показан на рис. 10.4.

Зависимость показывается прерывистой линией со стрелкой, направленной к классу, от которого имеется зависимость. Очевидно, что связи-зависимости существенны для объектно-ориентированных систем (в том числе и для ООБД). При проектировании реляционных БД непо-

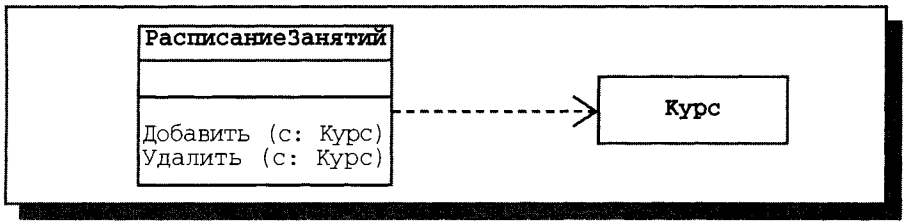


Рис. 10.4. Диаграмма классов со связью-зависимостью

нятно, что делать с зависимостями (как воспользоваться этой информацией в реляционной БД?).

Связи-обобщения и механизм наследования классов в UML

Связью-обобщением называется связь между общей сущностью, называемой *суперклассом*, или *родителем*, и более специализированной разновидностью этой сущности, называемой *подклассом*, или *потомком*. Обобщения иногда называют связями *«is a»*, имея в виду, что класс-потомок является частным случаем класса-предка. Класс-потомок наследует все атрибуты и операции класса-предка, но в нем могут быть определены дополнительные атрибуты и операции.

Объекты класса-потомка могут использоваться везде, где могут использоваться объекты класса-предка. Это свойство называют полиморфизмом по включению, имея в виду, что объекты потомка можно считать включаемыми во множество объектов класса-предка. Графически обобщения изображаются в виде сплошной линии с большой незакрашенной стрелкой, направленной к суперклассу. В качестве первой иллюстрации, приведенной на рис. 10.5, воспользуемся классификацией летательных аппаратов с рис. 9.12 из предыдущей лекции. На рис. 10.5 показан пример иерархии одиночного наследования: у каждого подкласса имеется только один суперкласс.

Одиночное наследование является достаточным в большинстве случаев применения связи-обобщения. Однако в UML допускается и множественное наследование, когда один подкласс определяется на основе нескольких суперклассов. В качестве одного из разумных (не слишком распространенных) примеров рассмотрим диаграмму классов на рис. 10.6 (для упрощения диаграммы имена атрибутов и операций указывать не будем).

На этой диаграмме классы `Студент` и `Преподаватель` порождены из одного суперкласса `ЧеловекИзУниверситета`. Вообще говоря, к классу `Студент` относятся те объекты класса `ЧеловекИзУниверситета`, которые соответствуют студентам, а к классу `Преподаватель` — объекты класса `ЧеловекИзУниверситета`, соответствующие преподавателям. Но, как это часто случается, многие студенты уже в студенческие годы начина-

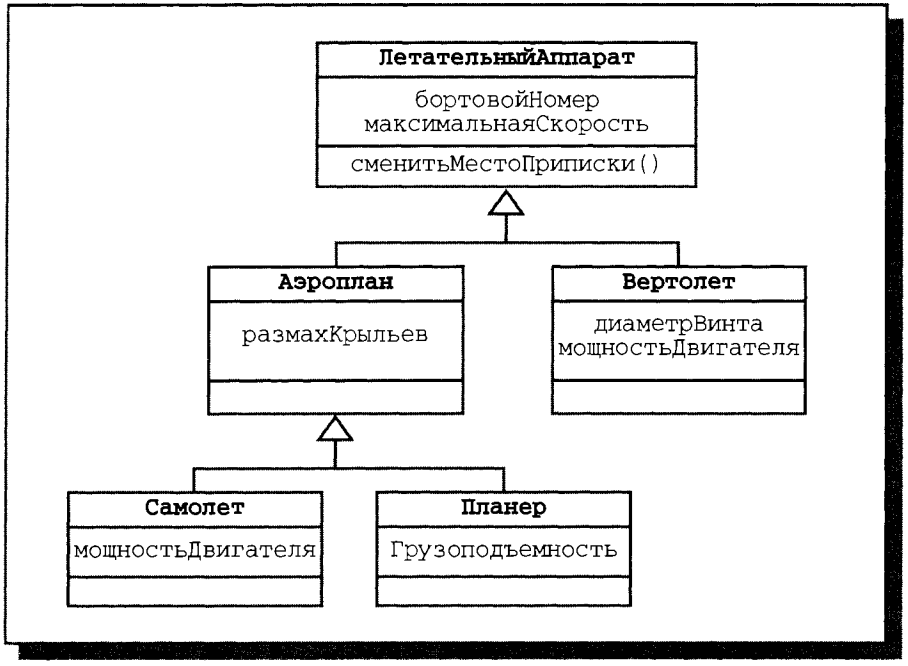


Рис. 10.5. Иерархия одиночного наследования классов

ют преподавать, так что могут существовать такие два объекта классов `Студент` и `Преподаватель`, которым соответствует один объект класса `ЧеловекИзУниверситета`. Итак, среди объектов класса `Студент` могут быть преподаватели, а некоторые преподаватели могут быть студентами. Тогда мы можем определить класс `СтудентПреподаватель` путем множественного наследования от суперклассов `Студент` и `Преподаватель`. Объект класса `СтудентПреподаватель` обладает всеми свойствами и операциями классов `Студент` и `Преподаватель` и может быть использован везде, где могут применяться объекты этих классов. Так что полиморфизм по включению продолжает работать.Следует отметить, что множественное наследование, помимо того что не слишком часто требуется на практике, порождает ряд проблем, из которых одной из наиболее известных является проблема именования атрибутов и операций в подклассе, полученном путем множественного наследования. Например, предположим, что при образовании подклассов `Студент` и `Преподаватель` в них обоих был определен атрибут с именем `номерКомнаты`. Очень вероятно, что для объектов класса `Студент` значениями этого атрибута будут номера комнат в студенческом общежитии, а для объектов класса `Преподаватель` – номера служебных кабинетов. Как быть с объектами класса `СтудентПреподаватель`, для

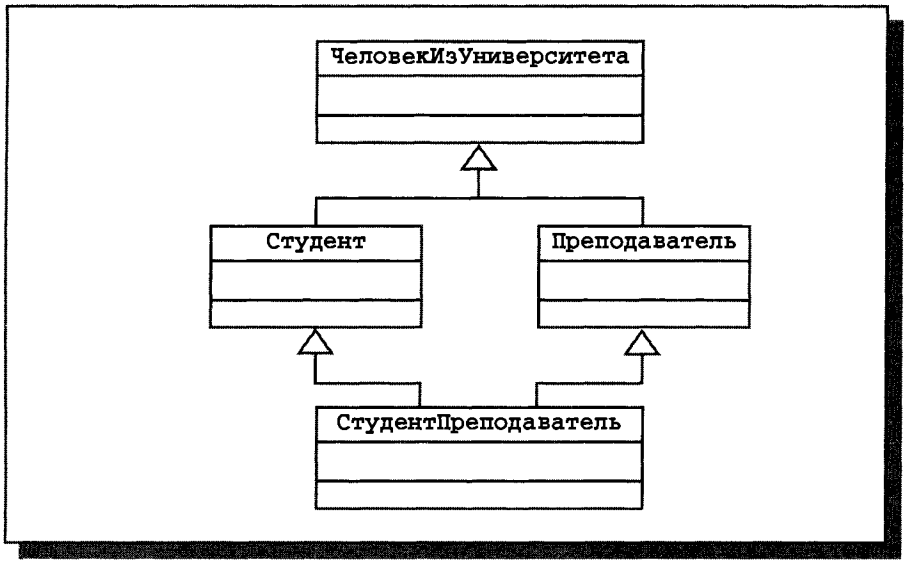


Рис. 10.6. Пример множественного наследования классов

которых существенны оба одноименных атрибута (у студента-преподавателя могут иметься и комната в общежитии, и служебный кабинет)? На практике применяется одно из следующих решений:

(1) запретить образование подкласса `СтудентПреподаватель`, пока в одном из суперклассов не будет произведено переименование атрибута `номерКомнаты`;

(2) наследовать это свойство только от одного из суперклассов, так что, например, значением атрибута `номерКомнаты` у объектов класса `СтудентПреподаватель` всегда будут номера служебных кабинетов;

(3) унаследовать в подклассе оба свойства, но автоматически переименовать оба атрибута, чтобы прояснить их смысл; назвать их, например, `номерКомнатыСтудента` и `номерКомнатыПреподавателя`.

Ни одно из решений не является полностью удовлетворительным. Первое решение требует возврата к ранее определенному классу, имена атрибутов и операций которого, возможно, уже используются в приложениях. Второе решение нарушает логику наследования, не давая возможности на уровне подкласса использовать все свойства суперклассов. Наконец, третье решение заставляет использовать длинные имена атрибутов и операций, которые могут стать недопустимо длинными, если процесс множественного наследования будет продолжаться от полученного подкласса.*

* Как кажется, здесь можно провести некоторую аналогию с ситуацией, по причине наличия которой в реляционной алгебре (см. лекции 3 и 4) была введена операция RENAME.

Но, конечно, сложность проблемы именования атрибутов и операций несопоставимо меньше сложности реализации множественного наследования в реляционных БД. Поэтому при использовании UML для проектирования реляционных БД нужно очень осторожно использовать наследование классов вообще и стараться избегать множественного наследования.*

Связи-ассоциации: роли, кратность, агрегация

Ассоциацией называется структурная связь, показывающая, что объекты одного класса некоторым образом связаны с объектами другого или того же самого класса. Допускается, чтобы оба конца ассоциации относились к одному классу. В ассоциации могут связываться два класса, и тогда она называется *бинарной*. Допускается создание ассоциаций, связывающих сразу *n* классов (они называются *n-арными* ассоциациями).** Графически ассоциация изображается в виде линии, соединяющей класс сам с собой или с другими классами.

С понятием ассоциации связаны четыре важных дополнительных понятия: *имя*, *роль*, *кратность* и *агрегация*. Во-первых, ассоциации может быть присвоено *имя*, характеризующее природу связи. Смысл имени уточняется с помощью черного треугольника, который располагается над линией связи справа или слева от имени ассоциации. Этот треугольник указывает направление чтения имя связи. Пример именованной ассоциации показан на рис. 10.7. Треугольник показывает, что именованная ассоциация должна читаться как «Студент учится в Университете».

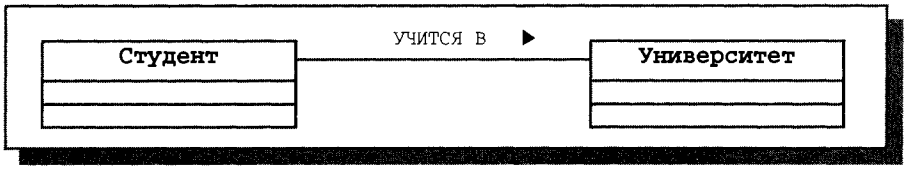


Рис. 10.7. Пример именованной ассоциации

Другим способом именования ассоциации является указание *роли* каждого класса, участвующего в этой ассоциации. Роль класса, как и имя конца связи в ER-модели, задается именем, помещаемым под линией ассоциации ближе к данному классу. На рис. 10.8 показаны две ассоциации между классами Человек и Университет, в которых эти классы играют разные роли. Как мы видим, объекты класса Человек могут выступать в

* Если под «реляционными» базами данных понимать SQL-ориентированные БД.

** Напомним, что в варианте ER-модели, рассмотренном нами в предыдущей лекции, допускались только бинарные связи. В свое время компания Oracle обосновывала это решение тем, что наличие бинарных ассоциаций всегда является достаточным. Здесь мы также ограничимся обсуждением бинарных ассоциаций.

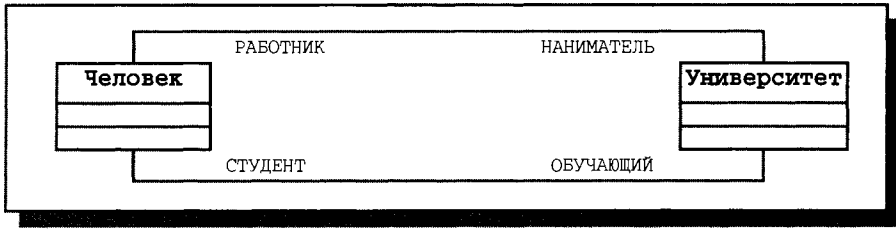


Рис. 10.8. Две ассоциации с разными ролями классов

роли РАБОТНИКОВ при участии в ассоциации, в которой объекты класса Университет играют роль НАНИМАТЕЛЯ. В другой ассоциации объекты класса Человек играют роль СТУДЕНТА, а объекты класса УНИВЕРСИТЕТ – роль ОБУЧАЮЩЕГО.

В общем случае, для ассоциации могут задаваться и ее собственное имя, и имена ролей классов. Это связано с тем, что класс может играть одну и ту же роль в разных ассоциациях, так что в общем случае пара имен ролей классов не идентифицирует ассоциацию. С другой стороны, в простых случаях, когда между двумя классами определяется только одна ассоциация, можно вообще не связывать с ней дополнительные имена.

Кратностью (multiplicity) роли ассоциации называется характеристика, указывающая, сколько объектов класса с данной ролью может или должно участвовать в каждом экземпляре ассоциации (в UML экземпляр ассоциации называется *соединением* – link, но мы не будем здесь использовать этот термин, чтобы не создавать путаницу – все-таки трудно одновременно говорить про *связи*, *ассоциации* и *соединения*, имея в виду разные понятия). Наиболее распространенным способом задания кратности роли ассоциации является указание конкретного числа или диапазона. Например, указание «1» говорит о том, что каждый объект класса с данной ролью должен участвовать в некотором экземпляре данной ассоциации, причем в каждом экземпляре ассоциации может участвовать ровно один объект класса с данной ролью. Указание диапазона «0..1» говорит о том, что не все объекты класса с данной ролью обязаны участвовать в какомлибо экземпляре данной ассоциации, но в каждом экземпляре ассоциации может участвовать только один объект. Аналогично, указание диапазона «1..*» говорит о том, что все объекты класса с данной ролью должны участвовать в некотором экземпляре данной ассоциации, и в каждом экземпляре ассоциации должен участвовать хотя бы один объект (верхняя граница не задана). Толкование диапазона «0..*» является очевидным расширением случая «0..1».

В более сложных (но крайне редко встречающихся на практике) случаях определения кратности можно использовать списки диапазонов. Например, список «2, 4..6, 8..*» говорит о том, что все объекты класса с

указанной ролью должны участвовать в некотором экземпляре данной ассоциации, и в каждом экземпляре ассоциации должны участвовать два, от четырех до шести или более семи объектов класса с данной ролью.

На диаграмме классов с рис. 10.9 показано, что произвольное (может быть, нулевое) число людей являются служащими произвольного числа университетов. Каждый университет обучает произвольное (может быть, нулевое) число студентов, но каждый студент может быть студентом только одного университета.

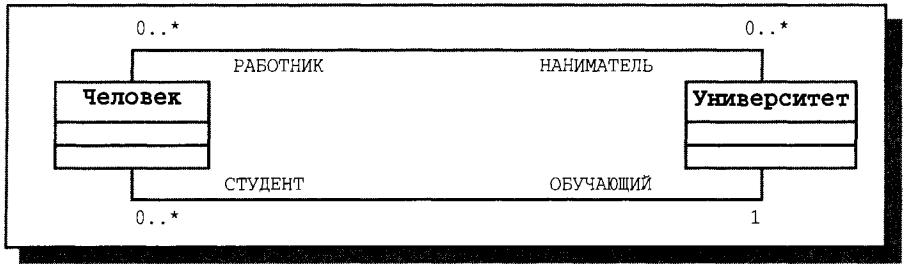


Рис. 10.9. Ассоциации с указанными кратностями ролей

Обычная ассоциация между двумя классами характеризует связь между равноправными сущностями: оба класса находятся на одном концептуальном уровне. Но иногда в диаграмме классов требуется отразить тот факт, что ассоциация между двумя классами имеет специальный вид «часть-целое». В этом случае класс «целое» имеет более высокий концептуальный уровень, чем класс «часть». Ассоциация такого рода называется *агрегатной*. Графически агрегатные ассоциации изображаются в виде простой ассоциации с незакрашенным ромбом на стороне класса-«целого». Простой пример агрегатной ассоциации показан на рис. 10.10.

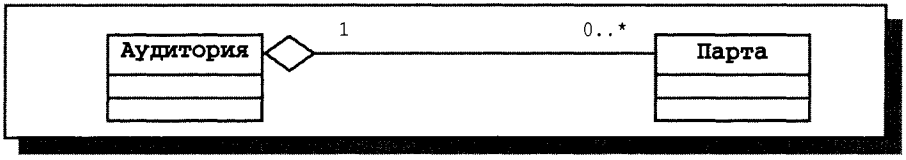


Рис. 10.10. Пример агрегатной ассоциации

Объектами класса `Аудитория` являются студенческие аудитории, в которых проходят занятия. В каждой аудитории должны быть установлены парты. Поэтому в некотором смысле класс `Парта` является «частью» класса `Аудитория`. Мы умышленно сделали роль класса `Парта` необязательной, поскольку могут существовать аудитории без парт (например, класс для занятий танцами) и некоторые парты могут находиться на складе. Обратите внимание, что, хотя аудитории, не оснащенные партами, как правило, непригодны для занятий, объекты классов `Аудитория` и `Парта` суще-

ствуют независимо. Если некоторая аудитория ликвидируется, то находящиеся в ней парты не уничтожаются, а переносятся на склад.

Бывают случаи, когда связь «части» и «целого» настолько сильна, что уничтожение «целого» приводит к уничтожению всех его «частей». Агрегатные ассоциации, обладающие таким свойством, называются *композитными*, или просто *композициями*. При наличии композиции объект-часть может быть частью только одного объекта-целого (композита). При обычной агрегатной ассоциации «часть» может одновременно принадлежать нескольким «целым». Графически композиция изображается в виде простой ассоциации, дополненной закрашенным ромбом со стороны «целого». Пример композитной агрегатной ассоциации показан на рис. 10.11.

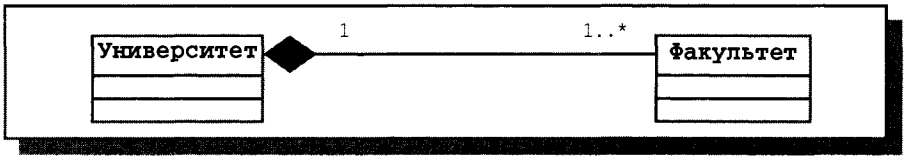


Рис. 10.11. Пример композитной агрегатной ассоциации

Любой факультет является частью одного университета, и ликвидация университета приводит к ликвидации всех существующих в нем факультетов (хотя во время существования университета отдельные факультеты могут ликвидироваться и создаваться).

Заметим, что в контексте проектирования реляционных БД агрегатные и в особенности композитные ассоциации влияют только на способ поддержки ссылочной целостности. В частности, композитная связь является явным указанием на то, что ссылочная целостность между «целым» и «частями» должна поддерживаться путем каскадного удаления частей при удалении целого. Подробнее способы поддержки ссылочной целостности в SQL-ориентированных БД рассматриваются в следующих лекциях.

При наличии простой ассоциации между двумя классами (например, ассоциации между классами `Студент` и `Университет` с рис. 10.7) предполагается возможность *навигации* между объектами, входящими в один экземпляр ассоциации. Если известен конкретный объект-студент, то должна обеспечиваться возможность узнать соответствующий объект-университет. Если известен конкретный объект-университет, то должна обеспечиваться возможность узнать все соответствующие объекты-студенты. Другими словами, если не оговорено иное, то навигация по ассоциации может проводиться в обоих направлениях.* Однако бывают слу-

* Поскольку UML – это высокоуровневый язык моделирования, в нем не уточняется, что такое навигация в реализационном смысле. Но очевидно, что само появление понятия навигации связано с объектно-ориентированной природой UML. Термин «навигация» является почти ругательным в мире реляционных БД, но для мира объектно-ориентированных БД он вполне естественен, поскольку в этом мире на модельном уровне присутствует понятие ссылки, или указателя.

чаи, когда желательно ограничить направление навигации для некоторых ассоциаций. В этом случае на линии ассоциации ставится стрелка, указывающая направление навигации. Пример показан на рис. 10.12.

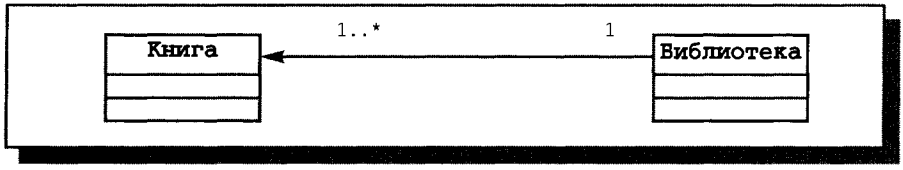


Рис. 10.12. Ассоциация с указанным направлением навигации

В библиотеке должно содержаться некоторое количество книг, и каждая книга должна принадлежать некоторой библиотеке. С точки зрения библиотечного хозяйства разумно иметь возможность найти книгу в библиотеке, т. е. произвести навигацию от объекта-библиотеки к связанным с ним объектам-книгам. Однако вряд ли потребуется по данному экземпляру книги узнать, в какой библиотеке она находится.*

Ограничения целостности и язык OCL

Как уже отмечалось, в диаграммах классов могут указываться ограничения целостности, которые должны поддерживаться в проектируемой БД. В UML допускаются два способа определения ограничений: на естественном языке и на языке OCL. На рис. 10.13 показана простая диаграмма классов Студент и Университет с ограничением, выраженным на естественном языке.

В данном случае накладывается ограничение на состояние объектов классов Студент и Университет, входящих в один экземпляр ассоциации. Объект класса Студент может входить в экземпляр связи с объектом клас-

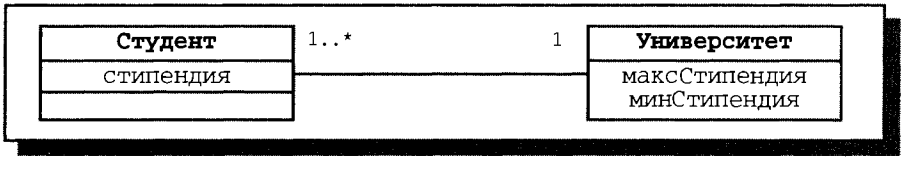


Рис. 10.13. Ограничение, выраженное на естественном языке

* С точки зрения реляционных БД ассоциации с однонаправленной навигацией можно считать указанием на необходимость ограничения видимости объектов БД. Соответствующую (но существенно более общую) возможность в SQL-ориентированных БД обеспечивает механизм представлений (view). Подробнее об этом см. в следующих лекциях.

са Университет только при условии, что размер стипендии данного студента находится в диапазоне, допустимом в данном университете.

Общая характеристика языка OCL

Более точный и лаконичный способ формулировки ограничений обеспечивает язык OCL (Object Constraints Language). Вот общая характеристика этого языка.

Из языка UML* в OCL заимствованы, в первую очередь, следующие концепции:

- класс, атрибут, операция;
- объект (экземпляр класса);
- ассоциация;
- тип данных (включая набор предопределенных типов Boolean, Integer, Real и String);
- значение (экземпляр типа данных).

Для понимания языка OCL существенны определяемые в UML традиционные для объектных моделей данных различия между объектом некоторого класса и значением некоторого типа:

- объект обладает уникальным идентификатором и может сравниваться с другими объектами только по значению идентификатора; следствием этого является возможность определения операций над множествами объектов в терминах их идентификаторов**;
- объект может быть ассоциирован через бинарную связь*** с другими объектами, что позволяет определить в OCL операцию перехода от данного объекта к связанным с ним объектам;
- в то же время значение является «чистым значением» в том смысле, что:
  - при сравнении двух значений проверяются сами эти значения;
  - кроме того, значения не могут участвовать в связях, поскольку понятие связи определено только для объектов классов.

В дополнение к *скалярным* типам данных, заимствованным из UML, в OCL предопределены *структурные* типы, которые являются разновидностями типов коллекций (*collection*):

- *математическое множество* (set), неупорядоченная коллекция, не содержащая одинаковых элементов;

* Хотя язык OCL формально считается частью UML, он специфицирован в отдельном документе, в котором присутствуют ссылки на другие части спецификации UML, а также вводятся собственные понятия и определения.

** Следует заметить, что ни в спецификации UML, ни в описании какой-либо другой объектной модели никогда прямо не говорится, что в операциях над множествами объектов в действительности участвуют идентификаторы объектов. Но другого понимания не существует.

*** Обратите внимание, что хотя в UML допускаются *n*-арные связи, в OCL речь идет только об уже привычном для нас бинарном варианте.

- *мультимножество* (bag), неупорядоченная коллекция, которая может содержать повторяющиеся элементы-дубликаты;
- *последовательность* (sequence), упорядоченная коллекция, которая может содержать элементы-дубликаты.*

В OCL элементами каждого из трех типов коллекций могут быть либо объекты, либо значения.

Язык OCL предназначен, главным образом, для определения ограничений целостности данных, соответствующих модели, которая представлена в терминах диаграммы классов UML. OCL может применяться для определения ограничений, описывающих пред- и постусловия операций классов, и ограничений, представляющих собой инварианты классов. При проектировании реляционных баз данных возможность определения пред- и постусловий операций вряд ли может оказаться существенной**. С точки зрения определения ограничений целостности баз данных более важны средства определения инвариантов классов.

Инвариант класса

Под инвариантом класса в OCL понимается условие, которому должны удовлетворять все объекты данного класса. Если говорить более точно, инвариант класса – это логическое выражение, вычисление которого должно давать *true* при создании любого объекта данного класса и сохранять истинное значение в течение всего времени существования этого объекта. При определении инварианта требуется указать имя класса и выражение, определяющее инвариант указанного класса. Синтаксически это выглядит следующим образом:

```
context <class_name> inv:
<OCL-выражение>
```

Здесь `<class-name>` является именем класса, для которого определяется инвариант, `inv` – ключевое слово, говорящее о том, что определяется именно инвариант, а не ограничение другого вида, и `context` – ключевое слово, которое говорит о том, что контекстом следующего после двоеточия OCL-выражения являются объекты класса `<class-name>`, т. е. OCL-выражение должно принимать значение *true* для всех объектов этого класса.

Заметим, что OCL является типизированным языком, поэтому у каждого выражения имеется некоторый тип. Естественно, что OCL-выражение в инварианте класса должно быть логического типа.

* В контексте проектирования реляционных БД (если не иметь в виду использование объектно-реляционных СУБД) последняя разновидность типа коллекции является бессмысленной, поскольку в реляционных БД упорядоченность не поддерживается. Поэтому мы не будем обсуждать детали операций над последовательностями.

** Если снова не иметь в виду использование объектно-реляционных СУБД.

В общем случае OCL-выражение в определении инварианта основывается на композиции операций, которым посвящена большая часть определения языка. В спецификации языка эти операции условно разделены на следующие группы:

- операции над значениями предопределенных в UML (скалярных) типов данных;
- операции над объектами;
- операции над множествами;
- операции над мультимножествами;
- операции над последовательностями.

Последовательно обсудим эти группы операций.

Операции над значениями предопределенных типов данных

Полагая очевидной семантику предопределенных скалярных типов данных и операций над ними, ограничимся лишь их перечислением. В OCL поддерживаются следующие заимствованные из определения UML скалярные типы данных: Boolean, Integer, Real и String.

Операции над объектами

В OCL определены три операции над объектами:

- получение значения атрибута;
- переход по соединению,
- вызов операции класса (последняя операция для целей проектирования реляционных БД несущественна).

Для записи этих трех операций используется «точечная нотация». Например, результатом выражения вида

```
<объект>.<имя атрибута>
```

является текущее значение атрибута с именем `имя атрибута`, если объект имеет такой атрибут. В противном случае использование подобного выражения приводит к возникновению ошибки типа.

Результатом применения к объекту операции перехода по соединению (экземпляру связи-ассоциации) является коллекция, содержащая все объекты, которые ассоциированы с данным объектом через указываемое соединение. Это соединение идентифицируется именем роли, противоположной по отношению к данному объекту. Таким образом, синтаксис выражения перехода по соединению следующий:

```
<объект>.<имя роли, противоположенной по отношению к объекту>
```

Операции над множествами, мультимножествами и последовательностями

В OCL поддерживается обширный набор операций над значениями коллекционных типов данных. Обсудим только те из них, которые являются уместными в контексте данной лекции. Синтаксически операции над коллекциями записываются в нотации, аналогичной точечной, но вместо точки используется стрелка (→). Таким образом, общий синтаксис применения операции к коллекции следующий:

```
<коллекция> → <имя операции> (<список фактических параметров>)
```

Операция `select`

В OCL определены три одноименных операции `select`, которые обрабатывают заданное множество, мультимножество или последовательность на основе заданного логического выражения над элементами коллекции. Результатом каждой операции является новое множество, мультимножество или последовательность соответственно из тех элементов входной коллекции, для которых результатом вычисления логического выражения является *true*.

Операция `collect`

Аналогично набору операций `select`, в OCL определены три операции `collect`, параметрами которых являются множество, мультимножество или последовательность и некоторое выражение над элементами соответствующей коллекции. Результатом является мультимножество для операций `collect`, определенных над множествами и мультимножествами, и последовательность для операции `collect`, определенной над последовательностью. При этом результирующая коллекция соответствующего типа (коллекция значений или объектов) состоит из результатов применения выражения к каждому элементу входной коллекции. Операция `collect` используется, главным образом, в тех случаях, когда от заданной коллекции объектов требуется перейти к некоторой другой коллекции объектов, которые ассоциированы с объектами исходной коллекции через некоторое соединение. В этом случае выражение над элементом исходной коллекции основывается на операции перехода по соединению.

Операции `exists`, `forAll`, `size`

В OCL определены три одноименных операции `exists` над множеством, мультимножеством и последовательностью, дополнительным параметром которых является логическое выражение. В результате каждой из этих операций выдается *true* в том и только в том случае, когда хотя бы

для одного элемента входной коллекции значением логического выражения является *true*. В противном случае результатом операции является *false*. Операции `forAll` отличаются от операций `exists` тем, что в результате каждой из них выдается *true* в том и только в том случае, когда для всех элементов входной коллекции результатом вычисления логического выражения является *true*. В противном случае результатом операции будет *false*. Операция `size` применяется к коллекции и выдает число содержащихся в ней элементов.*

Операции `union, intersect, symmetricDifference`

Параметрами двуместных операций `union`, `intersect`, `symmetricDifference` являются две коллекции, причем в OCL операции определены почти для всех возможных комбинаций типов коллекции. Не будем рассматривать все определения этих операций и кратко упомянем только две из них. Результатом операции union, определенной над множеством и мультимножеством, является мультимножество, т. е. из результата объединения таких двух коллекций дубликаты не исключаются. Результатом же операции union, определенной над двумя множествами, является множество, т. е. в этом случае возможные дубликаты должны быть исключены.

Примеры инвариантов

В заключение обзора языка OCL приведем примеры четырех инвариантов, выраженных на этом языке. Будем основываться на диаграмме классов, показанной на рис. 10.14.

Пример 10.1

Определить ограничение «возраст служащих должен быть больше 18 и меньше 100 лет».

```
context Служащий inv:
self.возраст >18 and self.возраст < 100
```

Условие инварианта накладывает требуемое ограничение на значения атрибута `возраст`, определенного в классе `Служащий`. В условном выражении инварианта ключевое слово `self` обозначает текущий объект класса-контекста инварианта. Можно считать, что при проверке данного условия будут перебираться существующие объекты класса `Служащий`, и для каждого объекта будет проверяться, что значения атрибута `возраст` находятся в пределах заданного диапазона. Ограничение удовлетворяется, если условное выражение принимает значение *true* для каждого объекта класса-контекста.

* Для коллекций значений возможно также применение операций `min`, `max` и `avg`, выдающих минимальное, максимальное и среднее значение элементов коллекции соответственно.

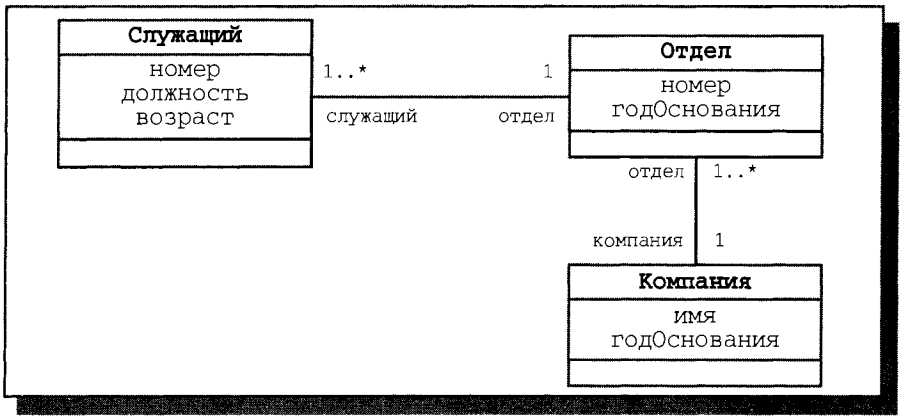


Рис. 10.14. Диаграмма классов, используемая для примеров на языке OCL

Пример 10.2

Выразить на языке OCL ограничение, в соответствии с которым в отделах с номерами больше 5 должны работать служащие старше 30 лет.

```
context Отдел inv:
self.номер ⩽ 5 or
self.служащий → select (возраст ⩽ 30) → size () = 0
```

В этом случае условное выражение инварианта будет вычисляться для каждого объекта класса `Отдел`. Подвыражение справа от операции `or` вычисляется слева направо. Сначала вычисляется подвыражение `self.служащий`, значением которого является множество объектов, соответствующих служащим, которые работают в текущем отделе. Далее к этому множеству применяется операция `select (возраст 30)`, в результате которой вырабатывается множество объектов, соответствующих служащим текущего отдела, возраст которых не превышает 30 лет. Значением операции `size ()` является число объектов в этом множестве. Все выражение принимает значение *true*, если последняя операция сравнения «=0» вырабатывает значение *true*, т. е. если в текущем отделе нет служащих младше 31 года. Ограничение в целом удовлетворяется только в том случае, если значением условия инварианта является true для каждого отдела.

Тот же инвариант можно сформулировать в контексте класса `Служащий`:

```
context Служащий inv:
self.возраст > 30 or self.отдел.номер ⩽ 5
```

Здесь следует обратить внимание на подвыражение `self.отдел.номер ⩽ 5`. Поскольку `отдел` – это имя роли соединения, значением подвыражения `self.отдел` является коллекция (множество). Но кратность роли `отдел` равна единице, т. е. каждому объекту служащего соответствует в точности один объект отдела. Поэтому в OCL допускается сокращенная запись операции `self.отдел.номер`, значением которой является номер отдела текущего служащего.

Пример 10.3

Определить ограничение, в соответствии с которым у каждого отдела должен быть менеджер, и любой отдел должен быть основан не раньше соответствующей компании:

```
context Отдел inv:
self.служащий → exists (должность = "manager") and
self.компания.годОснования ⩽ self.годОснования
```

Здесь должность – атрибут класса `Служащий`, а атрибуты с именем `годОснования` имеются и у класса `Отдел`, и у класса `Компания`. В условном выражении этого инварианта подвыражение `self.служащий → exists (должность = "manager")` эквивалентно выражению `self.служащий → select (должность = "manager") → size () > 1`. Если бы в ограничении мы потребовали, чтобы у каждого отдела был только один менеджер, то следовало бы написать ... `size () = 1`, и это было бы не эквивалентно варианту с `exists`.

Обратите внимание, что в этом случае снова законным является подвыражение `self.компания.годОснования`, поскольку кратность роли `компания` в ассоциации классов `Отдел` и `Компания` равна единице.

Пример 10.4

Условие четвертого инварианта ограничивает максимально возможное количество служащих компании числом 1000:

```
context Компания inv:
self.отдел → collect (служащие) → size ( ) < 1000
```

Здесь полезно обратить внимание на использование операции `collect`. Проследим за вычислением условного выражения. В нашем случае в классе `Компания` всего один объект, и он сразу становится текущим. В результате выполнения операции `self.отдел` будет получено множество объектов, соответствующих всем отделам компании. При выполнении операции `collect (служащие)` для каждого объекта-отдела по соединению с объектами класса `СЛУЖАЩИЕ` будет образовано множество объектов-служащих данного отдела, а в результате будет образовано мно-

жество объектов, соответствующих всем служащим всех отделов компании, т. е. всем служащим компании.

Плюсы и минусы использования языка OCL при проектировании реляционных баз данных

Плюсы и минусы использования языка OCL при проектировании реляционных БД очевидны. Язык позволяет формально и однозначно (без двусмысленностей, свойственных естественным языкам) определять ограничения целостности БД в терминах ее концептуальной схемы. Скорее всего, наличие подобной проектной документации будет полезным для сопровождения БД, даже если придется преобразовывать инварианты OCL в ограничения целостности SQL вручную.

К отрицательным сторонам использования OCL относится, прежде всего, сложность языка и неочевидность некоторых его конструкций. Кроме того, строгость синтаксиса и линейная форма языка в некотором роде противоречат наглядности и интуитивной ясности диаграммной части UML. Да, в инвариантах OCL используются те же понятия и имена, что и в соответствующей диаграмме классов, но используются совсем в другой манере. И последнее. Трудно доказать или опровергнуть как предположение, что на языке OCL можно выразить любое ограничение целостности, которое можно определить средствами SQL, так и утверждение, что на языке OCL нельзя выразить такой инвариант, для которого окажется невозможным сформулировать эквивалентное ограничение целостности на языке SQL. Лично мне неизвестны работы, в которых бы сравнивалась выразительная мощность этих языков в связи с ограничениями целостности реляционных БД.

Получение схемы реляционной базы данных из диаграммы классов UML

Если не обращать внимания на различия в терминологии*, то здесь выполняются практически те же шаги, что и в случае преобразования в схему реляционной БД ER-диаграммы. Поэтому ограничимся только некоторыми рекомендациями, специфичными для диаграмм классов.

* Очевидным аналогом *класса* является тип сущности, а аналогом *связи-ассоциации* — связь в смысле ER-модели. Кстати, различия и беспорядок в терминологии действительно удручают. В ER-модели *связь* (*relationship*) — это *ассоциация* (*association*) между двумя типами сущности. В UML *ассоциация* (*association*) — это один из видов *связи* (*relationship*). Да еще зачем-то в UML введен специальный термин *link* для обозначения экземпляра ассоциации. И снова хотелось бы использовать в качестве русского эквивалента термин *связь*, но он уже безнадежно занят, и приходится переводить *link* как *соединение*. Это, конечно, не противоречит смыслу, но тоже очень плохо, поскольку в области реляционных БД термин *соединение* и без этого имеет два разных смысла – операции соединения и соединения с сервером баз данных. Мне очень жаль переводчиков книг, посвященных UML.

Рекомендация 1. Прежде чем определять в классах операции, подумайте, что вы будете делать с этими определениями в среде целевой РСУБД. Если в этой среде поддерживаются хранимые процедуры, то, возможно, некоторые операции могут быть реализованы именно с помощью такого механизма. Но если в среде РСУБД поддерживается механизм определяемых пользователями функций, возможно, он окажется более подходящим.

Рекомендация 2. Помните, что сравнительно эффективно в РСУБД реализуются только ассоциации видов «один ко многим» и «многие ко многим». Если в созданной диаграмме классов имеются ассоциации «один к одному», следует задуматься о целесообразности такого проектного решения. Реализация в среде РСУБД ассоциаций с точно заданными кратностями ролей возможна, но требует определения дополнительных триггеров, выполнение которых понизит эффективность.

Рекомендация 3. Для технологии реляционных БД агрегатные и в особенности композитные ассоциации неестественны. Подумайте о том, что вы хотите получить в реляционной БД, объявив некоторую ассоциацию агрегатной. Скорее всего, ничего.

Рекомендация 4. В спецификации UML говорится о том, что, определяя однонаправленные связи, вы можете способствовать эффективности доступа к некоторым объектам. Для технологии реляционных баз данных поддержка такого объявления вызовет дополнительные накладные расходы и тем самым снизит эффективность.

Рекомендация 5. Не злоупотребляйте возможностями OCL.

Диаграммы классов UML – это мощный инструмент для создания концептуальных схем баз данных, но, как известно, все хорошо в меру.

Заключение

Нельзя сказать, что проектирование баз данных на основе семантических моделей в любом случае ускоряет и/или упрощает процесс проектирования. Все зависит от сложности предметной области, квалификации проектировщика и качества вспомогательных программных средств. Но так или иначе этап диаграммного моделирования обеспечивает следующие преимущества:

- на раннем этапе проектирования до привязки к конкретной РСУБД проектировщик может обнаружить и исправить логические недочеты проекта, руководствуясь наглядным графическим представлением концептуальной схемы;
- окончательный вид концептуальной схемы, полученной непосредственно перед переходом к формированию реляционной схемы, а может быть, и промежуточной версии концептуальной схемы, должен стать

частью документации целевой реляционной БД; наличие этой документации очень полезно для сопровождения и, в особенности, для изменения схемы БД в связи с изменившимися требованиями;

- при использовании CASE-средств концептуальное моделирование БД может стать частью всего процесса проектирования целевой информационной системы, что должно способствовать правильной структуризации процесса, эффективности и повышению качества проекта в целом*.

Мы также хотели показать, что в контексте проектирования реляционных БД структурные методы проектирования, основанные на использовании ER-диаграмм, и объектно-ориентированные методы, основанные на использовании языка UML, различаются, главным образом, лишь терминологией. ER-модель концептуально проще UML, в ней меньше понятий, терминов, вариантов применения. И это понятно, поскольку разные варианты ER-моделей разрабатывались именно для поддержки проектирования реляционных БД, и ER-модели почти не содержат возможностей, выходящих за пределы реальных потребностей проектировщика реляционной БД.

Язык UML принадлежит объектному миру. Этот мир гораздо сложнее (если угодно, непонятнее, запутаннее) реляционного мира. Поскольку UML может использоваться для унифицированного объектно-ориентированного моделирования всего чего угодно, в этом языке содержится масса различных понятий, терминов и вариантов использования, избыточных с точки зрения проектирования реляционных БД. Если вычленить из общего механизма диаграмм классов то, что действительно требуется для проектирования реляционных БД, то мы получим в точности ER-диаграммы с другой нотацией и терминологией.

Поэтому выбор конкретной концептуальной модели – это вопрос вкуса и сложившихся обстоятельств. Понятно, что если в организации уже имеется сложившаяся инфраструктура проектирования приложений, то разумно продолжать ею пользоваться до тех пор, пока это не станет тормозом. При построении же новой инфраструктуры стратегические соображения высшего руководства компании имеют больший вес, чем предпочтения технических специалистов, хотя эти предпочтения тоже обязательно должны учитываться.

* Хотя это несколько противоречит одной из главных исходных целей реляционного подхода к организации БД: обеспечить максимальную независимость приложения от структуры БД и увеличить возможность повторного использования существующих баз данных при возникновении новых приложений. С другой стороны, здоровая прагматика говорит о том, что, не забывая о возможных потребностях в будущем, нужно, прежде всего, стремиться к эффективному выполнению текущих задач.

Лекция 11. Язык баз данных SQL: общее введение, типы данных и средства определения доменов

Оставшаяся часть этого курса посвящается языку реляционных баз данных SQL. В курсе о реляционных базах данных невозможно обойтись без материала, который относится к этому языку. Это связано совсем не с тем, что язык представляет собой особое достижение в области реляционных БД. Напротив, многие черты SQL, начиная с самых первых его вариантов, противоречили принципам реляционной модели данных, заложенным Эдгаром Коддом. С другой стороны, спецификация языка SQL, по своей сути, является завершенной спецификацией модели данных, которая сегодня играет роль суррогата реляционной модели.

Если бы мы попытались обойтись в этом курсе без обсуждения языка SQL, курс был бы полностью оторван от жизни. Сегодня SQL является *lingua franca* в мире баз данных. Интерфейсы, основанные на SQL, поддерживаются почти во всех используемых СУБД, далеко не все из которых первоначально разрабатывались как реляционные системы. И похоже, что эта ситуация при жизни нынешнего поколения радикальным образом не изменится. Кроме того, язык сам по себе достаточно интересен. В нем нашел отражение многолетний практический опыт многих людей, и он впитал в себя многие положительные (и отрицательные) черты других языков и подходов (не только языков баз данных и не только реляционного подхода).

В данной лекции после небольшой исторической справки и краткого введения в структуру языка SQL будут рассмотрены типы данных, допустимые в языке SQL и в SQL-ориентированных базах данных, а также языковые средства определения, изменения определения и отмены определения доменов.

Ключевые слова: язык баз данных SQL, СУБД System R, SEQUEL, SQL/DS, DB2, Oracle, Informix, Sybase, Microsoft SQL Server, стандарт SQL/86, стандарт SQL/89, SQL2, стандарт SQL/92, SQL/CLI, ODBC, JDBC, SQL/PSM, SQL3, стандарт SQL:1999, SQL/Bindings, SQL/Transaction, SQL/Temporal, SQL/MED, SQL/OLB, SQL/OLAP, стандарт SQL:2003, SQL/Schemata, SQL/JRT, SQL/XML, прямой SQL, встроенный SQL, динамический SQL, базовый, промежуточный и полный уровни языка SQL, система типов языка SQL, точные числовые типы, приближенные числовые типы, типы символьных строк, типы битовых строк, типы даты и времени, типы временных интервалов, булевский тип, типы коллекций, анонимные строчные типы, типы, определяемые пользователем, ссылочные типы, истинно целые типы, точные типы, допускающие наличие дробной части, литералы типов, типы времени и временной метки с временной зоной, трехзначная логика, типы масси-

вов, конструктор типа массива, типы мультимножеств, конструктор типа мультимножества, конструктор анонимного строчного типа, структурные определяемые пользователем типы, индивидуальные определяемые пользователем типы, типизированная таблица, ссылочные значения, определение домена, оператор `CREATE DOMAIN`, значение домена по умолчанию, ограничение домена, изменение определения домена, оператор `ALTER DOMAIN`, отмена определения домена, оператор `DROP DOMAIN`, неявные преобразования типов, правила приводимости типов, явные преобразования типов или доменов, оператор `CAST`.

Введение

В начале лекции мы представим небольшой исторический обзор SQL. Язык уже далеко не молод. В 2004 г. сообщество баз данных отметило его 30-летний юбилей. Поэтому, чтобы правильно понимать и трактовать современные варианты SQL, нужно знать историю языка хотя бы в общих чертах.

Краткая история языка SQL

Язык SQL, предназначенный для взаимодействия с базами данных, появился в середине 70-х гг. (первые публикации датируются 1974 г.) и был разработан в компании IBM в рамках проекта экспериментальной реляционной СУБД System R. Исходное название языка SEQUEL (Structured English Query Language) только частично отражало суть этого языка. Конечно, язык был ориентирован главным образом на удобную и понятную пользователям формулировку запросов к реляционным БД. Но, в действительности, он почти с самого начала являлся полным языком БД, обеспечивающим помимо средств формулирования запросов и манипулирования БД следующие возможности:

- средства определения и манипулирования схемой БД;
- средства определения ограничений целостности и триггеров;
- средства определения представлений БД;
- средства определения структур физического уровня, поддерживающих эффективное выполнение запросов;
- средства авторизации доступа к отношениям и их полям*;
- средства определения точек сохранения транзакции и выполнения фиксации и откатов транзакций.

В языке отсутствовали средства явной синхронизации доступа к объектам БД со стороны параллельно выполняемых транзакций: с самого начала предполагалось, что необходимую синхронизацию неявно выполняет СУБД.

* В этом абзаце применяется терминология, которая использовалась в публикациях, посвященных System R.

В настоящее время язык SQL реализован во всех коммерческих реляционных СУБД и почти во всех СУБД, которые изначально основывались не на реляционном подходе. Все компании-производители провозглашают соответствие своей реализации стандарту SQL, и на самом деле реализованные диалекты SQL очень близки. Этого удалось добиться не сразу.

Наиболее близки к System R были две системы компании IBM – SQL/DS и DB2*. Разработчики обеих систем использовали опыт проекта System R, а СУБД SQL/DS напрямую основывалась на программном коде System R. Отсюда предельная близость диалектов SQL, реализованных в этих системах, к SQL System R. Из SQL System R были удалены только те части, которые были недостаточно проработаны (например, точки сохранения) или реализация которых вызывала слишком большие технические трудности (например, ограничения целостности и триггеры). Можно назвать этот путь к коммерческой реализации SQL движением сверху вниз.

Другой подход применялся в таких системах, как Oracle, Informix и Sybase. Несмотря на различие в способах разработки систем, реализация SQL везде происходила «снизу вверх». В первых выпущенных на рынок версиях этих систем использовалось ограниченное подмножество SQL System R. В частности, в первой известной нам реализации SQL в СУБД Oracle в операторах выборки не допускалось использование вложенных подзапросов и отсутствовала возможность формулировки запросов с соединениями нескольких отношений.

Тем не менее, несмотря на эти ограничения и на очень слабую, на первых порах, эффективность СУБД, ориентация компаний на поддержку разных аппаратных платформ и заинтересованность пользователей в переходе к реляционным системам позволили компаниям добиться коммерческого успеха и приступить к совершенствованию своих реализаций. В текущих версиях Oracle, Informix, Sybase и Microsoft SQL Server поддерживаются достаточно мощные диалекты SQL, хотя реализация иногда вызывает сомнения.**

Особенностью большинства современных коммерческих СУБД, затрудняющей сравнение существующих диалектов SQL, является отсутствие единообразного описания языка. Обычно описание разбросано по разным руководствам и перемешано с описанием специфических для данной

* Как это ни странно, компания IBM, имевшая уникальный и положительный опыт реализации экспериментальной реляционной СУБД System R, не стала первой компанией, выпустившей на рынок коммерческую реляционную СУБД. Компанию IBM опередила на два года незадолго до того образованная компания Oracle, выпустившая свою первую систему в 1979 г. Современные эксперты по разному объясняют причины этой «заторможенности» IBM, но, по всей видимости, основная причина кроется в традиционном консерватизме руководства компании.

** Например, одной из выигрышных черт SQL System R являлось то, что в одной транзакции разрешалось комбинировать все возможные операторы SQL. Поскольку технически это обеспечить достаточно трудно, почти во всех сегодняшних SQL-ориентированных СУБД имеются ограничения на состав операторов, которые можно выполнять в одной транзакции.

системы языковых средств, не имеющих прямого отношения к SQL. Тем не менее, можно сказать, что базовый набор операторов SQL, включающий операторы определения схемы БД, выборки и манипулирования данными, авторизации доступа к данным, поддержки встраивания SQL в языки программирования и операторы динамического SQL, в коммерческих реализациях устоялся и более или менее соответствует стандарту.

Деятельность по стандартизации языка SQL началась практически одновременно с появлением его первых коммерческих реализаций. В 1982 г. комитету по базам данных Американского национального института стандартов (ANSI) было поручено разработать спецификацию стандартного языка реляционных баз данных. Первый документ из числа имеющихся у автора проектов стандарта датирован октябрем 1985 г. и является уже не первым проектом стандарта ANSI. Стандарт был принят ANSI в 1986 г., а в 1987 г. одобрен Международной организацией по стандартизации (ISO). Этот стандарт принято называть SQL/86.

Понятно, что в качестве основы стандарта нельзя было использовать SQL System R. Во-первых, этот вариант языка не был должным образом технически проработан. Во-вторых, его слишком сложно было бы реализовать (кто знает, как бы сложилась судьба SQL, если бы все идеи проекта System R были реализованы полностью). Поэтому за основу был взят диалект языка SQL, сложившийся в IBM к началу 1980-х гг. В сущности, этот диалект представлял собой технически проработанное подмножество SQL System R.

К 1989 г. стандарт SQL/86 был несколько расширен, и был подготовлен и принят следующий стандарт, получивший название ANSI/ISO SQL/89. Анализ доступных документов показывает, что процесс стандартизации SQL происходил очень сложно с использованием не только научных доводов. В результате SQL/89 во многих частях имеет чрезвычайно общий характер и допускает очень широкое толкование. В этом стандарте полностью отсутствуют такие важные разделы, как манипулирование схемой БД и динамический SQL. Многие важные аспекты языка в соответствии со стандартом определяются в реализации.*

Возможно, наиболее важными достижениями стандарта SQL/89 являются четкая стандартизация синтаксиса и семантики операторов выборки данных и манипулирования данными и фиксация средств ограничения целостности БД. Были специфицированы средства определения первичного и внешних ключей отношений и так называемых проверочных ограничений целостности, которые представляют собой подмножество немедленно проверяемых ограничений целостности SQL System R. Средства определения внешних ключей позволяют легко формулировать

* Это практически обесценивает стандарт с точки зрения программистов приложений баз данных, поскольку не дает возможности создавать приложения, не привязанные к особенностям конкретных СУБД.

требования так называемой ссылочной целостности БД. Это распространенное в реляционных БД требование можно было сформулировать и на основе общего механизма ограничений целостности SQL System R, но формулировка на основе понятия внешнего ключа более проста и понятна.

Осознавая неполноту стандарта SQL, на фоне завершения разработки этого стандарта специалисты различных компаний начали работу над стандартом SQL2. Эта работа также длилась несколько лет, было выпущено множество проектов стандарта, пока наконец в марте 1992 г. не был принят окончательный проект стандарта (SQL/92). Этот стандарт существенно полнее стандарта SQL/89 и охватывает практически все аспекты, необходимые для реализации приложений: манипулирование схемой БД, управление транзакциями (появились точки сохранения) и сессиями (сессия – это последовательность транзакций, в пределах которой сохраняются временные отношения), подключения к БД, динамический SQL. Наконец, были стандартизованы отношения-каталоги БД, что вообще-то не связано непосредственно с языком, но очень сильно влияет на реализацию.*

В 1995 г. стандарт был дополнен спецификацией интерфейса уровня вызова (Call-Level Interface – SQL/CLI). SQL/CLI представляет собой набор спецификаций интерфейсов процедур, вызовы которых позволяют выполнять динамически задаваемые операторы SQL. По сути дела, SQL/CLI представляет собой альтернативу динамическому SQL. Интерфейсы процедур определены для всех основных языков программирования: C, Ada, Pascal, PL/1 и т. д. Следует заметить, что стандарт SQL/CLI послужил основой для создания повсеместно распространенных сегодня интерфейсов ODBC (Open Database Connectivity) и JDBC (Java Database Connectivity).

В 1996 г. к стандарту SQL/92 был добавлен еще один компонент – SQL/PSM (Persistent Stored Modules). Основная цель этой спецификации состоит в том, чтобы стандартизировать способы определения и использования *хранимых процедур*, т. е. специальным образом оформленных программ, включающих операторы SQL, которые сохраняются в базе данных, могут вызываться приложениями и выполняются внутри СУБД.

Незадолго до завершения работ по определению стандарта SQL2 была начата разработка стандарта SQL3. Первоначально планировалось завершить проект в 1995 г. и включить в язык некоторые объектные возможности: определяемые пользователями типы данных, поддержку триггеров, под-

* Среди прочих достижений System R нельзя не отметить то, что в базах данных, управляемых этой СУБД, хранились как данные, так и метаданные – описатели отношений, их полей, представлений, ограничения целостности и т.д. Для хранения метаданных использовались специальные служебные отношения, которые стали называть отношениями-каталогами. Из отношений-каталогов можно было выбрать данные с помощью обычных средств языка SQL. Конечно, организация служебных данных – это вопрос реализации SQL, но этот вопрос непосредственно касается потенциальных пользователей SQL-ориентированных СУБД, и поэтому стандартизация представления пользователю отношений-каталогов (в стандарте SQL, информационной схемы базы данных) является исключительно важным делом.

держку темпоральных свойств данных и т. д. Реально работу над новым стандартом удалось частично завершить только в 1999 г., и по этой причине (а также в связи с проблемой 2000 года) стандарт получил название SQL:1999.

Приведем краткую характеристику текущего состояния стандарта SQL:1999 и перспектив его развития. Прежде всего, заметим, что каждый новый вариант стандарта языка SQL был существенно объемнее предыдущих версий. Так, если стандарт SQL/89 занимал около 600 страниц, то объем SQL/92 составлял на 300 с лишним страниц больше. Самые первые проекты SQL3 занимали около 1500 страниц. Это вполне естественно, потому что язык усложняется, а его спецификации становятся более детальными и точными. Но разработчики SQL3 пришли к выводу, что при таких объемах стандарта вероятность его принятия и последующей успешной поддержки заметно уменьшается. Поэтому было принято решение разбить стандарт на относительно независимые части, которые можно было бы разрабатывать и поддерживать по отдельности.

В 1999 г. были приняты пять первых частей стандарта SQL:1999. Первая часть (SQL/Framework) посвящена описанию концептуальной структуры стандарта. В этой части приводится развернутая аннотация следующих четырех частей и формулируются требования к реализациям, претендующим на соответствие стандарту.

Вторая часть SQL:1999 (SQL/Foundation) образует базис стандарта. Вводится система типов языка, формулируются правила определения функциональных зависимостей и возможных ключей, определяются синтаксис и семантика основных операторов SQL:

- операторов определения и манипулирования схемой базы данных;
- операторов манипулирования данными;
- операторов управления транзакциями;
- операторов управления подключениями к базе данных и т. д.

Третью часть занимает уточненная по сравнению с SQL/92 спецификация SQL/CLI. В четвертой части специфицируется SQL/PSM – синтаксис и семантика языка определения хранимых процедур. Наконец, в пятой части – SQL/Bindings – определяются правила связывания SQL для стандартных версий языков программирования FORTRAN, COBOL, PL/1, Pascal, Ada, C и MUMPS.

В стандарт SQL:1999 должны были войти еще несколько частей. Среди них спецификации следующих средств:

- управление распределенными транзакциями (SQL/Transaction);
- поддержка темпоральных свойств данных (SQL/Temporal);
- управление внешними данными (SQL/MED);
- связывание с объектно-ориентированными языками программирования (SQL/OLB);
- поддержка оперативной аналитической обработки (SQL/OLAP).

В конце 2003 г. был принят и опубликован новый вариант международного стандарта SQL:2003. Многие специалисты считали, что в варианте стандарта, следующем за SQL:1999, будут всего лишь исправлены неточности SQL:1999. Но на самом деле, в SQL:2003 специфицирован ряд новых и важных свойств, часть из которых мы затронем в этом курсе.

Претерпела некоторые изменения общая организация стандарта. Стандарт SQL:2003 состоит из следующих частей:

- 9075-1, SQL/Framework;
- 9075-2, SQL/Foundation;
- 9075-3, SQL/CLI;
- 9075-4, SQL/PSM;
- 9075-9, SQL/MED;
- 9075-10, SQL/OLB;
- 9075-11, SQL/Schemata;
- 9075-13, SQL/JRT;
- 9075-14, SQL/XML.

Части 1-4 и 9-10 с необходимыми изменениями остались такими же, как и в SQL:1999 (разд. 7.4). Часть 5 (SQL/Bindings) перестала существовать; соответствующие спецификации включены в часть 2. Раздел части 2 SQL:1999, посвященный информационной схеме, выделен в отдельную часть 11. Появились две новые части – 13 и 14. Часть 13 полностью называется «SQL Routines and Types Using the Java Programming Language» («Использование подпрограмм и типов SQL в языке программирования Java»). Появление такой части стандарта оправдано повышенным вниманием к языку Java со стороны ведущих производителей SQL-ориентированных СУБД. Наконец, последняя часть SQL:2003 посвящена спецификациям языковых средств, позволяющих работать с XML-документами в среде SQL.

На мой взгляд, текущее состояние процесса стандартизации языка SQL отражает текущее состояние технологии SQL-ориентированных баз данных. Ведущие поставщики соответствующих СУБД (сегодня это компании IBM, Oracle и Microsoft) стараются максимально быстро реагировать на потребности и конъюнктуру рынка и расширяют свои продукты все новыми и новыми возможностями. Очевидна потребность в стандартизации соответствующих языковых средств, но процесс стандартизации явно не поспевает за происходящими изменениями.

Структура языка SQL

В данной лекции мы начинаем систематически описывать базовые механизмы языка SQL. Чтобы пояснить, о какой части языка пойдет речь в этой и следующих лекциях, обратимся к рис. 11.1.

Язык SQL, соответствующий последним стандартам SQL:2003, SQL:1999 (и даже SQL/92), – это очень богатый и сложный язык, все воз-

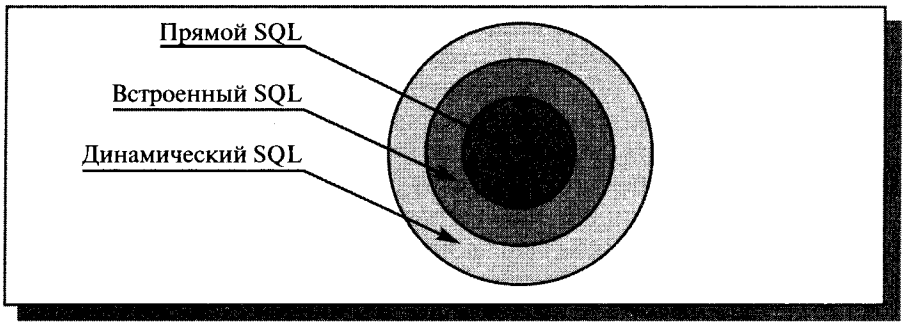


Рис. 11.1. Один из способов разделения языка SQL на уровни

можности которого трудно сразу осознать и тем более понять. Поэтому приходится разбивать язык на уровни, или слои, такие, что каждый уровень языка включает все конструкции, входящие в более низкие уровни. В стандарте определяется несколько способов разбиения языка на уровни. В одной из классификаций язык разбивается на *«базовый» (entry)*, *«промежуточный» (intermediate)* и *«полный» (full)* уровни.

Эта классификация ориентирована, прежде всего, на производителей СУБД, в которых поддерживается SQL. Реализация базового уровня языка является обязательным условием хотя бы какого-то соответствия стандарту. Реализация промежуточного уровня желательна, и обычно именно такой уровень языка поддерживается ведущими компаниями-производителями SQL-ориентированных СУБД. Наконец, полный уровень языка является целью, к достижению которой следует стремиться. В данной классификации критерием отнесения той или иной возможности языка к некоторому уровню является оцениваемая создателями стандарта SQL (большая часть которых является сотрудниками ведущих компаний, производящих SQL-ориентированные СУБД) техническая сложность реализации этой возможности. Конечно, такая классификация важна и для программистов приложений баз данных, но только для того, чтобы оценить реальные возможности конкретной СУБД. Для понимания языка SQL это разбиение на уровни несущественно.

Другая классификация показана на рис. 11.1. Среди всех конструкций языка SQL можно выделить такие конструкции, которые можно использовать при *«прямом» (direct)* взаимодействии конечного пользователя с СУБД (например, в интерактивном режиме). В некотором смысле этот уровень также является базовым, поскольку соответствующие средства языка в наибольшей степени отражают его ориентированность на работу с мультимножествами. На следующем уровне, уровне *«встраиваемого» (embedded)* SQL, язык расширяется конструкциями, позволяющими использовать возможности прямого SQL в программах, написанных на традиционных язы-

ках программирования. Наконец, на уровне *«динамического» (dynamic)* SQL во встраиваемый SQL добавляются конструкции, позволяющие приложениям обращаться к СУБД с конструкциями прямого SQL, которые динамически образуются во время выполнения программы.

Нам кажется, что вторая классификация является более полезной для читателя, постигающего основы языка SQL. По нашему мнению, дополнительные возможности, присутствующие во встраиваемом и в динамическом SQL, не слишком сильно влияют на модельное представление языка. Конечно, возможности встраиваемого и динамического SQL необходимо хорошо знать разработчикам приложений SQL-ориентированных баз данных. Но поскольку задачей этого курса не является обучение использованию языка SQL при программировании приложений баз данных, мы не будем затрагивать эти темы. Обратимся к прямому SQL, причем не в полном объеме стандартов SQL:2003 и SQL:1999 (этого не позволяет сделать объем курса). Обсудим только наиболее важные аспекты.

В этой лекции обсуждаются основные аспекты системы типов данных языка SQL и средства определения доменов.

Замечание: Лекции, посвященные языку SQL, опираются, главным образом, на стандарт SQL:1999. В тех случаях, когда будут упоминаться дополнительные возможности, специфицированные в наиболее свежей версии стандарта – SQL:2003, мы будем явно на это указывать. Поэтому здесь мы используем терминологию стандарта (таблицы, строки, столбцы и т. д.).*

Типы данных SQL

Данные, хранящиеся в столбцах таблиц SQL-ориентированной базы данных, являются типизированными, т. е. представляют собой значения одного из типов данных, предопределенных в языке SQL или определяемых пользователями путем применения соответствующих средств языка. Для этого при определении таблицы каждому ее столбцу назначается некоторый тип данных (или домен), и в дальнейшем СУБД должна следить, чтобы в каждом столбце каждой строки каждой таблицы присутствовали только допустимые значения. В этом разделе мы обсудим систему типов языка SQL.

Все допустимые в SQL типы данных, которые можно использовать при определении столбцов**, разбиваются на следующие категории***:

* К сожалению, приходится использовать термин *строка* в двух смыслах: *строка таблицы (table row)* и *символьная или битовая строка (character or bit string)*. Постараемся обеспечить правильное понимание смысла термина в контексте его использования.

** А также переменных, параметров и других типизированных объектов языка SQL, которые мы не затрагиваем в этом курсе.

*** В этот список не включен тип данных XML, поскольку в данном курсе вообще не рассматриваются проблемы управления базами XML-данных.

- точные числовые типы (*exact numerics*);
- приближенные числовые типы (*approximate numerics*);
- типы символьных строк (*character strings*);
- типы битовых строк (*bit strings*)*;
- типы даты и времени (*datetimes*);
- типы временных интервалов (*intervals*);
- булевский тип (*Booleans*);
- типы коллекций (*collection types*);
- анонимные строчные типы (*anonymous row types*);
- типы, определяемые пользователем (*user-defined types*);
- ссылочные типы (*reference types*).

В столбцах таблиц, определенных на любых типах данных, наряду со значениями этих типов, допускается сохранение неопределенного значения, которое обозначается ключевым словом NULL. В языке определено, что результатом выражений вида *x* a_op NULL, NULL a_op *x*, NULL a_op NULL является NULL для всех арифметических операций a_op (+, - и т. д.), допустимых для типа данных выражения *x* (выражение NULL a_op NULL является допустимым для любой арифметической операции a_op). Также по определению полагается, что значением выражений *x* comp_op NULL, NULL comp_op *x*, NULL comp_op NULL для всех операций сравнения (=, ≠, >, < и т. д.), определенных для типа выражения *x*, является третье логическое значение *unknown*** (выражение NULL comp_op NULL является допустимым для любой операции сравнения comp_op).

Точные числовые типы

К категории точных числовых типов в SQL относятся те типы, значения которых точно представляют числа. Типы данных этой категории распадаются на две части: истинно целые типы (INTEGER и SMALLINT) и типы, допускающие наличие дробной части (NUMERIC и DECIMAL). Охарактеризуем эти типы данных более подробно.

Истинно целые типы

- Тип INTEGER. Тип служит для представления целых чисел. Точность чисел (число сохраняемых бит) определяется в реализации. При определении столбца данного типа достаточно указать просто INTEGER.
- Тип SMALLINT. Тип также служит для представления целых чисел. Точность определяется в реализации, но она не должна быть больше точности типа INTEGER. При определении столбца указывается просто SMALLINT.***

* Спецификация предопределенного типа данных битовых строк была удалена в стандарте SQL:2003. Но поскольку эта спецификация появилась только в SQL:1999, мы сочли уместным оставить в курсе обсуждение этого типа данных.

** См. ниже *Булевский тип*.

*** Следует подчеркнуть, что в стандарте SQL не определяется число байт, занимаемых при хранении в памяти значений целых типов. Не следует думать, что в SQL для хранения значения типа INTEGER требуется четыре байта, а SMALLINT требует двух байтов.

- Литералы типов целых чисел представляются в виде строк символов, изображающих десятичные числа; в начале строки могут присутствовать символы «+» или «-» (если символ знака отсутствует, подразумевается «+»). Примеры литералов типов INTEGER и SMALLINT: 1826545, 876.

Точные типы, допускающие наличие дробной части

- Тип NUMERIC. На самом деле, это не просто тип данных, а параметризуемый тип. При определении столбца можно указать спецификацию NUMERIC (p, s), где p и s – литералы истинно целого типа, и p задает точность значений (число сохраняемых бит), а s – шкалу (число десятичных цифр в дробной части). Задаваемая шкала не должна быть отрицательной и не должна превышать значение точности. При определении столбца можно использовать сокращенные формы спецификации типа – NUMERIC и NUMERIC (p). Первая форма предполагает использование точности, определяемое по умолчанию в реализации, и шкалы, равной нулю, а вторая – использование заданной точности и шкалы, равной нулю. Допустимые диапазоны значений p и s определяются в реализации.
- Тип DECIMAL. Этот тип аналогичен типу NUMERIC. Отличие состоит в том, что если при определении столбца типа DECIMAL задается точность p, то на самом деле используется точность m, определяемая в реализации, такая, что $m \geqslant p$. Шкала всегда устанавливается такой, как явно или неявно (по умолчанию) задается. При указании типа столбца можно использовать спецификации DECIMAL, DECIMAL (p) и DECIMAL (p, s).
- Литералы типов точных чисел, допускающих наличие дробной части, представляются в виде строк символов, изображающих десятичные числа, в начале которых могут присутствовать символы «+» или «-» (если символ знака отсутствует, подразумевается «+»), а внутри последовательности цифр может присутствовать символ «.». Примеры литералов типов NUMERIC и DECIMAL: 125, 26.36.

Приближенные числовые типы

К категории приближенных числовых типов в SQL относятся те типы, значения которых представляют числа приближенным образом. Приближенные числа представляются в виде пары <*мантисса*, *порядок*>, где *мантисса* состоит из значащих цифр числа, а *порядок* определяет реальный размер числа. В реализациях приближенным числовым типам SQL обычно соответствуют типы с плавающей точкой. В SQL поддерживаются три варианта приближенных числовых типов.

- Тип REAL. Значения типа соответствуют числам с плавающей точкой одинарной точности. Точность определяется в реализации, но обычно совпадает с точностью одинарной плавающей арифметики, поддерживаемой на аппаратной платформе, которая используется реализацией. При определении столбца указывается просто REAL.

- Тип DOUBLE PRECISION. Точность значений этого типа определяется в реализации, но она должна быть больше точности типа REAL. Обычно приближенным числам SQL с двойной точностью соответствуют поддерживаемые аппаратурой числа с плавающей точкой двойной точности. При определении столбца указывается просто DOUBLE PRECISION.
- Тип FLOAT. Это параметризуемый тип, значение параметра p которого задает необходимую точность значений. Требуется, чтобы реально обеспечиваемая реализацией точность значений была не меньше p. Допустимый диапазон значений параметра p определяется в реализации. При определении столбца можно указать либо FLOAT (p), либо просто FLOAT. В последнем случае подразумевается точность, определяемая реализацией по умолчанию.
- Литералы приближенных числовых типов представляются в виде литерала точного числового типа, за которым могут следовать символ «E» и литерал целого числового типа. Примеры литералов приближенных числовых типов: 123, 123.12, 123E12, 123.12E12. Литеральное выражение xEy представляет значение $x*(10^y)$.

Типы символьных строк

В SQL определены три параметризуемых типа символьных строк: CHARACTER (или CHAR), CHARACTER VARYING (или CHAR VARYING, или VARCHAR) и CHARACTER LARGE OBJECT (или CLOB).*

- Тип CHARACTER. Значениями типа являются символьные строки. Конкретный набор допустимых символов определяется в реализации, но, как правило, включает набор символов ASCII. При определении столбца допускается использование спецификаций CHARACTER (x) и просто CHARACTER. Последний вариант эквивалентен заданию CHARACTER (1). После определения столбца типа CHARACTER (x) СУБД будет резервировать место для хранения x символов этого столбца во всех строках соответствующей таблицы. Если, например, определен столбец типа CHARACTER (8), и в некоторой строке таблицы в него заносится символьная строка длиной пять символов, то реально будут храниться восемь символов, последние три из которых будут пробелами**.
- Тип CHARACTER VARYING. При определении столбца допускается использование спецификаций CHARACTER VARYING (x) и просто CHARACTER VARYING. Последний вариант эквивалентен заданию CHARACTER VARYING (1). Если в некоторой таблице определяется столбец типа CHARACTER VARYING (x), то в каждой строке этой таблицы значения

* В контексте локализации SQL-ориентированной СУБД (средства локализации входят в стандарт языка) можно определить еще три типа символьных строк – NATIONAL CHARACTER, NATIONAL CHARACTER VARYING и NATIONAL CHARACTER LARGE OBJECT. Аспекты интернационализации и локализации составляют отдельное измерение языка и не обсуждаются в данном курсе.

** Именно пробелами, а не *«пустыми»* символами!

данного столбца будут занимать ровно столько места, сколько требуется для сохранения соответствующей символьной строки (но ни одна такая строка не может состоять более чем из x символов).*

- Определен ряд операций, которые можно выполнять над символьными строками. Перечислим некоторые из них.
  - Операция конкатенации (обозначается в виде «||») возвращает символьную строку, произведенную путем соединения строк-операндов в том порядке, в каком они заданы.
  - Функция выделения подстроки (SUBSTRING) принимает три аргумента – строку, номер начальной позиции и длину – и возвращает строку, выделенную из строки-аргумента в соответствии со значениями двух последних параметров.
  - Функция UPPER возвращает строку, в которой все строчные буквы строки-аргумента заменяются прописными. Функция LOWER, наоборот, заменяет в заданной строке все прописные буквы строчными.
  - Функция определения длины (CHARACTER_LENGTH, OCTET_LENGTH, BIT_LENGTH) возвращает длину заданной символьной строки в символах, октетах или битах (в зависимости от вида вычисляющей функции) в виде целого числа.
  - Функция определения позиции (POSITION) определяет первую позицию в строке S, с которой в нее входит заданная строка S1 (если не входит, то возвращается значение нуль).
- Тип CHARACTER LARGE OBJECT. Этот тип данных предназначен для определения столбцов, хранящих большие и разные по размеру группы символов. При определении столбца задается спецификация CLOB (z), где z задает максимальный размер соответствующей группы символов. Максимально возможное значение параметра z определяется в реализации, но, очевидно, что оно должно быть существенно больше максимально возможного значения параметра x, присутствующего в типах CHAR и CHAR VARYING.**
- Литералы типов символьных строк представляются в виде последовательностей символов, заключенных в одинарные или двойные кавычки. В первом случае среди набора символов литерала допускается наличие символов двойной кавычки, а во втором – символов одинарной кавычки. Примеры литералов символьных строк: 'ABCDEF', 'Ab"Ctd', "Fbcdef", "ab'cdtF".

Типы битовых строк

В SQL определены три параметризуемых типа битовых строк: BIT, BIT VARYING и BINARY LARGE OBJECT (или BLOB).

* Максимально допустимая длина строк постоянного и переменного размера (значение параметра x) определяется в реализации.

** Поскольку значения z могут быть очень большими, допускается сокращенная форма их задания в виде nK, nM и nG, где n – положительное целое число, а K, M и G означают *кило*, *мега* и *гига* соответственно.

- Тип BIT. Значениями типа являются битовые строки. При определении столбца допускается использование спецификаций BIT (x) и просто BIT. Последний вариант эквивалентен заданию BIT (1). После определения столбца типа BIT (x) СУБД будет резервировать место для хранения x бит этого столбца во всех строках соответствующей таблицы.
- Тип BIT VARYING. При определении столбца допускается использование только спецификации без умолчания вида BIT VARYING (x), где значение x определяет максимальную длину битовой строки, которую можно хранить в данном столбце.
- Над битовыми строками определен ряд операций. Некоторые из них мы рассмотрим.
  - Битовая конкатенация (обозначается в виде ||), которая возвращает результирующую битовую строку, полученную путем конкатенации строк-аргументов в том порядке, в котором они заданы.
  - Функция извлечения подстроки из битовой строки. Синтаксис и семантика этой функции идентичны синтаксису и семантике функции SUBSTRING для символьных строк, за исключением того, что первый аргумент и возвращаемое значение являются битовыми строками.
  - Функция определения длины (OCTET_LENGTH, BIT_LENGTH) возвращает длину заданной битовой строки в октетах или битах в зависимости от выбранной функции.
  - Функция определения позиции (POSITION) определяет первую позицию в битовой строке S, с которой в нее входит строка S1. Если строка S1 не входит в строку S, возвращается значение нуль.
- Тип BINARY LARGE OBJECT. Этот тип данных предназначен для определения столбцов, хранящих большие и разные по размеру группы байтов. При определении столбца задается спецификация BLOB (z), где z задает максимальный размер соответствующей группы байтов. С технической точки зрения типы CLOB и BLOB очень похожи. Их разделение требуется для того, чтобы подчеркнуть, что значения типа CLOB состоят из символов (в частности, в них может осмысленно производиться текстовый поиск), а значения типа BLOB состоят из произвольных байтов, не обязательно кодирующих символы.
- Литералы типов битовых строк представляются как заключенные в одинарные кавычки последовательности символов «0» и «1», предваряемые символом «B»; или предваряемые символом «X» последовательности символов, которые изображают шестнадцатеричные цифры (за цифрой «9» следуют «A», «B», «C», «D», «E» и «F»). Примеры литералов типов битовых строк: B'01110011110001111111111', X'78FBCD0012FFFFA'.*

* В литерале BLOB всегда должно содержаться четное число шестнадцатиричных цифр.

Типы даты и времени

Возможность сохранения в базе данных информации о дате и времени очень важна с практической точки зрения. Достаточно вспомнить взбудоражившую весь мир «проблему 2000 года», одним из основных источников которой было некорректное хранение дат в базах данных. В стандарте SQL поддержке средств работы с датой и временем уделяется большое внимание. В частности, поддерживаются специальные «темпоральные» типы данных `DATE`, `TIME`, `TIMESTAMP`, `TIME WITH TIME ZONE` и `TIMESTAMP WITH TIME ZONE`. Коротко обсудим эти типы.

Тип даты

- Тип `DATE`. Значения этого типа состоят из компонентов-значений года, месяца и дня некоторой даты. Значение года состоит из четырех десятичных цифр и соответствует летоисчислению от Рождества Христова до 9999 г. Значение месяца состоит из двух десятичных цифр и варьируется от 01 до 12. Значение номера дня месяца состоит из двух десятичных цифр и варьируется от 01 до 31, хотя значение месяца даты может накладывать ограничения на возможность использования значений дня месяца 29, 30 и 31. В стандарте SQL не накладываются какие-либо ограничения на внутренний способ представления дат, используемый в реализации. При определении столбца типа `DATE` указывается просто `DATE`.
- Литералы типа `DATE` представляются в виде строки «`'yyyy-mm-dd'`, где символы `y`, `m` и `d` должны изображать десятичные числа. Например, литерал `DATE '1949-04-08'` представляет дату 8 апреля 1949 г.

Типы времени

- Тип `TIME`. Значения этого параметризованного типа состоят из компонентов-значений часа, минуты и секунды некоторого времени суток. Значение часа состоит ровно из двух десятичных цифр и варьируется от 00 до 23. Значение минуты состоит из двух десятичных цифр и варьируется от 00 до 59. Основное значение секунды также состоит из двух цифр, но может включать дополнительные цифры, представляющие доли секунды. Так что в целом значение секунды варьируется от 00 до 61.999... В значении времени присутствуют две лишние секунды, поскольку Всемирная служба времени иногда добавляет две секунды к последней минуте года для синхронизации мирового времени с реальным. Решение о поддержке этих «високосных» секунд принимается на уровне реализации. Число цифр в доле секунды также определяется в реализации. В стандарте требуется только то, чтобы это число было не меньше шести. При определении столбца типа `TIME` может указываться `TIME (p)` (значение `p` задает точность долей секунды) или просто `TIME` (в этом случае доли секунды не учитываются).
- Литералы типа `TIME` представляются в виде строки `TIME 'hh:mm-ss:f...f'`, где символы `h`, `m`, `s` и `f` должны изображать десятичные чис-

ла. Например, литерал TIME '16:33-20:333' представляет время суток 16 часов 33 минуты 20 и 333 тысячных секунды.

Типы временной метки

- Тип TIMESTAMP. Значения этого параметризованного типа состоят из компонентов — значений года, месяца и дня некоторой даты, а также компонентов — значений часа, минуты и секунды некоторого времени суток (т. е. каждое значение задает некоторую абсолютную временную метку – отсюда название типа TIMESTAMP). Число десятичных цифр в значениях-компонентах и ограничения этих значений такие же, как у значений типов DATE и TIME. При определении столбца типа TIMESTAMP может указываться TIMESTAMP (p) (значение p задает точность долей секунды) или просто TIMESTAMP (в этом случае, в отличие от типа данных TIME, по умолчанию принимается, что в доли секунды используются шесть десятичных цифр). Максимально допустимое значение p определяется в реализации.
- Литералы типа TIMESTAMP представляются в виде строки TIMESTAMP 'yyyy-mm-dd hh:mm-ss:f...f', где символы y, m, d, h, m, s и f должны изображать десятичные числа. Например, литерал TIMESTAMP '1949-04-08 16:33-20:333' представляет временную метку 16 часов 33 минуты 20 и 333 тысячных секунды 8 апреля 1949 г.

Типы времени и временной метки с временной зоной

- Тип TIME WITH TIME ZONE. Этот тип данных похож на тип TIME с тем лишь отличием, что значения типа TIME WITH TIME ZONE включают дополнительный компонент — значение, характеризующее смещение соответствующего времени относительно гринвичского времени (теперь его называют UTC – *universal time coordinated*). Деталей представления этого дополнительного компонента мы касаться не будем.
- Тип TIMESTAMP WITH TIME ZONE. Этот тип данных отличается от типа TIMESTAMP тем, что значения типа TIMESTAMP WITH TIME ZONE включают дополнительный компонент-значение, характеризующее смещение соответствующего времени относительно гринвичского.

Типы временных интервалов

Вообще говоря, *временным интервалом* называется разность между двумя значениями даты или времени. В SQL определены две категории типов временных интервалов: «год-месяц» и «день-время суток». Временные интервалы языка SQL не привязываются к начальному и/или конечному значению даты/времени, а описывают только протяженность во времени. В общем случае при определении столбца типа временного интервала указывается INTERVAL start (p) [TO end (q)], где в качестве «start» и «end» могут задаваться YEAR, MONTH, DAY, HOUR, MINUTE и SECOND. Параметр p задает требуемую точность лидирующего поля интервала (число десятичных цифр).

Параметр q может задаваться только в том случае, когда в качестве end используется SECOND, и указывает точность долей секунды. Если говорить более точно, возможны следующие вариации типов временных интервалов.

- Типы категории «год-месяц». Можно определить столбцы следующих типов: INTERVAL YEAR, INTERVAL YEAR (p) (значения этих типов – временные интервалы в годах), INTERVAL MONTH, INTERVAL MONTH (p) (значения этих типов – временные интервалы в месяцах), INTERVAL YEAR TO MONTH, INTERVAL YEAR (p) TO MONTH (значения этих типов – временные интервалы в годах и месяцах). Если значение параметра p не указывается явно, по умолчанию принимается его значение «2».
- Типы категории «день-время суток». При определении столбца можно использовать следующие комбинации (для полноты перечислим все):

```
INTERVAL DAY (p),
INTERVAL DAY,
INTERVAL DAY (p) TO HOUR,
INTERVAL DAY TO HOUR,
INTERVAL DAY (p) TO MINUTE,
INTERVAL DAY TO MINUTE,
INTERVAL DAY (p) TO SECOND (q),
INTERVAL DAY TO SECOND (q),
INTERVAL DAY (p) TO SECOND,
INTERVAL DAY TO SECOND,
INTERVAL HOUR (p),
INTERVAL HOUR, INTERVAL HOUR (p) TO MINUTE,
INTERVAL HOUR TO MINUTE,
INTERVAL HOUR (p) TO SECOND (q),
INTERVAL HOUR TO SECOND (q),
INTERVAL HOUR TO SECOND,
INTERVAL MINUTE (p),
INTERVAL MINUTE,
INTERVAL MINUTE (p) TO SECOND (q),
INTERVAL MINUTE TO SECOND (q),
INTERVAL MINUTE (p) TO SECOND,
INTERVAL MINUTE TO SECOND,
INTERVAL SECOND (p, q),
INTERVAL SECOND (p),
INTERVAL SECOND.
```

Если значение параметра p не указывается явно, по умолчанию принимается его значение «2». Значением параметра q по умолчанию является «6».

- Приведем только один пример литерала одной из разновидностей типа INTERVAL: INTERVAL '10:20' MINUTE TO SECOND – временной интервал в 10 минут и 20 секунд.

Тип первого операнда	Операция	Тип второго операнда	Тип результата
Datetime	-	Datetime	Interval
Datetime	+ или -	Interval	Datetime
Interval	+	Datetime	Datetime
Interval	+ или -	Interval	Interval
Interval	* или /	Numeric	Interval
Numeric	*	Interval	Interval

- Над значениями темпоральных типов могут выполняться арифметические операции, смысл которых определяется следующей таблицей:

Значения типов данных временных интервалов образуются при вычитании одного значения типа даты или времени суток из другого значения соответствующего типа. При добавлении интервального значения к значению типа даты/времени образуется новое значение типа даты/времени. Кроме того, значение интервального типа можно умножать и делить на числовые значения, получая новое значение интервального типа.

Булевский тип

При определении столбца булевского типа указывается просто спецификация BOOLEAN. Булевский тип состоит из трех значений: *true*, *false* и *unknown* (соответствующие литералы обозначаются TRUE, FALSE и UNKNOWN). Поддерживается возможность построения булевских выражений, которые вычисляются в трехзначной логике. Таблицы истинности основных логических операций показаны на рис. 11.2.

Типы коллекций

Начиная с SQL:1999, в языке поддерживается возможность использования типов данных, значения которых являются коллекциями значений некоторых других типов. Обычно под термином *коллекция* понимается одно из следующих образований: *массив*, *список*, *множество* и *мультимножество*. В варианте SQL:1999, принятом в 1999 г., были специфицированы только типы массивов. В новом стандарте SQL:2003 появилась спецификация типа мультимножества.

Типы массивов

Любой возможный тип массива получается путем применения *конструктора типов* ARRAY. При определении столбца, значения которого

* В стандарте SQL:2003 имеется следующее уточнение: «В этой спецификации не проводится различие между NULL-значением булевского типа данных и истинностным значением UKNOWN, являющимся результатом вычисления предиката, условия поиска или булевского выражения; они могут использоваться взаимозаменяемо и означают в точности одно и то же». С моей точки зрения такой подход во многом является некорректным, но я не буду здесь на этом останавливаться.

AND	TRUE	FALSE	UNKNOWN
TRUE	TRUE	FALSE	UNKNOWN
FALSE	FALSE	FALSE	FALSE
UNKNOWN	UNKNOWN	FALSE	UNKNOWN

OR	TRUE	FALSE	UNKNOWN
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	UNKNOWN
UNKNOWN	TRUE	UNKNOWN	UNKNOWN

NOT	
TRUE	FALSE
FALSE	TRUE
UNKNOWN	UNKNOWN

Рис. 11.2. Таблицы истинности основных логических операций в трехзначной логике

должны принадлежать некоторому типу массива, используется конструкция `dt ARRAY [ mc ]`, где `dt` специфицирует некоторый допустимый в SQL тип данных, а `mc` является литералом некоторого точного числового типа с нулевой длиной шкалы и определяет максимальное число элементов в значении типа массива (в терминологии SQL:1999 это значение называется *максимальной кардинальностью* массива). В стандарте SQL:1999 многомерные массивы и массивы массивов не поддерживались. Однако в стандарте SQL:2003 это ограничение было снято, и теперь типом элементов любого типа коллекций может быть любой допустимый в SQL тип данных, кроме самого конструируемого типа коллекции.

Элементам каждого значения типа массива соответствуют их порядковые номера, называемые индексами. Значение индекса всегда должно принадлежать отрезку `[1, mc]`. Значениями типа массива `dt ARRAY [mc]` являются все массивы, состоящие из элементов типа `dt`, максимальное значение индекса которых `cs` не превосходит значения `mc`. При сохранении в базе данных значения типа массива занимает столько памяти, сколько требуется для сохранения `cs` элементов. Обеспечивается доступ к элементам массива по их индексам. В частности, можно объявить столбец типа `INTEGER ARRAY [10]` и при вставке строки в соответствующую таблицу задать значение только пятого элемента массива. Тогда в строку будет занесен массив из пяти элементов, причем первые четыре элемента будут содержать неопределенное значение (`NULL`).

Основными операциями над массивами являются выборка значения элемента массива по его индексу, изменение некоторого элемента массива или массива целиком и конкатенация (сцепление) двух массивов. Кроме того, для любого значения типа массива можно узнать значение его CS.

Типы мультимножеств

При определении столбца таблицы типа мультимножеств используется конструкция dt MULTISET, где dt задает тип данных элементов конструируемого типа мультимножеств. Значениями типа мультимножеств являются мультимножества, т. е. неупорядоченные коллекции элементов одного и того же типа, среди которых допускаются дубликаты. Например, значениями типа INTEGER MULTISET являются мультимножества, элементами которых —целые числа. Примером такого значения может быть мультимножество {12, 34, 12, 45, -64}.

В отличие от массива, мультимножество является неограниченной коллекцией; при конструировании типа мультимножеств не указывается предельная кардинальность значений этого типа. Однако это не означает, что возможность вставки элементов в мультимножество действительно не ограничена; стандарт всего лишь не требует явного объявления границы. Ситуация аналогична той, которая возникает при работе с таблицами, для которых в SQL не объявляется максимально допустимое число строк.*

Для типов мультимножеств поддерживаются операции преобразования типа значения-мультимножества к типу массивов или другому типу мультимножеств с совместимым типом элементов (операция CAST), для удаления дубликатов из мультимножества (функция SET), для определения числа элементов в заданном мультимножестве (функция CARDINALITY), для выборки элемента мультимножества, содержащего в точности один элемент (функция ELEMENT). Кроме того, для мультимножеств обеспечиваются операции объединения (MULTISET UNION), пересечения (MULTISET INTERSECT) и определения разности (MULTISET EXCEPT). Каждая из операций может выполняться в режиме с сохранением дубликатов (режим ALL) или с устранением дубликатов (режим DISTINCT).

Расширенные в SQL:2003 возможности работы с типами коллекций являются принципиально важными. Даже при наличии определяемых пользователями типов данных (см. ниже) и типов массивов SQL:1999 не предоставлял полных возможностей для преодоления исторически присущего реляционной модели данных вообще и SQL в частности ограничения «плоских таблиц». После появления конструктора типов мультимножеств и устранения ограничений на тип данных элементов коллекции это историческое ограничение полностью ликвидировано. Мультимно-

* Конечно, на практике такие ограничения устанавливаются в документации конкретной используемой СУБД, либо даже администратором конкретной базы данных.

жество, типом элементов которого является анонимный строчный тип (см. ниже), представляет собой полный аналог таблицы. Тем самым, в базе данных допускается произвольная вложенность таблиц. Возможности выбора структуры базы данных безгранично расширяются.

Анонимные строчные типы

Анонимный строчный тип* – это конструктор типов `ROW`, позволяющий производить безымянные типы строк (кортежей). Любой возможный строчный тип получается путем использования конструктора `ROW`. При определении столбца, значения которого должны принадлежать некоторому строчному типу, используется конструкция `ROW (fld1, fld2, …, fldn )`, где каждый элемент `fldi`, определяющий поле строчного типа, задается в виде тройки `fldname`, `fldtype`, `fldoptions`. Подэлемент `fldname` задает имя соответствующего поля строчного типа. Подэлемент `fldtype` специфицирует тип данных этого поля. В качестве типа данных поля строчного типа можно использовать любой допустимый в SQL тип данных, включая типы коллекций, определяемые пользователями типы и другие строчные типы. Необязательный подэлемент `fldoptions` может задаваться для указания применяемого по умолчанию порядка сортировки, если соответствующий подэлемент `fldtype` указывает на тип символьных строк, а также должен задаваться, если `fldtype` указывает на ссылочный тип (см. ниже). Степенью строчного типа называется число его полей.

Типы, определяемые пользователем

Эта категория типов данных связана с объектными расширениями языка SQL. Более подробно мы обсудим эту тему в заключительной лекции курса, а здесь для полноты картины приведем беглый набросок.

- Структурные типы (*Structured Types*). Соответствующие возможности SQL:1999 позволяют определять долговременно хранимые, именованные типы данных**, включающие один или более атрибутов любого из допустимых в SQL типа данных***, в том числе другие структурные типы, типы коллекций, строчные типы и т. д. Стандарт SQL не накладывает ограничений на сложность получаемой в результате структуры данных, однако не запрещает устанавливать такие ограничения в реа-

* В тексте стандарта SQL:1999 используется термин *anonymous row type*. Следуя соглашениям предыдущего пункта, мы должны были бы использовать термин *анонимные типы строк*. Но тогда уж точно возникла бы путаница с типами символьных строк. Конечно, можно было бы радикально отказаться от использования термина *строка таблицы* и вернуться к *кортежам отношений*. Но, к сожалению, этого сделать нельзя, поскольку в SQL таблицы – это не совсем (а иногда и совсем не) отношения, а строки таблиц – не совсем (*совсем не*) кортежи.

** Соответствующие определения сохраняются как часть метаданных базы данных (другими словами, являются частью схемы базы данных).

*** Требуется, чтобы в определении структурного типа `A` использовались только те типы, которые были определены ранее.

лизации. Дополнительные механизмы определяемых пользователями методов, функций и процедур позволяют определить поведенческие аспекты структурного типа.

- Индивидуальные типы (*Distinct Types*). Можно определить долговременно хранимый, именованный тип данных, опираясь на единственный предопределенный тип. Например, можно определить индивидуальный тип данных `PRICE`, опираясь на тип `DECIMAL (5, 2)`. Тогда значения типа `PRICE` представляются точно так же, как значения типа `DECIMAL (5, 2)`. Однако в SQL:1999 индивидуальный тип не наследует от своего опорного типа набор операций над значениями. Например, чтобы сложить два значения типа `PRICE` требуется явно сообщить системе, что с этими значениями нужно обращаться как со значениями типа `DECIMAL (5, 2)`. Другая возможность состоит в явном определении методов, функций и процедур, связанных с данным индивидуальным типом. Похоже, что в будущих версиях стандарта появятся и другие, более удобные возможности.

Ссылочные типы

Эта категория типов данных связана с объектными расширениями языка SQL, и мы снова отложим подробное обсуждение этого механизма до заключительной лекции курса и рассмотрим его здесь очень коротко. Обеспечивается механизм конструирования типов (*ссылочных типов*), которые могут использоваться в качестве типов столбцов некоторого вида таблиц (*типизированных таблиц*). Фактически значениями ссылочного типа являются строки соответствующей типизированной таблицы. Более точно, каждой строке типизированной таблицы приписывается уникальное значение (нечто вроде первичного ключа, назначаемого системой или приложением), которое может использоваться в методах, определенных для табличного типа, для уникальной идентификации строк соответствующей таблицы. Эти уникальные значения называются *ссылочными значениями*, а их тип – ссылочным типом. Ссылочный тип может содержать только те значения, которые действительно ссылаются на экземпляры указанного типа (т. е. на строки соответствующей типизированной таблицы).

Средства определения, изменения определения и отмены определения доменов

Как неоднократно упоминалось выше, при определении столбцов таблицы требуется явно указывать тип данных каждого столбца. Для этого можно использовать описанные выше средства спецификации типа. Но в SQL поддерживается и другой механизм— механизм доменов. Домен является долговременно хранимым, именованным объектом схемы базы

данных. Домены можно создавать (определять), изменять (изменять определения) и ликвидировать (отменять определение). Имена доменов можно использовать при определении столбцов таблиц. Можно считать, что в SQL определение домена представляет собой вынесенное за пределы определения индивидуальной таблицы «родовое» определение столбца, которое можно использовать для определения различных реальных столбцов реальных базовых таблиц. В языке SQL обеспечиваются средства определения доменов, изменения и отмены существующих определений.

Определение домена

Для определения домена в SQL используется оператор `CREATE DOMAIN`. Общий синтаксис этого оператора следующий:*

```
domain_definition ::= CREATE DOMAIN domain_name [AS] data_type
                          [ default_definition ]
                          [ domain_constraint_definition_list ]
```

Здесь `domain_name` задает имя создаваемого домена**, `data_type` есть спецификация определяющего типа данных. В необязательных разделах `default_definition` и `domain_constraint_definition_list` специфицируются значение домена по умолчанию*** и набор ограничений целостности, которые будут применяться к любому столбцу, определенному на этом домене.

Раздел `default_definition` имеет вид

```
DEFAULT { literal | niladic_function | NULL }****
```

Здесь `literal` представляет любое допустимое литеральное значение определяющего типа домена, `NULL` обозначает неопределенное значение, а `niladic_function` может задаваться в одной из следующих форм:

* Начиная с этого места мы будем приводить более или менее точный синтаксис конструкций языка SQL (не злоупотребляя излишествами). Без этого текст был бы менее точным и более объемным. Прописными буквами показываются «терминалы» – ключевые слова языка SQL.

** Здесь мы в первый раз сталкиваемся с именем объекта базы данных. Не будем углубляться в детали, но в общем случае имена объектов SQL-ориентированных баз данных имеют вид `имя_каталога.имя_схемы.имя_объекта`. Этот подход к именованию объектов базы данных позволяет независимо создавать объекты в разных схемах, не заботясь о том, чтобы эти объекты имели разные простые имена. При использовании в операторе SQL простого имени объекта система должна автоматически уточнить это имя, исходя из идентификатора пользователя, от имени которого выполняется оператор.

*** Это значение будет использоваться в качестве значения по умолчанию для любого столбца, определенного на данном домене, для которого не определено собственное значение по умолчанию (см. следующую лекцию).

**** { $element_1$ | $element_2$ |...| $element_n$ } означает, что в данной синтаксической конструкции должен присутствовать один и только один $element_i$.

```
USER
CURRENT_USER
SESSION_USER
SYSTEM_USER
CURRENT_DATE
CURRENT_TIME
CURRENT_TIMESTAMP*
```

Если в операторе `CREATE DOMAIN` значение по умолчанию не специфицируется, считается, что такого значения нет. Однако позже к определению домена можно добавить раздел значения по умолчанию с помощью оператора `ALTER DOMAIN`. Кроме того, этот оператор позволяет удалить раздел значения по умолчанию из существующего определения домена.

Элемент списка `domain_constraint_definition_list` имеет вид

```
[CONSTRAINT constraint_name] CHECK (conditional_expression)
```

Необязательный раздел `CONSTRAINT constraint_name` позволяет определить имя нового ограничения целостности. Если явное указание имени отсутствует, ограничению назначается имя, автоматически генерируемое системой. Что касается вида условного выражения, служащего собственно ограничением целостности, то в стандарте запрещается лишь прямое или косвенное использование в нем домена, в определение которого входит данное условное выражение.** Однако наиболее естественным (и наиболее распространенным) видом ограничения домена является следующий:

```
CHECK (VALUE IN (list_of_valid_values))
```

Такое ограничение запрещает появление в любом столбце, определенном на данном домене, любого значения определяющего типа, не входящего в список допустимых значений.

Примеры определений доменов

В дальнейших примерах нам понадобятся определения нескольких доменов. Приведем их в этом подразделе. В примерах мы будем иметь де-

* Значение `niladic_function` «вычисляется» в тот момент, когда требуется значение по умолчанию (обычно при вставке в таблицу новой строки, значение соответствующего столбца которой явно не указано). Смысл `CURRENT_DATE`, `CURRENT_TIME` и `CURRENT_TIMESTAMP` очевиден. `USER` (или, что то же, `CURRENT USER`), `SESSION_USER` и `SYSTEM_USER` задают идентификатор пользователя, от имени которого выполняется текущая транзакция, текущая сессия, и идентификатор операционной системы, в которой работает пользователь, соответственно. В стандарте не определяется представление этих идентификаторов, но в реализациях они обычно представляются в виде символьных строк.

** Более подробно мы обсудим допустимые в SQL виды условных выражений в следующих лекциях.

ло с таблицами служащих (EMP), отделов (DEPT) и проектов (PRO). Каждый служащий обладает уникальным номером (EMP_NO) и получает заработную плату (SALARY). Определим домены EMP_NO и SALARY.

```
CREATE DOMAIN EMP_NO AS INTEGER
     CHECK (VALUE BETWEEN 1 AND 10000);
```

Номера служащих являются целыми числами, поэтому базовый тип домена EMP_NO есть тип INTEGER. Кроме того, на значения этого домена устанавливается следующее ограничение: они должны быть больше нуля и не превосходить целое значение 10000.

Домен SALARY определим следующим образом:

```
CREATE DOMAIN SALARY AS NUMERIC (10, 2)
     DEFAULT 10000.00
     CHECK (VALUE BETWEEN 10000.00 AND 20000000.00)
     CONSTRAINT SAL_NOT_NULL CHECK (VALUE IS NOT NULL);
```

Размер заработной платы является значением точного числового типа NUMERIC из десяти десятичных цифр, две из которых составляют дробную часть. По умолчанию размер заработной платы составляет 10000 руб. Установлен диапазон допустимого размера зарплаты от 10000 руб. до 20000000 руб. Неопределенное значение зарплаты не допускается (на уровне определения домена).

Изменение определения домена

Для изменения характеристик ранее определенного домена используется оператор SQL ALTER DOMAIN. Синтаксис этого оператора выглядит следующим образом:

```
domain_alternation ::=
        ALTER DOMAIN domain_name domain_alternation_action
domain_alternation_action ::=
        domain_default_alternation_action
      | domain_constraint_alternation_action
```

Как видно из синтаксических правил, при изменении определения домена можно выполнить действие по изменению раздела значения по умолчанию либо изменить ограничение домена. Для первого варианта действует следующий синтаксис:

```
domain_default_alternation_action ::=
        SET default_definition
      | DROP DEFAULT
```

В случае установки нового значения по умолчанию (SET) это значение автоматически применяется ко всем столбцам, определенным на данном домене. Более точно, это значение становится новым значением по умолчанию. Операция не оказывает влияния на состояние существующих строк таблиц базы данных. В случае отмены раздела значения по умолчанию в определении домена (DROP) существовавшее значение домена по умолчанию становится значением по умолчанию каждого столбца, который определен на данном домене и для которого не специфицировано собственное значение по умолчанию.

Действие по изменению ограничения домена определяется следующим синтаксисом:

```
domain_constraint_alternation_action ::=
        ADD domain_constraint_definition
      | DROP CONSTRAINT constraint_name
```

Действие по добавлению нового определения ограничения домена (ADD) приводит к тому, что новое условие добавляется через AND к существующему ограничению домена. Если к моменту выполнения соответствующего оператора ALTER DOMAIN существуют столбцы некоторых таблиц, текущие значения которых противоречат новому ограничению, то СУБД должна отвергнуть этот оператор ALTER DOMAIN. Действие по отмене ограничения домена (DROP) приводит к исчезновению соответствующей части общего ограничения соответствующего домена, что, естественно, не влияет на существующие значения столбцов имеющихся таблиц.

Примеры изменения определения домена

Немного поупражняемся с доменом SALARY. Для изменения значения заработной платы по умолчанию с 10000 на 11000 руб. нужно выполнить оператор

```
ALTER DOMAIN SALARY SET DEFAUL 11000.00;
```

Для отмены значения по умолчанию в домене SALARY следует воспользоваться оператором

```
ALTER DOMAIN SALARY DROP DEFAUL;
```

Если к определению домена SALARY требуется добавить ограничение (например, запретить значение зарплаты, равное 15000 руб.), необходимо выполнить оператор

```
ALTER DOMAIN SALARY ADD CHECK (VALUE <> 15000.00);
```

Наконец, если требуется отменить (именованное!) ограничение целостности, препятствующее наличию неопределенных значений в столбцах, которые определены на домене SALARY, то нужно выполнить оператор

```
ALTER DOMAIN SALARY DROP CONSTRAINT SAL_NOT_NULL;
```

Отмена определения домена

Чтобы отменить ранее созданное определение домена, нужно воспользоваться оператором DROP DOMAIN в следующем синтаксисе:

```
DROP DOMAIN domain_name {RESTRICT | CASCADES}
```

Если в операторе указано RESTRICT, и если соответствующий домен использован в определении некоторого столбца, в определении некоторого представления или в определении ограничения целостности (см. следующие лекции), то оператор DROP DOMAIN отвергается. В противном случае определение домена ликвидируется.

Если в операторе DROP DOMAIN указано CASCADES, то оператор выполняется всегда. При этом уничтожаются все представления и ограничения целостности, в определении которых использовалось имя данного домена. Столбцы, определенные на этом домене, автоматически переопределяются следующим образом:

- считается, что каждый такой столбец теперь относится к определяющему типу уничтожаемого домена;
- если у столбца не было определено собственное значение по умолчанию, то считается, что теперь у него имеется такое значение по умолчанию, совпадающее со значением по умолчанию уничтожаемого домена;
- каждый столбец наследует все ограничения уничтожаемого домена.

Неявные и явные преобразования типа или домена

В языке SQL обеспечивается возможность использования в различных операциях не только значений тех типов, для которых предопределена операция, но и значений типов, неявным или явным образом приводимых к требуемому типу.

Неявные преобразования типов в SQL

В SQL поддерживается совместимость некоторых типов данных за счет неявного преобразования значений одного типа к значениям другого типа данных (например, при необходимости FLOAT неявно приводится к DOUBLE). Опишем наиболее важные правила совместимости типов, принятые в SQL:1999. Начнем с определения приводимости типов. Тип данных *A* приводим к типу данных *B* в том и только в том случае, когда в любом месте, где ожидается значение типа *B*, может быть использовано значение типа *A*.

Основные правила приводимости типов состоят в следующем.

- *Типы символьных строк.* Тип CHARACTER (x) приводим к любому типу CHARACTER (y), если y≥x. Типы VARCHAR (x) и CHARACTER (x) приводимы к любому типу VARCHAR (y), если y≥x. Типы CHARACTER (x) и VARCHAR (x) приводимы к любому типу CLOB.
- *Типы битовых строк.* Тип BIT (x) приводим к любому типу BIT (y), если y≥x. Типы BIT VARYING (x) и BIT (x) приводимы к любому типу BIT VARYING (y), если y≥x.
- *Типы* BLOB. Тип BLOB (x) приводим к любому типу BLOB (y), если y≥x.
- Типы точных чисел. Тип EN (p_1, s_1) приводим к любому типу EN (p_2, s_2), у которого $s_2 \geq s_1$ и p_2 определяется в реализации. Тип EN (p, s) приводим к любому типу приблизительных чисел AN (p_1), где p_1 определяется в реализации.
- *Типы приблизительных чисел.* Тип AN (p_1) приводим к любому типу (p_2), если $p_2 \geq p_1$.

Явные преобразования типов или доменов и оператор CAST

Неявные преобразования типов не всегда удобны, недостаточно гибки и иногда могут вызывать ошибки. Поэтому, как показывает предыдущий подраздел, число допустимых неявных преобразований типов в SQL весьма ограничено. Однако SQL существует специальный оператор CAST, с помощью которого можно явно преобразовывать типы или домены в более широких пределах допускаемых преобразований. Конструкция имеет следующий синтаксис:

```
CAST ({scalar-expression | NULL } AS {data_type | domain_name})
```

Оператор преобразует значение заданного скалярного выражения к указанному типу или к базовому типу указанного домена. Результатом применения оператора CAST к неопределенному значению является неопределенное значение. Для значений, отличных от неопределенных, в стандарте приводятся подробные правила выполнения преобразований, которые интуитивно понятны.

Поясним действие оператора CAST в наиболее важных случаях. Примем следующие обозначения типов данных:

EN – точные числовые типы (Exact Numeric)
AN – приблизительные числовые типы (Approximate Numeric)
C – типы символьных строк (Character)
FC – типы символьных строк постоянной длины (Fixed-length Character)
VC – типы символьных строк переменной длины (Variable-length Character)
B – типы битовых строк (Bit String)
FB – типы битовых строк постоянной длины (Fixed-length Bit String)
VB – типы битовых строк переменной длины (Variable-length Bit String)
D – тип Date
T – типы Time
TS – типы Timestamp
YM – типы Interval Year-Month
DT – типы Interval Day-Time

Пусть TD – это тип данных, к которому производится преобразование, а SD – тип данных операнда. Тогда допустимы следующие комбинации («да» означает безусловную допустимость, «нет» – безусловную недопустимость и «?» – допустимость с оговорками).

		TD										
		EN	**AN**	**VC**	**FC**	**VB**	**FB**	**D**	**T**	**TS**	**YM**	**DT**
SD	**EN**	Да	Да	Да	Да	Нет	Нет	Нет	Нет	Нет	?	?
	AN	Да	Да	Да	Да	Нет	Нет	Нет	Нет	Нет	Нет	Нет
	C	Да	Да	?	?	Да	Да	Да	Да	Да	Да	Да
	B	Нет	Нет	Да	Да	Да	Да	Нет	Нет	Нет	Нет	Нет
	D	Нет	Нет	Да	Да	Нет	Нет	Да	Нет	Да	Нет	Нет
	T	Нет	Нет	Да	Да	Нет	Нет	Нет	Да	Да	Нет	Нет
	TS	Нет	Нет	Да	Да	Нет	Нет	Да	Да	Да	Нет	Нет
	YM	?	Нет	Да	Да	Нет	Нет	Нет	Нет	Нет	Да	Нет
	DT	?	Нет	Да	Да	Нет	Нет	Нет	Нет	Нет	Нет	Да

По поводу ячеек таблицы, содержащих знак вопроса, необходимо сделать несколько оговорок:

(1) если TD – интервал и SD – тип точных чисел, то TD должен содержать единственное поле даты-времени;

(2) если TD – тип точных чисел и SD – интервал, то SD должен содержать единственное поле даты-времени;
(3) если SD – тип символьных строк и TD – тип символьных строк постоянной или переменной длины, то набор символов SD и TD должен быть одним и тем же.

Заключение

В этой лекции мы начали рассматривать средства языка SQL, позволяющие определять и динамически изменять схему базы данных. Наиболее важным для общего понимания языка является раздел «Типы данных SQL» – система типов языка SQL (и любой SQL-ориентированной базы данных). В последних стандартах языка SQL поддерживаются:

- развитый набор предопределенных типов, включая ряд параметризованных типов;
- генераторы типов массивов и мультимножеств, элементами которых могут быть значения предопределенных типов, типов коллекций, анонимных строчных типов строк и типов, определенных пользователями;
- генератор анонимных строчных типов, в которых типом элемента строки может быть любой предопределенный тип, тип коллекции, анонимный строчный тип и тип, определенный пользователями;
- определяемый пользователем структурный тип, в котором типом элемента структуры может быть любой предопределенный тип, тип коллекции, анонимный строчный тип и тип, определенный пользователями; для определяемых пользователем структурных и индивидуальных типов можно определять пользовательские операции.

Нельзя с уверенностью сказать, что система типов языка SQL настолько полна, что может удовлетворить любые потребности, но можно отметить, что в этой системе типов отсутствует единый логический подход и имеется избыточность. Возможно, это станет понятнее после обсуждения в конце курса средств объектно-реляционных расширений языка SQL.

Как должно быть ясно из этой лекции, механизм доменов в SQL играет вспомогательную роль. Это не совсем те (может быть, и совсем не те) домены, поддержка которых предполагается реляционной моделью. Фактически определение домена обеспечивает спецификацию ограничений и значений по умолчанию, выносимых за пределы определения столбца. В комитете по стандартизации SQL обсуждается идея полного отказа от поддержки механизма доменов и замены его на соответствующим образом адаптированный механизм индивидуальных типов (см. последнюю лекцию курса).

Лекция 12. Язык баз данных SQL: средства определения базовых таблиц и ограничений целостности

Лекция посвящена средствам языка SQL, позволяющим определять (создавать) базовые таблицы, изменять определения базовых таблиц и отменять их. Поскольку важными составляющими определения базовой таблицы являются определения ограничений на уровнях столбцов и таблицы целиком, мы сочли уместным включить в эту же лекцию материал, посвященный средствам определения ограничений целостности общего вида (не привязанных к определениям базовых таблиц), изменения и отмены таких определений.

Ключевые слова: базовая таблица, мультимножество строк, операторы `CREATE TABLE`, `ALTER TABLE` и `DROP TABLE`, определение столбца, значения столбца по умолчанию, ограничения целостности столбца, ограничение `NOT NULL`, ограничение первичного ключа, ограничение возможного ключа, определение внешнего ключа на уровне столбца, проверочное ограничение на уровне столбца, определение табличного ограничения, табличное ограничение первичного или возможного ключа, проверочное табличное ограничение, табличное ограничение внешнего ключа, способы сопоставления значений внешнего и возможного ключей, ссылочные действия, добавление, изменение или удаление определения столбца, изменение набора табличных ограничений, общие ограничения целостности, операторы `CREATE ASSERTION` и `DROP ASSERTION`, немедленная и откладываемая проверка ограничений, оператор `SET CONSTRAINTS`.

Введение

Как мы уже отмечали в лекции 11, к спецификации языка SQL можно относиться как к спецификации некоторой модели данных, в определенных аспектах близкой к реляционной модели. Мы стремимся к тому, чтобы порядок лекций, посвященных языку SQL, способствовал правильному пониманию именно этой модели, а не технических тонкостей языка. Предыдущая лекция посвящалась тому, *что* (т. е. данные каких типов) может храниться в SQL-ориентированной базе данных.

Теперь следует понять, *где* хранятся эти данные. Как и в реляционной модели данных, в модели SQL поддерживается единственная родовая структура данных, называемая в данном случае *базовой таблицей*. В первом из двух основных разделов лекции обсуждаются средства языка SQL, предназначенные для определения, изменения определения и отмены определения базовых таблиц.

Понятие базовой таблицы родственно понятию отношения: можно считать, что базовая таблица обладает заголовком*, в котором содержатся различаемые имена столбцов и их типы данных (заголовок базовой таблицы является множеством и представляет собой близкий аналог заголовка отношения), и телом, включающим строки, которые соответствуют заголовку таблицы (казалось бы, здесь мы имеем аналоги тела отношения и кортежей). Но коренное отличие базовой таблицы от истинного отношения состоит в том, что тело таблицы не обязательно является *множеством*. Среди строк тела таблицы могут встречаться дубликаты, и в общем случае тело базовой таблицы SQL представляет собой *мультимножество* строк.

Забегая вперед (см. следующие лекции), следует заметить, что *порождаемые таблицы* SQL, которые формируются при выполнении запросов к SQL-ориентированной базе данных, еще более отдаляют SQL от реляционной модели. В таких таблицах может отсутствовать и правильно сформированный заголовок (могут иметься одноименные столбцы).

Почему же, понимая принципиальные отклонения языка SQL от реляционной модели данных, мы включили эти две темы в один курс и, более того, иногда неформально называем SQL языком *реляционных баз данных*? Тому есть несколько причин.

- Во-первых, используя язык SQL, можно не нарушать предписаний реляционной модели, и тогда к «правильно построенной» SQL-ориентированной базе данных применимы все фундаментальные результаты теории реляционных баз данных, включая принципы проектирования на основе нормализации.
- Во-вторых, полностью отвергая родство языка SQL с реляционной моделью данных, мы выступали бы против установившихся исторических традиций. Этот язык возник около 30 лет тому назад во время реализации в компании IBM проекта по созданию экспериментальной СУБД System R, основной целью которого являлось обоснование практической реализации реляционного подхода к организации баз данных. Так что исторически SQL базировался на реляционной модели данных (возможно, не совсем верно понятой и/или воплощенной).
- Наконец, по нашему мнению, в области информационной технологии любой практически используемый инструмент не может быть полностью свободен от компромиссов. Идеологически чистые решения возможны только в научно-экспериментальной работе. «Великий и ужасный» язык SQL – это порождение ряда компромиссов между теорией, практикой и маркетинговой деятельностью. Этот язык является настолько реляционным, насколько это понадобилось потребителям коммерческих СУБД, прямо или косвенно финансировавшим разработку языка.

* В SQL не используются термины *заголовок* и *тело* таблицы. Здесь мы временно пользуемся этой терминологией только для целей сопоставления модели SQL с реляционной моделью данных.

В операторе SQL CREATE TABLE специфицируются не только столбцы таблицы, но и ограничения целостности, которым должны удовлетворять данные, хранящиеся в базовой таблице. Эти ограничения являются частным случаем ограничений базы данных целиком, для определения которых, а также изменения и ликвидации определений имеются специальные операторы. Обсуждению этих средств посвящен второй основной раздел этой лекции.

Средства определения, изменения и ликвидации базовых таблиц

Базовые (реально хранимые в базе данных) таблицы создаются (определяются) с использованием оператора CREATE TABLE. Для изменения определения базовой таблицы применяется оператор ALTER TABLE. Уничтожить хранимую таблицу (отменить ее определение) можно с помощью оператора DROP TABLE.

Замечание: хотя внешне операторы CREATE TABLE, ALTER TABLE и DROP TABLE похожи на соответствующие операторы определения, изменения определения и отмены определения домена, между ними имеется принципиальное различие. Определение домена приводит всего лишь к созданию некоторых новых описателей, входящих в состав метаданных базы данных. Создание базовой таблицы, кроме создания соответствующих описателей, порождает новую область внешней памяти, в которой будут храниться данные, поставляемые пользователями. Тем самым, базовая таблица SQL-ориентированной базы данных является прямым аналогом переменной отношения реляционной модели данных.

Определение базовой таблицы

Оператор создания базовой таблицы CREATE TABLE имеет следующий синтаксис:

```
base_table_definition ::= CREATE TABLE base_table_name
                          (base_table_element_commalist)*
base_table_element ::= column_definition
                     | base_table_constraint_definition
```

Здесь base_table_name задает имя новой (изначально пустой) базовой таблицы. Каждый элемент определения базовой таблицы является либо определением столбца, либо определением табличного ограничения целостности.

* В круглых скобках указывается список определений элементов базовой таблицы (должно присутствовать определение хотя бы одного столбца), разделенных запятыми.

Определение столбца

Элемент определения столбца специфицируется на основе следующих синтаксических правил:

```
column_definition ::= column_name { data_type | domain_name }
                      [ default_definition ]
                      [ column_constraint_definition_list ]
```

В элементе определения столбца column_name задает имя определяемого столбца. Тип столбца специфицируется путем явного указания типа данных (data_type) или путем указания имени ранее определенного домена (domain_name).

Значения столбца по умолчанию

Необязательный раздел определения значения столбца по умолчанию имеет тот же синтаксис, что и раздел определения значения по умолчанию в операторах определения или изменения определения домена:

```
DEFAULT { literal | niladic_function | NULL }
```

Действующее значение по умолчанию для данного столбца определяется следующим образом:

- если в определении столбца явно присутствует раздел DEFAULT, то значением столбца по умолчанию является значение, указанное в этом разделе;
- иначе, если столбец определяется на домене и в определении этого домена явно присутствует раздел DEFAULT, то значением столбца по умолчанию является значение, указанное в этом разделе;
- иначе значением по умолчанию столбца является NULL.*

Заметим, что если значением по умолчанию неявно объявлено неопределенное значение (NULL), но среди ограничений целостности столбца присутствует ограничение NOT NULL (см. ниже), то считается, что у столбца вообще отсутствует значение по умолчанию. Это означает, что при любой вставке новой строки в соответствующую базовую таблицу значение данного столбца должно быть задано явно.

Ограничения целостности столбца

Элемент необязательного списка ограничений целостности столбца определяется следующими синтаксическими правилами:

* Заметим, что хотя столбец может получить значение NULL по умолчанию как явным, так и неявным образом, эти два случая не являются эквивалентными. Явное задание NULL в качестве значения по умолчанию запрещает наследование столбцом значения по умолчанию домена.

```
column_constraint_definition ::= [CONSTRAINT constraint_name ]
                                   NOT NULL
                                 | { PRIMARY KEY | UNIQUE }
                                 | references_definition
                                 | CHECK ( conditional_expression )
```

Как мы увидим немного позже, любое ограничение целостности, включаемое в определение столбца, может быть эквивалентным образом выражено в виде табличного ограничения целостности. Единственный резон определения ограничений на уровне столбца состоит в том, что в этом случае в ограничении целостности не требуется явно указывать имя столбца. Тем не менее, кратко рассмотрим ограничения целостности столбца отдельно.

Ограничение `NOT NULL` означает, что в определяемом столбце никогда не должны содержаться неопределенные значения. Если определяемый столбец имеет имя `C`, то это ограничение эквивалентно следующему табличному ограничению: `CHECK (C IS NOT NULL)`.

В определение столбца может входить одно из ограничений: ограничение первичного ключа (`PRIMARY KEY`) или ограничение возможного ключа (`UNIQUE`). Включение в определение столбца любого из этих ограничений означает требование уникальности значений определяемого столбца (т. е. во все время существования определяемой таблицы во всех ее строках значения данного столбца должны быть различны*). Ограничение `PRIMARY KEY`, в дополнение к этому, влечет за собой ограничение `NOT NULL` для определяемого столбца. Эти ограничения столбца эквивалентны следующим табличным ограничениям: `PRIMARY KEY (C)` и `UNIQUE (C)`.

Ограничение `references_definition` означает объявление определяемого столбца внешним ключом таблицы и обладает следующим синтаксисом:

```
references_definition ::=
     REFERENCES base_table_name [ (column_commalist) ]
        [ MATCH { SIMPLE | FULL | PARTIAL } ]
        [ ON DELETE referential_action ]
        [ ON UPDATE referential_action ]
```

На самом деле, данная синтаксическая конструкция работает и в случае определения внешнего ключа на уровне таблицы (в одном из определений табличных ограничений целостности). Поэтому мы отложим обсуждение до рассмотрения этого общего случая. Пока отметим только, что при использовании конструкции на уровне определения столбца

* В этом случае SQL опирается на семантику неопределенных значений, отличную от используемой в большинстве других случаев. Считается, что `(NULL = NULL) = true` и что `(a = NULL) = (NULL = a) = false` для любого значения `a`, отличного от `NULL`.

column_commalist может содержать имя только одного столбца (потому что внешний ключ состоит из одного определяемого столбца). Ограничение эквивалентно следующему табличному ограничению: FOREGN KEY (C) references_definition.

Проверочное ограничение CHECK (conditional_expression) приводит к тому, что в данном столбце могут находиться только те значения, для которых вычисление conditional_expression не приводит к результату *false*. В условном выражении проверочного ограничения столбца разрешается использовать имя только определяемого столбца. Заметим, что проверочное ограничение столбца может быть безо всяких изменений перенесено на уровень определения табличных ограничений.

Определение табличного ограничения

Элемент определения табличного ограничения целостности задается в следующем синтаксисе:

```
base_table_constraint_definition ::= [ CONSTRAINT constraint_name ]
      { PRIMARY KEY | UNIQUE } ( column_commalist )
      | FOREIGN KEY ( column_commalist ) references_definition
      | CHECK ( conditional_expression )
```

Как мы видим, имеется три разновидности табличных ограничений: ограничение первичного или возможного ключа (PRIMARY KEY или UNIQUE), ограничение внешнего ключа (FOREGN KEY) и проверочное ограничение (CHECK). Любому ограничению может явным образом назначаться имя, если перед определением ограничения поместить конструкцию CONSTRAINT constraint_name.

Табличное ограничение первичного или возможного ключа

Табличное ограничение первичного или возможного ключа { PRIMARY_KEY | UNIQUE } (column_commalist) означает требование уникальности составных значений указанной группы столбцов (т. е. во все время существования определяемой таблицы во всех ее строках составные значения данной группы столбцов должны быть различны*). Ограничение PRIMARY KEY, в дополнение к этому, влечет ограничение NOT NULL для всех столбцов, упоминаемых в определении ограничения. В определении таблицы допускается произвольное число определений возможного ключа (для разных комбинаций столбцов), но не более од-

* С учетом замечания по поводу особого толкования семантики неопределенных значений, сделанного в предыдущей сноске.

ного определения первичного ключа. Обратите особое внимание на последнюю часть предыдущего предложения: в языке SQL действительно допускается определение таблиц, у которых отсутствуют возможные ключи. Эта особенность языка, среди прочего, очевидным образом противоречит базовым требованиям реляционной модели данных.

Проверочное табличное ограничение

Определение табличного ограничения вида `CHECK (conditional_expression)` приводит к тому, что указанное условное выражение будет вычисляться при каждой попытке обновления соответствующей таблицы (вставке новой строки, удалении или модификации существующей строки). Считается, что попытка обновления таблицы нарушает проверочное ограничение целостности, если после выполнения операции обновления вычисление условного выражения дает результат *false*. Другими словами, таблица находится в соответствии с данным проверочным табличным ограничением, если для всех строк таблицы результатом вычисления соответствующего условного выражения не является *false*.

Мы отложим обсуждение допустимых разновидностей условных выражений до следующей лекции, где оно будет более уместно в контексте рассмотрения оператора `SELECT` языка SQL.

Табличное ограничение внешнего ключа

Синтаксис и семантика определения внешнего ключа в операторе SQL определения базовой таблицы являются довольно запутанными и сложными. По этой причине мы посвящаем этой языковой конструкции отдельный подраздел.

Табличное ограничение `FOREIGN KEY (column_commalist) references_definition` означает объявление внешним ключом группы столбцов, имена которых перечислены в списке `column_commalist`. Обсудим теперь смысл ограничения внешнего ключа при разных вариантах формирования определения ссылок (`references_definition`). Для удобства повторим синтаксическое правило.

```
references_definition ::=
     REFERENCES base_table_name [ (column_commalist) ]
          [ MATCH { SIMPLE | FULL | PARTIAL } ]
          [ ON DELETE referential_action ]
          [ ON UPDATE referential_action ]
```

В этом определении `base_table_name` должно представлять собой имя некоторой базовой таблицы (пусть, например, эта таблица имеет имя *T*). Ес-

ли определение ссылок включает список столбцов (column_commalist), то этот список должен совпадать (с точностью до порядка следования имен столбцов) со списком имен столбцов, использованных в некотором определении первичного или возможного ключа (PRIMARY_KEY или UNIQUE) в определении таблицы *T*. Если в определении ссылок список столбцов явно не задан, то считается, что он совпадает со списком столбцов, использованных в определении первичного ключа (PRIMARY_KEY) таблицы *T*.

Разновидности способов сопоставления значений внешнего и возможного ключей

Пусть определяемая таблица имеет имя *S*. Обсудим смысл необязательного раздела определения внешнего ключа MATCH { SIMPLE | FULL | PARTIAL }. Если этот раздел отсутствует или если присутствует и имеет вид MATCH SIMPLE, то ограничение внешнего ключа (ссылочное ограничение) удовлетворяется в том и только в том случае, когда для каждой строки таблицы *S* либо:

(a) *какой-либо* столбец, входящий в состав внешнего ключа, содержит NULL;

(b) таблица *T* содержит в *точности одну строку*, такую, что значение внешнего ключа в данной строке таблицы S совпадает со значением соответствующего возможного ключа в этой строке таблицы *T*.

Если раздел MATCH присутствует в определении внешнего ключа и имеет вид MATCH PARTIAL, то ограничение внешнего ключа удовлетворяется в том и только в том случае, когда для каждой строки таблицы *S* либо:

(a) *каждый столбец*, входящий в состав внешнего ключа, содержит NULL;

(b) таблица *T* содержит по *крайней мере одну* такую строку, что для каждого столбца данной строки таблицы *S*, *значение которого отлично от NULL*, его значение совпадает со значением соответствующего столбца возможного ключа в этой строке таблицы *T*.

Если раздел MATCH имеет вид MATCH FULL, то ограничение внешнего ключа удовлетворяется в том и только в том случае, когда для каждой строки таблицы S выполняется одно из следующих условий:

(a) *каждый столбец*, входящий в состав внешнего ключа, содержит NULL;

(b) *ни один* столбец, входящий в состав внешнего ключа, не содержит NULL, и таблица *T* содержит *в точности одну строку*, такую, что значение внешнего ключа в данной строке таблицы S совпадает со значением соответствующего возможного ключа в этой строке таблицы *T*.

Очевидно, что только при наличии спецификации MATCH FULL ссылочное ограничение соответствует требованиям реляционной модели. Тем

не менее в определении ограничения внешнего ключа базовых таблиц в SQL по умолчанию предполагается наличие спецификации MATCH SIMPLE.*

Поддержка ссылочной целостности и ссылочные действия

В связи с определением ограничения внешнего ключа нам осталось рассмотреть еще два необязательных раздела – ON DELETE referential_action и ON UPDATE referential_action. Прежде всего, приведем синтаксическое правило:

```
referential_action ::=
   { NO ACTION | RESTRICT | CASCADE | SET DEFAULT | SET NULL }
```

Чтобы объяснить, в каких случаях и каким образом выполняются эти действия, требуется сначала определить понятие *ссылающейся строки (referencing row)*. Если в определении ограничения внешнего ключа отсутствует раздел MATCH или присутствуют спецификации MATCH SIMPLE либо MATCH FULL, то для данной строки t таблицы T строкой таблицы S, ссылающейся на строку t, называется каждая строка таблицы S, значение внешнего ключа которой совпадает со значением соответствующего возможного ключа строки t. Если в определении ограничения внешнего ключа присутствует спецификация MATCH PARTIAL, то для данной строки t таблицы T строкой таблицы S, ссылающейся на строку t, называется каждая строка таблицы S, отличные от NULL значения столбцов внешнего ключа которой совпадают со значениями соответствующих столбцов соответствующего возможного ключа строки t. В случае MATCH PARTIAL строка таблицы S называется ссылающейся исключительно на строку t таблицы T, если эта строка таблицы S является ссылающейся на строку t и не является ссылающейся на какую-либо другую строку таблицы T.**

Теперь приступим к ссылочным действиям. Пусть определение ограничения внешнего ключа содержит раздел ON DELETE referential_action. Предположим, что предпринимается попытка удалить строку t из таблицы T. Тогда:

- если в качестве требуемого ссылочного действия указано NO ACTION или RESTRICT, то операция удаления отвергается, если ее выполнение вызвало бы нарушение ограничения внешнего ключа;
- если в качестве требуемого ссылочного действия указано CASCADE, то строка t удаляется, и если в определении ограничения внешнего ключа отсутствует раздел MATCH или присутствуют спецификации MATCH

* Если определяется внешний ключ, состоящий из одного столбца, то явное указание спецификации MATCH любой разновидности становится бессмысленным, поскольку в этом случае MATCH SIMPLE, MATCH PARTIAL и MATCH FULL ведут себя одинаково.

** Из приведенных ранее объяснений действия ограничения внешнего ключа при наличии в определении внешнего ключа раздела MATCH PARTIAL ясно следует, что в этом случае одна строка таблицы S может являться ссылающейся на несколько *разных строк* таблицы T.

SIMPLE или MATCH FULL, то удаляются все строки, ссылающиеся на *t*. Если же в определении ограничения внешнего ключа присутствует спецификация MATCH PARTIAL, то удаляются только те строки, которые ссылаются исключительно на строку *t*;

- если в качестве требуемого ссылочного действия указано SET DEFAULT, то строка *t* удаляется, и во всех столбцах, которые входят в состав внешнего ключа, всех строк, ссылающихся на строку *t*, проставляется заданное при их определении значение по умолчанию. Если в определении внешнего ключа содержится спецификация MATCH PARTIAL, то подобному воздействию подвергаются только те строки таблицы *S*, которые ссылаются исключительно на строку *t*;
- если в качестве требуемого ссылочного действия указано SET NULL, то строка *t* удаляется, и во всех столбцах, которые входят в состав внешнего ключа, всех строк, ссылающихся на строку *t*, проставляется NULL. Если в определении внешнего ключа содержится спецификация MATCH PARTIAL, то подобному воздействию подвергаются только те строки таблицы S, которые ссылаются исключительно на строку *t*.

Пусть определение ограничения внешнего ключа содержит раздел ON UPDATE referential_action. Предположим, что предпринимается попытка обновить столбцы соответствующего возможного ключа в строке *t* из таблицы *T*. Тогда:

- если в качестве требуемого ссылочного действия указано NO ACTION или RESTRICT, то операция обновления отвергается, если ее выполнение вызвало бы нарушение ограничения внешнего ключа;
- если в качестве требуемого ссылочного действия указано CASCADE, то строка *t* обновляется, и если в определении ограничения внешнего ключа отсутствует раздел MATCH или присутствуют спецификации MATCH SIMPLE или MATCH FULL, то соответствующим образом обновляются все строки, ссылающиеся на *t* (в них должным образом изменяются значения столбцов, входящих в состав внешнего ключа). Если же в определении ограничения внешнего ключа присутствует спецификация MATCH PARTIAL, то обновляются только те строки, которые ссылаются исключительно на строку *t*;
- если в качестве требуемого ссылочного действия указано SET DEFAULT, то строка *t* обновляется, и во всех столбцах, которые входят в состав внешнего ключа и соответствуют изменяемым столбцам таблицы *T*, всех строк, ссылающихся на строку *t*, проставляется заданное при их определении значение по умолчанию. Если в определении внешнего ключа содержится спецификация MATCH PARTIAL, то подобному воздействию подвергаются только те строки таблицы *S*, которые ссылаются исключительно на строку *t*, причем в них изменяются значения только тех столбцов, которые не содержали NULL;

- если в качестве требуемого ссылочного действия указано SET NULL, то строка t обновляется, и во всех столбцах, которые входят в состав внешнего ключа и соответствуют изменяемым столбцам таблицы T, всех строк, ссылающихся на строку t, проставляется NULL. Если в определении внешнего ключа содержится спецификация MATCH PARTIAL, то подобному воздействию подвергаются только те строки таблицы S, которые ссылаются исключительно на строку t.*

Примеры определений базовых таблиц

Определим таблицы служащих (EMP), отделов (DEPT) и проектов (PRO). Эти таблицы имеют заголовки, показанные на рис. 12.1.

Столбцы EMP_NO, EMP_SAL, DEPT_NO, PRO_NO, DEPT_TOTAL_SAL, DEPT_MNG и PRO_MNG определяются на ранее определенных доменах (определения доменов EMP_NO и SALARY приведены в предыдущей лекции). Первичными ключами отношений EMP, DEPT и проектов PRO являются столбцы EMP_NO, DEPT_NO и PRO_NO соответственно. В таблице EMP столбцы DEPT_NO и PRO_NO являются внешними ключами, указывающими на отдел, в котором работает служащий, и на выполняемый им проект соответственно. В таблице DEPT внешним ключом является столбец DEPT_NO, указывающий на служащего, являющегося руководителем соответствующего отдела, а в таблице PRO внешним ключом является столбец PRO_MNG, указывающий на служащего, являющегося менеджером соответствующего проекта. Другие ограничения целостности мы обсудим позже.

Определим таблицу EMP:

```
(1) CREATE TABLE EMP (
(2) EMP_NO EMP_NO PRIMARY KEY,
(3) EMP_NAME VARCHAR(20) DEFAULT 'Incognito' NOT NULL,
(4) EMP_BDATE DATE DEFAULT NULL CHECK (VALUE >= DATE '1917-10-24'),
(5) EMP_SAL SALARY,
(6) DEPT_NO DEPT_NO DEFAULT NULL REFERENCES DEPT ON DELETE SET NULL,
(7) PRO_NO PRO_NO DEFAULT NULL,
(8) FOREIGN KEY PRO_NO REFERENCES PRO (PRO_NO) ON DELETE SET NULL,
(9) CONSTRAINT PRO_EMP_NO CHECK
        ((SELECT COUNT (*) FROM EMP E
          WHERE E.PRO_NO = PRO_NO) <= 50));
```

Последовательно обсудим части этого определения. В части (1) указывается, что создается таблица с именем EMP. В части (2) определяется

* Как можно видеть из приведенных объяснений, ссылочные действия служат тому, чтобы автоматически поддерживать ссылочную целостность при обновлениях таблиц, к строкам которых ведут ссылки. Довольно часто ссылочные действия, являющиеся частью определения внешнего ключа, называют *декларативными триггерами*.

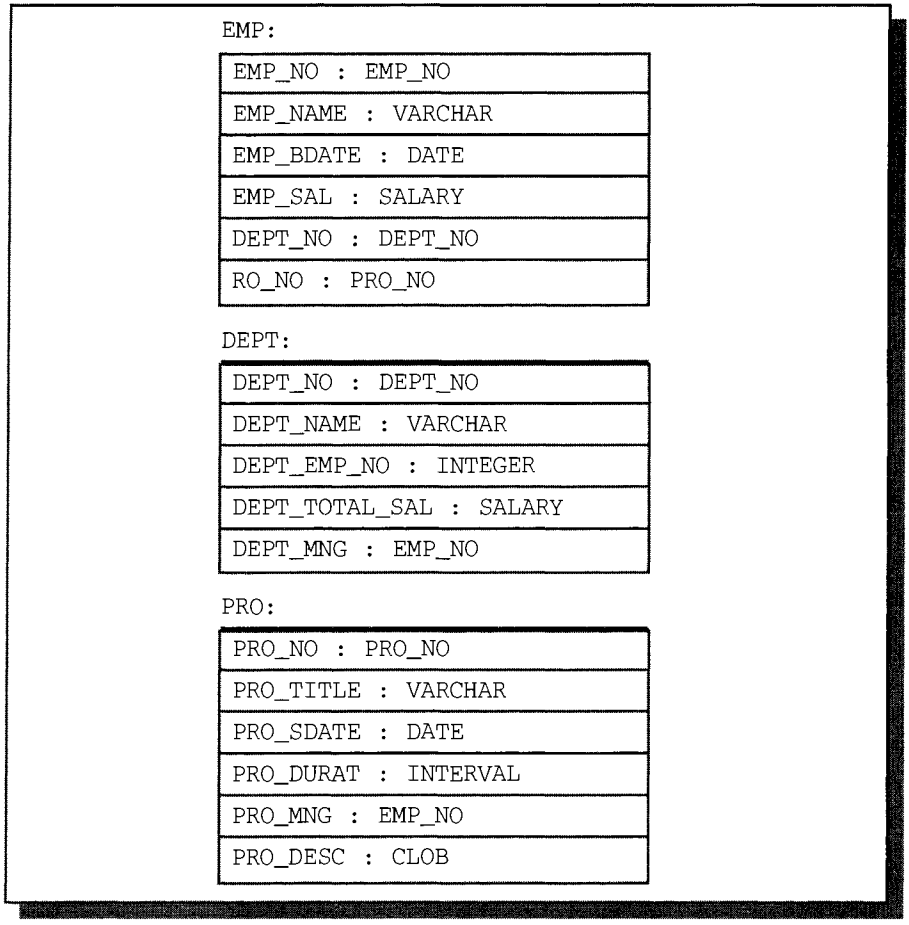


Рис. 12.1. Заголовки таблиц EMP, DEPT и PRO

столбец EMP_NO на домене EMP_NO. У этого столбца не определено значение по умолчанию, и он объявлен первичным ключом таблицы (это ограничение целостности добавляется через AND к ограничениям, унаследованным столбцом от определения домена). Помимо прочего, это означает неявное указание запрета для данного столбца неопределенных значений. В части (3) определен столбец EMP_NAME на базовом типе данных символьных строк переменной длины с максимальной длиной 20. Для столбца указано значение по умолчанию – строка 'Incognito', и в качестве ограничения целостности запрещены неопределенные значения. В части (4) определяется столбец EMP_BDATE (дата рождения служащего). Он имеет тип данных DATE, значением по умолчанию является NULL (даты рождения некоторых

служащих неизвестны). Кроме того, ограничение столбца запрещает принимать на работу лиц, о которых известно, что они родились до Октябрьского переворота. В части (5) определен столбец EMP_SAL на домене SALARY. Значение по умолчанию и ограничения целостности наследуются из определения домена. В части (6) столбец DEPT_NO определяется на одноименном домене (для наших целей его определение несущественно), но явно объявляется, что значением по умолчанию этого столбца будет NULL (некоторые служащие не приписаны ни к какому отделу). Кроме того, добавляется ограничение внешнего ключа: столбец DEPT_NO ссылается на первичный ключ таблицы DEPT. Определено ссылочное действие: при удалении строки из таблицы DEPT во всех строках таблицы EMP, ссылавшихся на эту строку, столбцу DEPT_NO должно быть присвоено неопределенное значение. В части (7) определяется столбец PRO_NO. Его определение аналогично определению столбца DEPT_NO, но ограничение внешнего ключа вынесено в часть (8), где оно определяется в полной форме как табличное ограничение. Наконец, в части (9) определяется табличное проверочное ограничение с именем PRO_EMP_NO, которое требует, чтобы ни в одном проекте не участвовало больше 50 служащих (правила построения соответствующего условного выражения поясняются в следующих лекциях).

Определим таблицу DEPT:

```
(1) CREATE TABLE DEPT (
(2) DEPT_NO DEPT_NO PRIMARY KEY,
(3) DEPT_EMP_NO INTEGER NO NULL CHECK (VALUE BETWEEN 1 AND 100),
(4) DEPT_NAME VARCHAR(200) DEFAULT 'Nameless' NOT NULL,
(5) DEPT_TOTAL_SAL SALARY DEFAULT 1000000.00
     NO NULL CHECK (VALUE > = 100000.00),
(6) DEPT_MNG EMP_NO DEFAULT NULL
     REFERENCES EMP ON DELETE SET NULL
     CHECK (IF (VALUE IS NOT NULL) THEN
       ((SELECT COUNT(*) FROM DEPT
         WHERE DEPT.DEPT_MNG = VALUE) = 1),
(7) CHECK (DEPT_EMP_NO =
     (SELECT COUNT(*) FROM EMP
       WHERE DEPT_NO = EMP.DEPT_NO)),
(8) CHECK (DEPT_TOTAL_SAL >=
     (SELECT SUM(EMP_SAL) FROM EMP
         WHERE DEPT_NO = EMP.DEPT_NO)));
```

Это определение мы обсудим в менее систематической манере, чем предыдущее. Отметим только наиболее интересные моменты. В части (3) столбец DEPT_EMP_NO (число служащих в отделе) определен на базовом типе INTEGER без значения по умолчанию, с запретом неопределенного значения и с проверочным ограничением, устанавливающем допустимый ди-

апазон значений числа служащих в отделе. Еще одно проверочное ограничение этого столбца – (7) – вынесено на уровень определения табличного ограничения. Это ограничение устанавливает, что в каждой строке таблицы DEPT значение столбца DEPT_EMP_NO должно равняться общему числу строк таблицы EMP, в которых значение столбца DEPT_NO равно значению одноименного столбца данной строки таблицы DEPT.* В части (5) для определения столбца DEPT_TOTAL_SAL (объем фонда заработной платы отдела) используется домен SALARY**. Но при этом явно установлено значение столбца по умолчанию (отличное от значения по умолчанию домена), запрещено наличие неопределенных значений и введено дополнительное проверочное ограничение, определяющее нижний порог объема фонда заработной платы отдела. Еще одно проверочное ограничение – (8) – вынесено на уровень определения табличного ограничения. Это ограничение устанавливает, что в каждой строке таблицы DEPT значение столбца DEPT_TOTAL_SAL должно быть не меньше суммы значений столбца EMP_SAL во всех строках таблицы EMP, в которых значение столбца DEPT_NO равно значению одноименного столбца данной строки таблицы DEPT.*** Обратите внимание на определение столбца DEPT_MNG – часть (6). Этот столбец объявляется внешним ключом таблицы DEPT. Но мы хотим сказать больше. У отдела могут временно отсутствовать руководители, поэтому в столбце допускаются неопределенные значения. Но если у отдела имеется руководитель, то он должен являться руководителем только этого отдела. На первый взгляд можно было бы воспользоваться ограничением столбца UNIQUE. Но такое ограничение допускало бы наличие неопределенного столбца DEPT_MNG только в одной строке таблицы DEPT, а мы хотим допустить отсутствие руководителя у нескольких отделов. Поэтому потребовалось ввести более громоздкое проверочное ограничение столбца.

По поводу двух приведенных определений базовых таблиц у читателей могут возникнуть два вопроса:

(a) почему проверочное ограничение (9) в первом определении и проверочные ограничения (7) и (8) во втором определении вынесены из определений соответствующих столбцов, хотя формально являются именно ограничениями столбцов?

(b) почему ограничению (9) в первом определении присвоено явное имя, а ограничения (7) и (8) во втором определении оставлены безымянными?

---

* Другими словами, это естественное ограничение требует, чтобы значения столбца DEPT_EMP_NO были «правильными», т.е. действительно соответствовали числу служащих, работающих в данном отделе.

** По этой причине мы ввели в предыдущей лекции такую большую верхнюю границу – 20000000.00 – значений домена SALARY.

*** Другими словами, это естественное ограничение требует, чтобы размер фонда заработной платы отдела никогда не был меньше суммарной зарплаты, получаемой служащими этого отдела.

На первый вопрос можно ответить следующим образом. Да, эти ограничения можно было бы включить в определения столбцов. Это дело вкуса. Но все три ограничения являются очень важными с точки зрения организации таблиц в целом. Поэтому лучше показывать их на уровне определения табличных ограничений.

Вот ответ на второй вопрос. Ограничение (9) в первом определении и ограничения (7) и (8) во втором определении внешне похожи, но сильно отличаются по своей сути. Ограничения (7) и (8) связаны с агрегатной семантикой столбцов DEPT_EMP_NO и DEPT_TOTAL_SAL таблицы DEPT. Отмена ограничений изменила бы смысл этих столбцов. Ограничение (9) является текущим административным ограничением. Если руководство предприятия примет решение разрешить использовать в одном проекте более 50 служащих, ограничение можно отменить без изменения смысла столбцов таблицы EMP. Имея это в виду, мы ввели явное имя ограничения (9), чтобы при необходимости имелась простая возможность отменить это ограничение с помощью оператора ALTER TABLE.

Наконец, определим таблицу PRO.

```
(1) CREATE TABLE PRO (
(2) PRO_NO PRO_NO PRIMARY KEY,
(3) PRO_TITLE VARCHAR(200)DEFAULT 'No title' NOT NULL,
(4) PRO_SDATE DATE DEFAULT CURRENT_DATE NOT NULL,
(5) PRO_DURAT INTERVAL YEAR DEFAUL INTERVAL '1' YEAR NOT NULL,
(6) PRO_MNG EMP_NO UNIQUE NOT NULL REFERENCES EMP ON DELETE NO ACTION,
(7) PRO_DESC CLOB(10M));
```

Столбец PRO_SDATE содержит дату начала проекта, а столбец PRO_DURAT – продолжительность проекта в годах. В этом определении имеет смысл прокомментировать часть (6). Мы считаем, что если отдел, по крайней мере временно, может существовать без руководителя, то у проекта всегда должен быть менеджер. Поэтому определение столбца PRO_MNG является гораздо более строгим, чем определение столбца DEPT_MNG в таблице DEPT. Сочетание ограничений UNIQUE и NOT NULL при отсутствии значений по умолчанию приводит к абсолютной уникальности значений столбца PRO_MNG. Другими словами, этот столбец обладает всеми характеристиками первичного ключа, хотя объявлен только как возможный ключ. Кроме того, он объявлен как внешний ключ с действием при удалении строки таблицы EMP с соответствующим значением первичного ключа NO ACTION, запрещающим такие удаления. В совокупности это гарантирует, что у любого проекта будет существовать менеджер, являющийся служащим предприятия. В части (5) столбец PRO_DESC (описание проекта) определен как большой символьный объект с максимальным размером 10 Мбайт.

Изменение определения базовой таблицы

Оператор изменения определения базовой таблицы ALTER TABLE имеет следующий синтаксис:

```
base_table_alteration ::= ALTER TABLE base_table_name
            column_alteration_action
          | base_table_constraint_alternation_action
```

Как видно из этого синтаксического правила, при выполнении одного оператора ALTER TABLE может быть выполнено либо действие по изменению определения столбца, либо действие по изменению определения табличного ограничения целостности.

Добавление, изменение или удаление определения столбца

Действие по изменению определения столбца специфицируется в следующем синтаксисе:

```
column_alteration_action ::=
        ADD [ COLUMN ] column_definition
      | ALTER [ COLUMN ] column_name
            { SET default_definition | DROP DEFAULT }
      | DROP  [ COLUMN ] column_name
            { RESTRICT | CASCADE }
```

Итак, с использованием оператора ALTER TABLE можно добавлять к определению таблицы определение нового столбца (действие ADD) и изменять или отменять определение существующего столбца (действия ALTER и DROP соответственно).

Смысл действия ADD COLUMN почти полностью совпадает со смыслом раздела определения столбца в операторе CREATE TABLE. Указывается имя нового столбца, его тип данных или домен. Могут определяться значение по умолчанию и ограничения целостности. Однако имеется одно существенное отличие: столбец, определяемый в действии ADD оператора ALTER TABLE, добавляется к уже существующей таблице, которая, скорее всего, содержит некоторый набор строк. В каждой из существующих строк новый столбец должен содержать некоторое значение, и считается, что сразу после выполнения действия ADD этим значением является значение столбца по умолчанию. Поэтому столбец, определяемый в действии ADD, обязательно должен иметь значение по умолчанию, т. е. для него недопустима ситуация, когда значением по умолчанию явно или неявно объявлено неопределенное значение (NULL), но среди ограничений целостности столбца присутствует ограничение NOT NULL.

В действии ALTER COLUMN можно изменить (SET default_definition) или отменить определение значения по умолчанию для существующего столбца. Правила определения нового действующего значения столбца по умолчанию совпадают с соответствующими правилами, обсуждавшимися в подразделе определения столбца в операторе CREATE TABLE. Заметим, что изменение значения столбца по умолчанию не оказывает влияния на состояние существующих строк таблицы (даже если в некоторых из них хранится предыдущее значение столбца по умолчанию). Если столбец определен на домене, у которого существует значение по умолчанию, то после отмены определения значения столбца по умолчанию для этого столбца начинает действовать значение по умолчанию домена.

Действие DROP COLUMN отменяет определение существующего столбца (удаляет его из таблицы). Действие DROP COLUMN отвергается, если:

(a) указанный столбец является единственным столбцом таблицы;

(b) или в этом действии присутствует спецификация RESTRICT, и данный столбец используется в определении каких-либо представлений или ограничений целостности*.

Если в действии присутствует спецификация CASCADE, то его выполнение порождает неявное выполнение оператора DROP для всех представлений и ограничений целостности, в определении которых используется данный столбец.

Примеры изменения определения столбца

Предположим, что на предприятии ввели систему премирования служащих. Каждый служащий может дополнительно к зарплате получать ежемесячную премию, не превышающую размер его зарплаты. Тогда разумно добавить к таблице EMP новый столбец EMP_BONUS, используя оператор ALTER TABLE:

```
ALTER TABLE EMP
ADD EMP_BONUS SALARY DEFAULT NULL
   CONSTRAINT BONSAL CHECK (VALUE < EMP_SAL);
```

Обратите внимание, что мы присвоили проверочному ограничению столбца явное имя, чтобы в случае, если ограничения на размер премии изменятся (что вполне возможно), можно было бы легко отменить это ограничение, воспринимая его как табличное.

При определении столбца EMP_SAL таблицы EMP для этого столбца явно не определялось значение по умолчанию (оно наследовалось из оп-

* Не считая те табличные ограничения целостности, которые (a) определены в составе определения базовой таблицы, содержащей данный столбец и (b) не содержат ссылок на какие-либо другие столбцы.

ределения домена). Если в какой-то момент это стало неправильным (например, повысился размер минимальной зарплаты), можно установить новое значение по умолчанию:

```
ALTER TABLE EMP ALTER EMP_SAL SET DEFAULT 15000.00.
```

При определении столбца DEPT_TOTAL_SAL таблицы DEPT для него было установлено значение по умолчанию 1000000. Главный бухгалтер предприятия может быть недоволен тем, что такие важные данные, как объем фонда зарплаты отделов, могут устанавливаться по умолчанию. Тогда можно отменить это значение по умолчанию:

```
ALTER TABLE DEPT ALTER DEPT_TOTAL_SAL DROP DEFAULT.
```

Обратите внимание, что после выполнения этого оператора при вставке новой строки в таблицу DEPT всегда потребуется явно указывать значение столбца DEPT_TOTAL_SAL. Хотя формально у столбца будет существовать значение по умолчанию, наследуемое от домена SALARY (10000.00), оно не может быть занесено в таблицу DEPT, поскольку противоречит ограничению столбца DEPT_TOTAL_SAL CHECK (VALUE >= 100000.00).

Можно задуматься, действительно ли требуется поддерживать в таблице DEPT столбец DEPT_EMP_NO. Как мы видели, для его поддержки требуется проверять громоздкое ограничение целостности, а число служащих в любом отделе можно получить динамически с помощью простого запроса к таблице EMP (собственно, этот запрос входит в ограничение целостности). Поэтому может оказаться разумным отменить определение столбца DEPT_EMP_NO, выполнив следующий оператор ALTER TABLE:

```
ALTER TABLE DEPT DROP DEPT_EMP_NO CASCADE.
```

Напомним, что спецификация CASCADE ведет к тому, что при выполнении оператора будет уничтожено не только определение указанного столбца, но и определения всех ограничений целостности и представлений, в которых используется уничтожаемый столбец. В нашем случае единственное связанное с этим столбцом ограничение целостности, определенное вне определения столбца, было бы отменено, даже если бы в операторе отмены определения столбца DEPT_EMP_NO содержалась спецификация RESTRICT, поскольку это единственное внешнее определение ограничения является ограничением только столбца DEPT_EMP_NO.

Изменение набора табличных ограничений

Действие по изменению набора табличных ограничений специфицируется в следующем синтаксисе:

```
base_table_constraint_alternation_action ::=
     ADD [ CONSTRAINT ] base_table_constraint_definition
   | DROP CONSTRAINT constraint_name
        { RESTRICT | CASCADE }
```

Действие ADD [CONSTRAINT] позволяет добавить к набору существующих ограничений таблицы новое ограничение целостности. Можно считать, что новое ограничение добавляется через AND к конъюнкции существующих ограничений, как если бы оно определялось в составе оператора CREATE TABLE. Но здесь имеется одно существенное отличие. Если внимательно посмотреть на все возможные виды табличных ограничений, можно убедиться, что любое из них удовлетворяется на пустой таблице. Поэтому, какой бы набор табличных ограничений ни был определен при создании таблицы, это определение является допустимым и не препятствует выполнению оператора CREATE TABLE. При добавлении нового табличного ограничения с использованием действия ADD [CONSTRAINT] мы имеем другую ситуацию, поскольку таблица, скорее всего, уже содержит некоторый набор строк, для которого условное выражение нового ограничения может принять значение *false*. В этом случае выполнение оператора ALTER TABLE, включающего действие ADD [CONSTRAINT], отвергается.

Выполнение действия DROP CONSTRAINT приводит к отмене определения существующего табличного ограничения. Можно отменить определение только именованных табличных ограничений. Спецификации RESTRICT и CASCADE осмыслены только в том случае, если отменяемое ограничение является ограничением возможного ключа (UNIQUE или PRIMARY KEY)*. При указании RESTRICT действие отвергается, если на данный возможный ключ ссылается хотя бы один внешний ключ. При указании CASCADE действие DROP CONSTRAINT выполняется в любом случае, и все определения таких внешних ключей также отменяются.

Примеры изменения набора табличных ограничений

Напомним, что мы добавили к таблице EMP столбец EMP_BONUS, в котором сохраняются размеры ежемесячных премий служащих. Предположим, что премии выплачиваются из фонда заработной платы отдела, в котором работает служащий. Тогда проверочное ограничение столбца DEPT_TOTAL_SAL, устанавливающее, что объем фонда зарплаты отдела

* Хотя формально требуется указывать одно из этих ключевых слов в любом действии DROP CONSTRAINT.

не должен быть меньше суммарной зарплаты служащих этого отдела, становится недостаточным, и нам требуется добавить к набору ограничений таблицы DEPT новое ограничение:

```
ALTER TABLE DEPT ADD CONSTRAINT TOTAL_INCOME
CHECK (DEPT_TOTAL_SAL >=
   (SELECT SUM(EMP_SAL + COALESCE(EMP_BONUS,0))FROM EMP
      WHERE EMP.DEPT_NO = DEPT_NO)).
```

Хотя это ограничение на вид довольно сложное, смысл его очень прост: суммарный доход служащих отдела не должен превышать объем зарплаты отдела. В арифметическом выражении под знаком агрегатной операции SUM используется операция COALRSCE. Эта двуместная операция определяется следующим образом:

```
COALESCE (x, y) IF x IS NOT NULL THEN x ELSE y,
```

т. е. значением операции является значение первого операнда, если оно не равно NULL, и значение второго операнда – в противном случае. Нам пришлось воспользоваться этой операцией, поскольку в столбце EMP_BONUS допускается наличие неопределенных значений.

Понятно, что новое ограничение столбца DEPT_TOTAL_SAL сильнее предыдущего, и это предыдущее ограничение можно было бы отменить. Конечно, с логической точки зрения наличие обоих ограничений ничему не повредит (предыдущее ограничение является логическим следствием нового), но при использовании не слишком интеллектуальной реализации SQL может привести к замедлению работы системы, поскольку оба ограничения могут проверяться независимо. К сожалению, при определении таблицы EMP мы не присвоили явное имя проверочному ограничению столбца DEPT_TOTAL_SAL и поэтому не можем немедленно продемонстрировать оператор отмены этого ограничения. Это не значит, что его нельзя отменить вообще. В стандарте языка SQL требуется, чтобы ограничения целостности, которым не назначены явные имена, получали имена, автоматически генерируемые системой. Любой квалифицированный пользователь SQL-ориентированной СУБД (скорее всего, администратор) может обнаружить имя любого ограничения, обратившись к системной таблице-каталогу ограничений целостности.

Кстати, новому ограничению мы присвоили явное имя. К этому привели следующие рассуждения. Когда создавалась исходная схема базы данных, руководство предприятия ничего не говорило о премиях служащих. Теперь начальство решило, что премии будут выплачиваться из фонда зарплаты. Для этого, мы добавили новый столбец и новое ограничение

целостности. Но кто знает, не изменится ли снова решение о премиях? Чтобы не добавлять себе работы в будущем, дадим новому ограничению явное имя и не будем отменять предыдущее ограничение.*

При определении таблицы EMP было специфицировано проверочное табличное ограничение PRO_EMP_NO, устанавливающее, что над одним проектом не должно работать более 50 служащих. Мы уже отмечали, что это ограничение носит чисто административный характер и может быть отменено без нарушения логики базы данных. Для отмены ограничения нужно выполнить следующий оператор:

```
ALTER TABLE EMP DROP CONSTRAINT PRO_EMP_NO;
```

Отмена определения (уничтожение) базовой таблицы

Для отмены определения (уничтожения) базовой таблицы служит оператор DROP TABLE, задаваемый в следующем синтаксисе:

```
DROP TABLE base_table_name { RESTRICT | CADCADE }
```

Успешное выполнение оператора приводит к тому, что указанная базовая таблица перестает существовать. Уничтожаются все ее строки, определения столбцов и табличные определения целостности. При наличии спецификации RESTRICT выполнение оператора DROP TABLE отвергается, если имя таблицы используется в каком-либо определении представления или ограничения целостности**. При наличии спецификации CASCADE оператор выполняется в любом случае, и все определения представлений и ограничений целостности, содержащие ссылки на данную таблицу, также отменяются.

Средства определения и отмены общих ограничений целостности

Виды ограничений целостности, с которыми мы имели дело в предыдущих разделах этой лекции, образуют иерархию (рис. 12.2).

Ограничения целостности, входящие в определение домена, наследуются всеми столбцами, определенными на этих доменах, и являются ограничениями этих столбцов. Кроме того, в определение столбца могут входить определения дополнительных ограничений. Ограничения целостности, входящие в определение столбца (включая ограничения, унаследованные из определения домена), являются ограничениями таблицы, в состав опреде-

* Не следует расценивать эти рассуждения как руководство к действию. Мы привели их только для того, чтобы обосновать пример, хотя рассуждения, конечно, не лишены смысла.
** Не считая те ограничения целостности, которые (a) определены в составе определения данной базовой таблицы и (b) не ссылаются на какие-либо другие базовые таблицы.

ления которой входит определение данного столбца. Кроме того, в определение таблицы могут входить определения дополнительных ограничений.

Но иерархия видов ограничений целостности этим не исчерпывается. Ограничения целостности, входящие в определение таблицы (включая явные и унаследованные от определения доменов ограничения столбцов), представляют собой ограничения базы данных, частью которой является данная таблица. Кроме того, могут определяться дополнительные ограничения базы данных. В стандарте SQL такие дополнительные ограничения базы данных называются ASSERTION, а мы их будем называть общими ограничениями целостности.

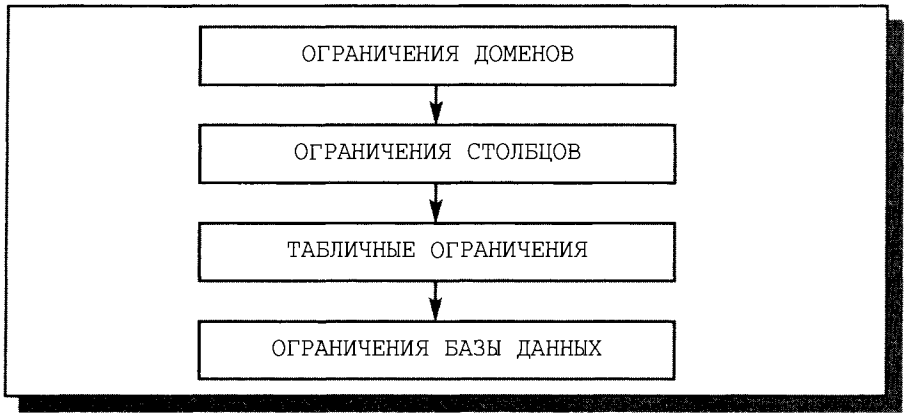


Рис. 12.2. Иерархия видов ограничений целостности

Определение общих ограничений целостности

Для определения общего ограничения целостности служит оператор CREATE ASSERTION, задаваемый в следующем синтаксисе:

```
CREATE ASSERTION constraint_name CHECK (conditional_expression)
```

Заметим, что при создании общего ограничения целостности его имя всегда должно указываться явно. Хотя синтаксис определения общего ограничения совпадает с синтаксисом определений ограничений столбца и таблицы, в данном случае допускаются только специальные виды условных выражений. Мы не можем сейчас точно сформулировать свойства этих видов условий, поскольку отложили подробное рассмотрение разновидностей условных выражений до следующих лекций. Если говорить неформально, то особые свойства условий связаны с тем, что при определении общих ограничений целостности контекстом, в котором вычисляется условное выражение, является весь набор таблиц

базы данных, а не набор строк таблицы, как это было при определении табличных ограничений. Продемонстрируем и прокомментируем несколько примеров определений общих ограничений целостности.

В определении таблицы EMP содержалось ограничение столбца EMP_BDATE:

```
CHECK (EMP_BDATE >= '1917-10-24')
```

(к работе на предприятии допускаются только те лица, которые родились после Октябрьского переворота). Вот каким образом можно определить такое же ограничение на уровне общих ограничений целостности:

```
CREATE ASSERTION MIN_EMP_BDATE CHECK
   ((SELECT MIN(EMP_BDATE)) FROM EMP) >= '1917-10-24').
```

В логическом условии этого общего ограничения выбирается минимальное значение столбца EMP_BDATE (дата рождения самого старого служащего). Значением условного выражения будет *false* в том и только в том случае, если среди служащих имеется хотя бы один, родившийся до указанной даты.

Теперь переформулируем в виде общего ограничения целостности ограничение таблицы EMP PRO_EMP_NO, которое определялось следующим образом:

```
CONSTRAINT PRO_EMP_NO CHECK
   ((SELECT COUNT (*) FROM EMP E
     WHERE E.PRO_NO = PRO_NO) <= 50)
```

(над одним проектом не может работать более 50 служащих).

Вот формулировка эквивалентного общего ограничения целостности:

```
CREATE ASSERTION NEW_PRO_EMP_NO CHECK
   ( NOT EXISTS (SELECT PRO_NO FROM EMP GROUP BY PRO_NO
                 HAVING COUNT* > 50)).
```

Логическое выражение этого ограничения может принимать только значения *true* и *false*. Внутренний оператор выборки группирует строки таблицы EMP таким образом, что в одну группу попадают все строки с одинаковым значением столбца PRO_NO. Затем эти группы фильтруются по условию раздела HAVING, и остаются только группы, включающие более 50 строк. В результирующей таблице содержатся строки из одного столбца, содержащего значение PRO_NO оставшихся групп. Предикат NOT

EXISTS принимает значение *true* тогда и только тогда, когда эта результирующая таблица не содержит ни одной строки, т. е. нет ни одного проекта, в котором работает больше 50 служащих.

Покажем, как можно сформулировать в виде общего ограничения целостности ограничение внешнего ключа. Например, приведем такую эквивалентную формулировку для определения внешнего ключа PRO_NO, входящего в состав определения таблицы EMP:

```
FOREIGN KEY PRO_NO REFERENCES PRO (PRO_NO)
```

В виде общего ограничения целостности это может выглядеть следующим образом:

```
(1) CREATE ASSERTION FK_PRO_NO CHECK
(2) ( NOT EXISTS (SELECT * FROM EMP WHERE PRO_NO IS NOT NULL AND
(3)     NOT EXISTS (SELECT * FROM PRO
(4)       WHERE PRO.PRO_NO = EMP.PRO_NO))).
```

Логическое выражение этого ограничения выглядит достаточно сложным и нуждается в пояснении. Условие выборки оператора SELECT на строке (2) состоит из двух частей, связанных через AND. Первая часть отфильтровывает те строки таблицы EMP, у которых в столбце PRO_NO содержится NULL. Если этот столбец содержит NULL во всех строках таблицы, то результирующая таблица оператора выборки на строке (2) будет пустой, и значением предиката NOT EXISTS будет *true*, т. е. ограничение удовлетворяется.

Теперь предположим, что в таблице EMP нашлась строка emp, в столбце PRO_NO которой содержится значение, отличное от NULL. Назовем это значение cand_pro_no. Для него вычисляется вторая часть условия выборки оператора SELECT на строке (2). Оператор выборки на строке (3) выбирает все строки таблицы PRO, значение столбца PRO_NO которых равняется cand_pro_no. Если для данного значения cand_pro_no нашлась хотя бы одна такая строка, то результирующая таблица оператора выборки на строке (3) будет непустой, и значением предиката NOT EXISTS на строке (3) будет *false*. Соответственно, все условие выборки первого оператора SELECT примет значение *false*, и строка со значением cand_pro_no в столбце PRO_NO будет отфильтрована.*

Если же найдется хотя бы одна строка таблицы EMP с таким значением cand_pro_no столбца PRO_NO, что в таблице PRO не найдется ни одной строки, значение столбца PRO_NO которой равнялось бы этому cand_pro_no, то результирующая таблица оператора выборки на строке (3) будет пустой, и значением предиката NOT EXISTS на строке (3) будет *true*. Тогда все условие выборки первого оператора SELECT примет значение *true*, и эта строка таб-

* Это означает, что cand_pro_no является допустимым значением внешнего ключа.

лицы EMP будет пропущена в результирующую таблицу. Значением предиката NOT EXISTS будет *false*, т. е. ограничение не удовлетворяется.

Мы сознательно привели такое подробное пояснение не только для того, чтобы прояснить смысл условного выражения общего ограничения целостности FK_PRO_NO, но и чтобы дать понять, во что реально вырождается простая синтаксическая конструкция определения внешнего ключа. Как показывает опыт, многие начинающие проектировщики SQL-ориентированных баз данных думают, что ссылочные ограничения так же легко поддерживать, как определять.

Наконец, сформулируем общее ограничение целостности, состоящее в том, что никакой менеджер проекта не должен иметь суммарный общий доход, больший суммарного дохода руководителя отдела, в котором работает этот менеджер.

```
(1) CREATE ASSERTION PRO_MNG_CONSTR CHECK
(2) NOT EXISTS (SELECT * FROM EMP EMP1, EMP EMP2, DEPT, PRO WHERE
(3)             EMP1.EMP_NO = PRO.PRO_MNG AND
(4)             EMP1.DEPT_NO = DEPT.DEPT_NO AND
(5)             DEPT.DEPT_MNG = EMP2.EMP_NO AND
(6)             EMP1.EMP_SAL + COALESCE (EMP1.EMP_BONUS,0) >
(7)             EMP2.EMP_SAL + COALESCE (EMP2.EMP_BONUS,0);
```

В логическом выражении этого ограничения используется оператор выборки SELECT, в разделе перечня таблиц (FROM) впервые в этом курсе используется несколько таблиц. Такие запросы в SQL называются *запросами с соединениями*, и мы воспользуемся случаем, чтобы пояснить на примере (конечно, предварительно), как их следует понимать в соответствии со стандартом языка SQL.

Итак, в разделе FROM оператора выборки, используемого в логическом условии этого ограничения, через запятую перечислены четыре элемента — EMP EMP1, EMP EMP2, DEPT и PRO. Выражение вида EMP ANOTHER_NAME означает применение своего рода операции переименования. Внутри запроса столбцы этого «экземпляра» EMP имеют «квалифицированные» имена вида ANOTHER_NAME.column_name, где column_name обозначает имя существующего столбца таблицы EMP.

Вычисление оператора выборки начинается с того, что формируется расширенное декартово произведение всех таблиц, указанных в разделе FROM. В данном случае схема результирующей таблицы раздела FROM будет содержать следующие имена столбцов: EMP1.EMP_NO, EMP1.EMP_NAME, EMP1. EMP_BDATE, EMP1. EMP_SAL, EMP1.EMP_BONUS, EMP1. DEPT_NO, EMP1. PRO_NO, EMP2.EMP_NO, EMP2.EMP_NAME, EMP2. EMP_BDATE, EMP2. EMP_SAL, EMP2.EMP_BONUS, EMP2. DEPT_NO, EMP2. PRO_NO, DEPT.DEPT_NO, DEPT.DEPT_EMP_NO, DEPT.DEPT_TOTAL_SAL, DEPT.DEPT_MNG, PRO.PRO_NO,

PRO.PRO_TITLE, PRO.PRO_SDATE, PRO.PRO_DURAT, PRO.PRO_MNG, PRO_DESC. Для удобства назовем эту «широкую» таблицу ALL_TOGETHER.*

Условие раздела WHERE состоит из четырех частей, связанных через AND. Обсудим их последовательно. После проверки условия EMP1.EMP_NO = PRO.PRO_MNG в таблице ALL_TOGETHER останутся все служащие-менеджеры проектов вместе со своими проектами в комбинации со всеми возможными отделами и всеми возможными служащими (назовем эту отфильтрованную таблицу ALL_TOGETHER_STEP1). После проверки условия EMP1.DEPT_NO = DEPT.DEPT_NO в таблице ALL_TOGETHER_STEP1 останутся все служащие-менеджеры проектов вместе со своими проектами и вместе с описанием своих отделов в комбинации со всеми возможными служащими (назовем эту отфильтрованную таблицу ALL_TOGETHER_STEP2). После проверки условия DEPT.DEPT_MNG = EMP2.EMP_NO в таблице ALL_TOGETHER_STEP2 останутся все служащие-менеджеры проектов вместе со своими проектами, вместе с описанием своих отделов и вместе с руководителями этих отделов (по одной строке для каждого допустимого сочетания «проект-менеджер_проекта-отдел_менеджера_проекта-руководитель_отдела_менеджера_проекта»). Назовем эту отфильтрованную таблицу ALL_TOGETHER_STEP3. Легко видеть, что после проверки условия EMP1.EMP_SAL + EMP1.EMP_BONUS > EMP2.EMP_SAL + EMP2.EMP_BONUS в таблице ALL_TOGETHER_STEP3 могут остаться только строки проект-менеджер_проекта-отдел_менеджера_проекта-руководитель_отдела_менеджера_проекта, в которых суммарный доход менеджера проекта превышает суммарный доход руководителя отдела, где работает менеджер проекта. Если хотя бы одна такая строка существует, то результат оператора выборки будет непустым, значением предиката NOT EXISTS будет *false*, и тем самым ограничение целостности PRO_MNG_CONSTR будет нарушено.

Отмена определения общего ограничения целостности

Для того чтобы отменить ранее определенное общее ограничение целостности, нужно воспользоваться оператором DROP ASSERTION, задаваемым в следующем синтаксисе:

```
DROP ASSERTION constraint_name
```

Вот пример оператора, отменяющего определение дискриминационного общего ограничения целостности PRO_MNG_CONSTR:

* Не следует воспринимать этот и следующие абзацы как описание того, как на самом деле выполняются подобные запросы в SQL-серверах. Это наиболее прямолинейный и малоэффективный способ выполнения запроса (хотя, в принципе, его можно применять и на практике). Мы выбрали этот способ описания, поскольку он максимально соответствует подходу к описанию семантики языка SQL, применяемому в стандарте языка. Кстати, основным отличием более практичных способов выполнения запросов с соединением является стремление к тому, чтобы избежать явного декартова произведения.

```
DROP ASSERTION PRO_MNG_CONSTR;
```

Немедленная и откладываемая проверка ограничений

На первый взгляд кажется, что ограничения целостности (всех видов) должны немедленно проверяться в случае выполнения любого действия, изменяющего содержимое базы данных (вставка в любую таблицу новой строки, изменение или удаление существующих строк). Однако можно определить такие ограничения целостности, логическое выражение которых будет принимать значение *false* при любой немедленной проверке. Одним из примеров такого ограничения является ограничение

```
CHECK (DEPT_EMP_NO =
   (SELECT COUNT(*) FROM EMP
     WHERE DEPT_NO = EMP.DEPT_NO))
```

из определения таблицы DEPT. Предположим, например, что в отдел зачисляется новый служащий. Тогда нужно выполнить две операции: (a) вставить новую строку в таблицу EMP и (b) изменить соответствующую строку таблицы DEPT (прибавить единицу к значению столбца DEPT_EMP_NO). Очевидно, что в каком бы порядке ни выполнялись эти операции, сразу после выполнения первой из них ограничение целостности будет нарушено, соответствующее действие будет отвергнуто, и мы никогда не сможем принять на работу нового служащего.

Поскольку ограничения целостности, немедленная проверка которых бессмысленна, являются нужными и полезными, в язык SQL включены средства, позволяющие регулировать время проверки ограничений. Если говорить более точно, в контексте каждой выполняемой транзакции каждое ограничение целостности должно находиться в одном из двух режимов: режиме *немедленной проверки (immediate)* или режиме *отложенной проверки (deferred)*. Все ограничения целостности, находящиеся в режиме немедленной проверки, проверяются при выполнении в транзакции любой операции, изменяющей состояние базы данных. Если действие операции нарушает какое-либо немедленно проверяемое ограничение целостности, то это действие отвергается.* Ограничения целостности, находящиеся в режиме отложенной проверки, проверяются при завершении транзакции (выполнении операции COMMIT)**. Если действия этой транзакции нарушают какое-либо отложенно проверяемое ограничение цело-

* Конечно, в грамотных реализациях SQL при выполнении операции проверяются не все немедленно проверяемые ограничения целостности, а только те, которые в принципе могут быть нарушены данной операцией.

** Мы снова вынуждены забегать вперед. Средства SQL для управления транзакциями более подробно обсуждаются в следующих лекциях.

стности, то транзакция откатывается (операция COMMIT трактуется как операция ROLLBACK).*

Для этого в качестве заключительной синтаксической конструкции к любому определению ограничения целостности (любого вида) может быть добавлена спецификация INITIALLY в следующей синтаксической форме:

```
INITIALLY { DEFERRED | IMMEDIATE } [ [ NOT ] DEFERRABLE ]
```

Эта спецификация указывает, в каком режиме должно находиться данное ограничение целостности в начале выполнения любой транзакции (INITIALLY IMMEDIATE означает, что в начале выполнения транзакции данное ограничение будет находиться в режиме немедленной проверки, а INITIALLY DEFERRED – что в начале любой транзакции ограничение будет находиться в режиме отложенной проверки), а также возможности смены режима этого ограничения при выполнении транзакции (DEFERRABLE означает, что для данного ограничения может быть установлен режим отложенной проверки, а NOT DEFERRABLE – что не может**).

Комбинация INITIALLY DEFERRED NOT DEFERRABLE является недопустимой. Если в определении ограничения спецификация начального режима проверки отсутствует, то подразумевается наличие спецификации INITIALLY IMMEDIATE. При наличии явной или неявной спецификации INITIALLY IMMEDIATE и отсутствии явного указания возможности смены режима подразумевается наличие спецификации NOT DEFERRABLE. При наличии спецификации INITIALLY DEFERRED и отсутствии явного указания возможности смены режима подразумевается наличие спецификации DEFERRABLE.

При выполнении транзакции можно изменить режим проверки некоторых или всех ограничений целостности для данной транзакции. Для этого используется оператор SET CONSTRAINTS, задаваемый в следующем синтаксисе:

```
SET CONSTRAINTS { constraint_name_commalist | ALL }
        { DEFERRED | IMMEDIATE }
```

Если в операторе указывается список имен ограничений целостности, то все они должны быть DEFERRABLE; если хотя бы для одного ограни-

* Конечно, в грамотных реализациях SQL при завершении транзакции проверяются не все отложенно проверяемые ограничения целостности, а только те, которые в принципе могут быть нарушены данной транзакцией.

** Для некоторых ограничений целостности режим отложенной проверки не имеет смысла. К таким ограничениям относятся, например, ограничения домена, ограничения NOT NULL и ограничения возможного ключа (хотя при их определении допускается указание DEFFERABLE). Если же возможный ключ используется в некотором определении внешнего ключа, то в стандарте SQL требуется, чтобы ограничение этого возможного ключа было NOT DEFFERABLE.

чения из списка это требование не выполняется, то операция SET CONSTRAINTS отвергается. При указании ключевого слова ALL режим устанавливается для всех ограничений, в определении которых явно или неявно было указано DEFERRABLE. Если в качестве желаемого режима проверки ограничений задано DEFERRED, то все указанные ограничения переводятся в режим отложенной проверки. Если в качестве желаемого режима проверки ограничений задано IMMEDIATE, то все указанные ограничения переводятся в режим немедленной проверки. При этом если хотя бы одно из этих ограничений не удовлетворяется, то операция SET CONSTRAINTS отвергается, и все указанные ограничения остаются в предыдущем режиме.

При выполнении операции COMMIT неявно выполняется операция SET CONSTRAINTS ALL IMMEDIATE. Если эта операция отвергается, то COMMIT срабатывает как ROLLBACK.

Заключение

В этой и предыдущей лекциях мы обсудили наиболее важные аспекты языка SQL, связанные с определением схемы базы данных, – типы данных SQL, средства определения доменов, базовых таблиц и ограничений целостности. Кроме того, были рассмотрены средства SQL, позволяющие динамически изменять и удалять определения этих объектов. Язык SQL устроен таким образом, что практически невозможно изложить какую-либо его часть независимо от других частей. И хотя эти две лекции по смыслу должны быть первыми среди лекций, посвященных SQL (было бы странно обсуждать операторы выборки строк из таблиц, вставки, изменения и удаления строк до обсуждения средств создания таблиц и ограничений целостности), нам пришлось забежать вперед и воспользоваться материалом следующих лекций для объяснения средств определения ограничений целостности. Надеюсь, что это не создало слишком больших неудобств для читателей, и отсутствие формальных определений удалось компенсировать наличием простых примеров.

Лекция 13. Язык баз данных SQL: общая характеристика оператора SELECT и организация списка ссылок на таблицы в разделе FROM

В этой и следующих трех лекциях рассматривается важнейший оператор языка SQL – оператор SELECT, предназначенный для выборки данных из SQL-ориентированной базы данных. Этот оператор имеет довольно сложную и развитую структуру, но, по нашему мнению, его необходимо знать любому специалисту, так или иначе связанному с использованием баз данных; поэтому в нашем курсе ему уделяется так много внимания. Первая лекция носит подготовительный характер. В ней мы рассматриваем виды скалярных выражений, используемые, прежде всего, в конструкциях оператора SELECT, обсуждаем базовую семантику выполнения этого оператора и анализируем принципы и разновидности указания таблиц, из которых производится выборка данных.

Ключевые слова: скалярное выражение, первичное выражение, агрегатная функция, скалярный подзапрос, численное выражение, вызов функций с численным значением, выражения символьных и битовых строк, вызов функций, возвращающих строчные значения, выражения даты-времени, вызовы функций, возвращающих значение дата-время, выражения со значениями типа временного интервала, булевские выражения, выражения с переключателем, выражение с поисковым переключателем, выражение с простым переключателем, выражение NULLIF, выражение COALESCE, оператор SELECT, семантика оператора выборки, раздел FROM, раздел WHERE, раздел GROUP BY, раздел HAVING, раздел SELECT, раздел ORDER BY, порождаемая таблица с горизонтальной связью, соединенная таблица, табличное выражение, спецификация запроса, выражение запросов, «теоретико-множественные» операции в SQL, раздел WITH выражения запросов, конструктор значения-строки, конструктор значения-таблицы, ссылки на базовые и представляемые таблицы, ссылки на порождаемые таблицы, представляемые таблицы, или представления (VIEW), оператор CREATE VIEW, оператор DROP VIEW.

Введение

Несмотря на то, что язык SQL является полным языком баз данных, включающим множество разнообразных средств определения схемы, ограничения и поддержки целостности базы данных, поддержки администрирования, заполнения и модификации таблиц базы данных, поддержки разработки приложений и т. д., для подавляющего большинства пользова-

телей этот язык остается языком запросов, т. е. языком, позволяющим формулировать произвольно сложные и точные декларативные запросы к базе данных.

Как отмечалось в конце предыдущей лекции, структура стандарта языка SQL фактически не позволяет описать одну часть языка (в частности, средства запросов) в отрыве от других частей. Тем не менее, полагая, что средства выборки данных составляют наиболее интересную и практически значимую часть языка, мы выделили для их рассмотрения несколько отдельных лекций.

Напомним, что в этом курсе мы ограничиваемся базовым подмножеством SQL:1999 и SQL:2003 («прямым SQL») и даже это подмножество описываем не в полном объеме стандарта. Кроме того, в данной лекции мы не будем точно придерживаться порядка введения понятий и синтаксических конструкций, принятого в стандарте языка. Мы начнем с некоторой общей картины, дающей представление об операторе выборки, а затем будем постепенно уточнять ее.

Скалярные выражения

Скалярное выражение* – это выражение, вырабатывающее результат некоторого типа, специфицированного в стандарте. Скалярные выражения являются основой языка SQL, поскольку, хотя это реляционный язык, все условия, элементы списков выборки и т. д. базируются именно на скалярных выражениях. В SQL:1999 имеется несколько разновидностей скалярных выражений. К числу наиболее важных разновидностей относятся численные выражения; выражения со значениями-строками символов; выражения со значениями даты-времени; выражения со значениями-временными интервалами; булевские выражения. Мы не будем слишком глубоко вникать в тонкости, но тем не менее приведем некоторые базовые спецификации и пояснения.

Конечно, материал этой лекции опирается на разделы «Типы данных SQL» и «Неявные и явные преобразования типа или домена» лекции 11, в которых упоминались некоторые базовые операции над значениями типов данных SQL и обсуждался оператор `CAST`, позволяющий разрешенным образом изменять тип данных результата скалярного выражения.

Прежде чем перейти к конкретным видам скалярных выражений, рассмотрим некоторые наиболее общие языковые конструкции, на которых эти выражения базируются.

* В стандарте языка SQL в качестве общего термина для обозначения таких выражения используется термин *value expression*. Однако в менее формальных публикациях обычно применяется более понятный термин *scalar expression*, для которого, вдобавок, существует адекватный русский эквивалент *скалярное выражение*. В этом курсе мы также предпочитаем использовать именно этот термин.

Общие синтаксические правила построения скалярных выражений

В SQL:2003 имеются девять разновидностей выражений в соответствии с девятью категориями типов данных, значения которых вырабатываются при вычислении выражения

```
value_expression ::=
     numeric_value_expression
   | string_value_expression
   | datetime_value_expression
   | interval_value_expression
   | boolean_value_expression
   | array_value_expression
   | multiset_value_expression
   | row_value_expression
   | user_defined_value_expression
   | reference_value_expression
```

Как уже отмечалось в начале этого раздела, мы ограничимся обсуждением первых пяти разновидностей выражений. В основе построения этих видов выражений лежит *первичное выражение*, определяемое следующим синтаксическим правилом:

```
value_expression_primary ::=
     unsigned_value_specification
   | column_reference
   | set_function_specification
   | scalar_subquery
   | case_expression
   | (value_expression)
   | cast_specification
```

В пределах этого курса можно считать, что спецификация беззнакового значения (unsigned_value_specification) – это всегда литерал соответствующего типа или вызов *ниладической* функции (например, CURRENT_USER)*. При вычислении выражения *V* для строки таблицы каждая ссылка на столбец (column_reference) этой таблицы, непосредственно

* Другие варианты появляются во встраиваемом и динамическом SQL, а также расширении языка, предназначенного для написания кода хранимых процедур, триггеров, методов определяемых пользователями типов и т.д. В любом случае беззнаковое значение известно до начала компиляции любой содержащей его конструкции языка SQL.

содержащаяся в V, рассматривается как ссылка на значение данного столбца в данной строке. Агрегатные функции (*функции над множествами* – set_function_specification) обсуждаются в следующих лекциях. Если первичное выражение является скалярным подзапросом (scalar_subquery, или подзапросом, результатом которого является таблица, состоящая из одной строки и одного столбца) и результат подзапроса пуст, то результат первичного выражения – неопределенное значение. (*Подзапросы* обсуждаются в следующей лекции, *выражения с переключателем* (case_expression) рассматриваются ниже в этом разделе.) Оператор явного преобразования типов (cast_specification) рассматривался в разделе «Неявные и явные преобразования типа или домена» лекции 11.

Численные выражения

Численное выражение – это выражение, значение которого относится к числовому типу данных. Вот формальный синтаксис численного выражения:

```
numeric_value_expression> ::= numeric_term
          | numeric_value_expression + term
          | numeric_value_expression - term
numeric_term ::= numeric_factor
          | numeric_term * numeric_factor
          | numeric_term / numeric_factor
numeric_factor ::= [ { + | - } ] numeric_primary
numeric_primary ::= value_expression_primary
          | numeric_value_function
```

Следует обратить внимание на то, что в численных выражениях SQL первичная составляющая (numeric_primary) является либо первичным выражением (см. выше), либо вызовом функции с численным значением (numeric_value_function). Из этого, в частности, следует, что в численные выражения могут входить выражения с переключателем и операции преобразования типов. Вызовы функций с численным значением определяются следующими синтаксическими правилами:

```
numeric_value_function ::=
          POSITION (character_value_expression
          IN character_value_expression)
          | { CHAR_LENGTH | CHARACTER_LENGTH }
          (string_value_expression)
```

```
| OCTET_LENGTH (string_value_expression)
| BIT_LENGTH (string_value_expression)
| EXTRACT ({ datetime_field | time_zone field }
    FROM { datetime_value_expression
         | interval_value_expression })
| CARDINALITY (array_value_expression
              | multiset_value_expression)
| ABS (numeric_value_expression)
| MOD (numeric_value_expression)
```

Мы достаточно подробно обсуждали функции определения позиции и длины по отношению к символьным и битовым строкам при рассмотрении соответствующих типов данных; здесь приводится только уточненный синтаксис их вызова. Функция EXTRACT извлечения поля из значений дата-время или интервал позволяет получить в виде точного числа с масштабом 0 значение любого поля (года, месяца, дня и т. д.). Какой конкретный тип точных чисел будет выбран – определяется в реализации. Функцию CARDINALITY мы обсуждали в лекции 11. Функции ABS и MOD возвращают абсолютное значение числа и остаток от деления одного целого значения на другое соответственно.

Выражения, значениями которых являются символьные или битовые строки

Выражения символьных и битовых строк – это выражения, значениями которых являются символьные или битовые строки. Соответствующие конструкции определяются следующим синтаксисом:

```
string_value_expression ::= character_value_expression
                          | bit_value_expression
character_value_expression ::= concatenation
                             | character_factor
concatenation ::= character_value_expression || character_factor
character_factor ::= character_primary [ collate_clause ]
character_primary ::= value_expression_primary
                    | string_value_function
bit_value_expression ::= bit_concatenation
                       | bit_factor
bit_concatenation ::= bit_value_expression || bit_primary
bit_primary ::= value_expression_primary
              | string value function
```

Если не вдаваться в тонкости, смысл выражений символьных и битовых строк понятен из описания синтаксиса: единственная применимая для построения выражений операция – это конкатенация, производящая «склейку» строк-операндов. Более важно то, что первичной составляющей выражения над строками может быть как первичное скалярное выражение (см. выше), так и вызов функций, возвращающих строчные значения. Репертуар и синтаксис вызова таких функций определяются следующими правилами:

```
string_value_function ::= character_value_function
                        | bit_value_function
character _value_function ::= SUBSTRING
   (character _value_expression
                          FROM start_position
                             [ FOR string_length ])
                        | SUBSTRING (character _value_expression
                          SIMILAR character _value_expression
                             ESCAPE character_value_expression)
                        | { UPPER | LOWER }
                             (character_value_expression)
                        | CONVERT (character_value_expression
                          USING conversion_name)
                        | TRANSLATE (character_value_expression)
                          USING translation_name)
                        | TRIM ([ {LEADING | TRAILING | BOTH} ]
                          [ character_value_expression ]
                          [ character_value_expression ])
                        | OVERLAY (character_value_expression
                          PLACING character_value_expression
                             FROM start_position
                                [ FOR string_length ])
bit _value_function ::= SUBSTRING (bit_value_expression
                          FROM start_position
                             [ FOR string_length ])
start_position ::= numeric_value_expression
string length ::= numeric_value_expression
```

Основные полезные функции – выделение подстроки (SUBSTRING) и замена малых букв на заглавные и наоборот (UPPER и LOWER) – мы упоминали при рассмотрении типов символьных и битовых строк. Обсуждение функции SUBSTRING ... SIMILAR ... ESCAPE отложим до следующей лекции. Как видно из описания синтаксиса функций, возвращающих

строчные значения, для символьных строк имеются еще четыре функции: CONVERT, TRANSLATE, TRIM и OVERLAY. По смыслу все они очень просты. Функция CONVERT меняет кодировку символов в заданной строке, причем набор символов не меняется. Способ задания правил перекодировки определяется в реализации. Функция TRANSLATE, наоборот, в соответствии с правилами трансляции «переводит» текстовую строку на другой язык (используя набор символов целевого алфавита). Кодировка не меняется. Функция TRIM «отсекает» последовательности указанного символа в начале, в конце или в конце и начале заданной строки. Наконец, функция OVERLAY заменяет указанную подстроку первого операнда строкой, заданной в качестве второго операнда.

Выражения даты-времени

К выражениям даты-времени мы относим выражения, вырабатывающие значения типа дата-время и интервал. Выражения даты-времени определяются следующими синтаксическими правилами:

```
datetime_value_expression ::=
              datetime_term
          | interval_value_expression + datetime term
          | datetime_value_expression + interval term
          | datetime value expression - interval term
datetime_term ::= datetime_primary
     [ AT { LOCAL | TIME ZONE interval_value_expression } ]
datetime_primary ::= value_expression_primary
          | datetime_value_function
```

Как видно из описания синтаксиса, сами выражения строятся очень просто – на основе обычных арифметических операций. Снова более интересны первичные составляющие – вызовы функций, возвращающих значение дата-время. Эти вызовы определяются следующим синтаксисом:

```
datetime_value_function ::= CURRENT_DATE
                          | CURRENT_TIME [ (precision) ]
                          | LOCALTIME [ (precision) ]
                          | CURRENT_TIMESTAMP [ (precision) ]
                          | LOCALTIMESTAMP [ (precision) ]
```

Видимо, приведенные синтаксические правила не нуждаются в комментариях: можно получить текущую дату, а также текущее время с желаемой точностью. Отличие функций LOCALTIME и LOCALTIMESTAMP от

CURRENT_TIME и CURRENT_TIMESTAMP, соответственно, состоит в том, что первая пара функций не возвращает смещение локального времени от Гринвича.

Синтаксис выражений со значениями типа интервал определяется следующими правилами:

```
interval_value_expression ::=
            interval_term
          | interval_value_expression + interval term
          | interval_value_expression - interval term
          | (datetime value expression - datetime term)
               interval_qualifier
interval_term ::= interval_factor
          | interval_term * numeric_factor
          | interval_term / numeric_factor
          | numeric_term * interval_factor

interval_factor ::= [ { + | - } ]
                    interval_primary [ <interval qualifier> ]
interval_primary ::= value_expression_primary
          | interval_value_function
```

Как видно из приведенных правил, выражения со значениями типа интервал устроены очень просто; почти вся содержательная информация была приведена при обсуждении соответствующего типа данных. Стоит только заметить, что квалификатор интервала указывается для того, чтобы явно специфицировать единицу измерения интервала. Поддерживается только одна функция ABS (абсолютное значение), аргументом которой является выражение со значением типа интервал.

Булевские выражения

К булевским выражениям относятся выражения, вырабатывающие значения булевского типа (напомним, что булевский тип языка SQL содержит три логических значения – *true*, *false* и *unknown*). Булевские выражения определяются следующими синтаксическими правилами:

```
boolean_value_expression ::= boolean_term
             | boolean_value_expression OR boolean_term
boolean_term ::= boolean_factor
               | boolean_term AND boolean_factor
boolean_factor ::= [ NOT ] boolean_test
```

```
boolean_test ::= boolean_primary [ IS [ NOT ] truth_value ]
truth_value ::= TRUE | FALSE | UNKNOWN
boolean_primary ::= predicate
      | (boolean_value_expression)
      | value_expression_primary
```

Выражения вычисляются слева направо с учетом приоритетов операций (наиболее высокий приоритет имеет унарная операция NOT, следующим уровнем приоритета обладает «мультипликативная» операция конъюнкции AND, и самый низкий приоритет у «аддитивной» операции дизъюнкции OR) и круглых скобок. Операции IS и IS NOT определяются следующими таблицами истинности:

IS	TRUE	FALSE	UNKNOWN
TRUE	*TRUE*	*FALSE*	*FALSE*
FALSE	*FALSE*	*TRUE*	*FALSE*
UNKNOWN	*FALSE*	*FALSE*	*TRUE*

IS NOT	TRUE	FALSE	UNKNOWN
TRUE	*FALSE*	*TRUE*	*TRUE*
FALSE	*TRUE*	*FALSE*	*TRUE*
UNKNOWN	*TRUE*	*TRUE*	*FALSE*

Выражения с переключателем

Выражения с переключателем в некотором смысле ортогональны рассмотренным выше видам выражений, поскольку разные выражения с переключателем могут вырабатывать значения разных типов в зависимости от типа данных элементов. Поскольку мы еще вообще не рассматривали этот вид выражений, обсудим их более подробно. Как обычно, начнем с синтаксиса:

```
case_expression ::= case_abbreviation
                  | case_specification

case_abbreviation ::= NULLIF (value_expression , value_expression)
                    | COALESCE (value_expression_comma_list)
case specification ::= simple_case | searched_case
```

```
simple_case ::= CASE value_expression simple_when_clause_list
                [ ELSE value_expression ] END

searched_case ::= CASE searched_when_clause_list
                [ ELSE value_expression ] END
simple_when_clause ::= WHEN value_expression
                           THEN value_expression
searched_when_clause ::= WHEN conditional_expression
                           THEN value_expression
```

Наиболее общим видом выражения с переключателем является выражение с поисковым переключателем (searched_case). Правила вычисления выражений этого вида состоят в следующем. Вычисляется логическое выражение, указанное в первом разделе WHEN списка (searched_when_clause_list). Если значение этого логического выражения равняется *true*, то значением всего выражения с поисковым переключателем является значение выражения, указанного в первом разделе WHEN после ключевого слова THEN. Иначе аналогичные действия производятся для второго раздела WHEN и т. д. Если ни для одного раздела WHEN при вычислении логического выражения не было получено значение *true*, то значением всего выражения с поисковым переключателем является значение выражения, указанного в разделе ELSE. Типы всех выражений, значения которых могут являться результатом выражения с поисковым переключателем, должны быть совместимыми, и типом результата является «наименьший общий» тип набора типов выражений-кандидатов на выработку результата.* Если в выражении отсутствует раздел ELSE, предполагается наличие раздела ELSE NULL.

В выражении с простым переключателем (simple_case) тип данных операнда переключателя (выражения, непосредственно следующего за ключевым словом CASE, назовем его CO – Case Operand) должен быть совместим с типом данных операнда каждого варианта (выражения, непосредственно следующего за ключевым словом WHEN; назовем WO – When Operand). Выражение с простым переключателем

```
CASE CO WHEN WO1 THEN result1
        WHEN WO2 THEN result2
        . . . . . .
```

* Для набора типов *T1*, *T2*, ..., *Tn* будем называть тип *T* наименьшим общим по приводимости, если значения каждого из типов *T1*, *T2*, ..., *Tn* неявно приводимы к типу *T*, и не существует типа *T'* такого, что значения типов *T1*, *T2*, ..., *Tn* неявно приводимы к типу *T'*, и значения типа *T'* неявно приводимы к типу *T*.

```
        WHEN WOn THEN resultn
        ELSE result
END
```

эквивалентно выражению с поисковым переключателем

```
CASE WHEN CO = WO1 THEN result1
     WHEN CO = WO2 THEN result2
        . . . . . .
     WHEN CO = WOn THEN resultn
     ELSE result
END
```

Выражение `NULLIF (V1, V2)` эквивалентно следующему выражению с переключателем:

```
CASE WHEN V1 = V2 THEN NULL ELSE V1 END.
```

Выражение `COALESCE (V1, V2)` эквивалентно следующему выражению с переключателем:

```
CASE WHEN V1 IS NOT NULL THEN V1 ELSE V2 END.
```

Выражение `COALESCE (V1, V2, . . . Vn)` для $n \geqslant 3$ эквивалентно следующему выражению с переключателем:

```
CASE WHEN V1 IS NOT NULL THEN V1 ELSE COALESCE (V2,... n) END.
```

Общая структура оператора выборки в языке SQL

Для выборки данных в прямом SQL используется оператор `SELECT`, возвращающий набор* из одной или нескольких строк одинаковой структуры и задаваемый в следующем синтаксисе:

```
SELECT [ ALL | DISTINCT ] select_item_commalist
  FROM table_reference_commalist
     [ WHERE conditional_expression ]
        [ GROUP BY column_name_commalist ]
           [ HAVING conditional_expression ]
              [ ORDER BY order_item_commalist ]
```

* Мы сознательно используем здесь термин *набор*, поскольку в общем случае результатом выполнения оператора выборки не является таблица.

Семантика оператора выборки

Для начала опишем общую схему выполнения оператора SELECT в соответствии с предписаниями стандарта.* Выполнение запроса состоит из нескольких шагов, соответствующих разделам оператора выборки. На первом шаге выполняется раздел FROM. Если список ссылок на таблицы (table_reference_commalist) этого раздела соответствует таблицам *A*, *B*, ... *C***, то в результате выполнения раздела FROM образуется таблица (назовем ее *T*), являющаяся расширенным декартовым произведением таблиц *A*, *B*, ..., *C*. Если в разделе FROM указана только одна таблица, то она же и является результатом выполнения этого раздела. Как говорилось в лекции 3, в реляционной алгебре для корректного выполнения операции взятия расширенного декартова произведения отношений в общем случае требуется применение операции переименования атрибутов. Соответствующие возможности переименования столбцов таблиц, указанных в списке раздела FROM, поддерживаются и в SQL. Альтернативный способ именования столбцов результирующей таблицы *T* основывается на использовании квалифицированных имен столбцов. Идея этого подхода (более раннего в истории SQL) заключается в том, что с любой таблицей, ссылка на которую содержится в списке раздела FROM, можно связать некоторое имя-псевдоним (в стандарте оно называется correlation name***). Тогда если с такой таблицей A связан псевдоним *Z*, то в пределах оператора выборки можно ссылаться на любой столбец *a* таблицы A по квалифицированному имени *Z.a*. Мы обсудим это подробнее в следующем подразделе. Пока же будем считать, что имена всех столбцов таблицы *T* определены и различны.

На втором шаге выполняется раздел WHERE. Условное выражение (conditional_expression) этого раздела применяется к каждой строке таблицы *T*, и результатом является таблица *T1*, содержащая те и только те строки таблицы *T*, для которых результатом вычисления условного выражения является *true*. (Заголовки таблиц *T* и *T1* совпадают.) Если раздел WHERE в операторе**** выборки отсутствует, то это трактуется как наличие раздела WHERE *true*, т. е. *T1* содержит те и только те строки, которые

* Не следует понимать эту схему таким образом, что запросы к SQL-ориентированной базе данных действительно должны выполняться именно таким образом. Более того, ни одна реализация SQL не придерживается в точности этой схеме. Но как бы реально ни выполнялся оператор выборки, результат должен быть таким же, как если бы он получался при точном следовании описываемой схеме выполнения.

** *A*, *B*, и *C* не обязаны являться *базовыми таблицами*. См. следующий подраздел.

*** Причины использования в стандарте этого термина будут более понятны после ознакомления в следующей лекции с механизмом *коррелирующих вложенных подзапросов*.

**** Заметим, что эта форма раздела WHERE в языке SQL не допускается, поскольку после ключевого слова WHERE должно следовать булевское выражение, а true булевским выражением не является.

содержатся в таблице *T*. Обратите внимание на разницу в трактовке логических выражений в операторах выборки и в табличных ограничениях целостности. Логическое выражение раздела WHERE (и раздела HAVING) оператора выборки разрешает выборку строки в том и только в том случае, когда результатом вычисления логического выражения на данной строке является *true* (значения *false* и *uknown* не являются разрешающими). Логическое выражение табличного ограничения целостности запрещает наличие строки в таблице в том и только в том случае, когда результатом вычисления логического выражения на данной строке является *false* (значения *true* и *uknown* не являются запрещающими).

Если в операторе выборки присутствует раздел GROUP BY, то он выполняется на третьем шаге. Каждый элемент списка имен столбцов (column_name_commalist), указываемого в этом разделе, должен быть одним из имен столбцов таблицы *T1*. В результате выполнения раздела GROUP BY образуется *сгруппированная таблица T2*, в которой строки таблицы *T1* расставлены в минимальное число групп, таких, что во всех строках одной группы значения столбцов, указанных в списке имен столбцов раздела GROUP BY (*столбцов группировки*), одинаковы.* Заметим, что сгруппированные таблицы не могут являться окончательным результатом оператора выборки. Они существуют только на концептуальном уровне на стадии выполнения запроса, содержащего раздел GROUP BY.

Если в операторе выборки присутствует раздел HAVING, то он выполняется на следующем шаге. Условное выражение этого раздела применяется к каждой группе строк таблицы *T2*, и результатом является сгруппированная таблица *T3*, содержащая те и только те группы строк таблицы *T2*, для которых результатом вычисления условного выражения является *true*. Условное выражение раздела HAVING строится по синтаксическим правилам, общим для всех условных выражений, но обладает той спецификой, что применяется к группам строк, а не к отдельным строкам. Поэтому предикаты, из которых строится это условное выражение, должны быть предикатами на группу в целом. В них могут использоваться имена столбцов группировки (*инварианты группы*) и так называемые агрегатные функции (COUNT, SUM, MIN, MAX, AVG) от других столбцов. Мы обсудим агрегатные функции более подробно в следующих лекциях.

При наличии в запросе раздела HAVING, которому не предшествует раздел GROUP BY, таблица *T1* рассматривается как сгруппированная таблица, состоящая из одной группы строк, без столбцов группирования. В этом случае логическое выражение раздела HAVING может состоять только из предикатов с агрегатными функциями, а результат вычисления этого раздела *T3* либо совпадает с таблицей *T1*, либо является пустым.

* Если говорить более точно, то в одной группе все строки, составленные из значений столбцов группировки, являются дубликатами.

Если в операторе выборки присутствует раздел GROUP BY, но отсутствует раздел HAVING, то это трактуется как наличие раздела HAVING *true**, т. е. *T3* содержит те и только те группы строк, которые содержатся в таблице *T2*.

После выполнения раздела WHERE (если в запросе отсутствуют разделы GROUP BY и HAVING, случай (a)) или явно или неявно заданного раздела HAVING (случай (b)) выполняется раздел SELECT. При выполнении этого раздела на основе таблицы *T1* в случае (a) или на основе сгруппированной таблицы *T3* в случае (b) строится таблица *T4*, содержащая столько строк, сколько строк или групп строк содержится в таблицах *T1* или *T3* соответственно. Число столбцов в таблице *T4* зависит от числа элементов в списке элементов выборки (select_item_commalist) и от вида элементов.

Рассмотрим, каким образом формируются значения столбцов в таблице *T4*. Элемент списка выборки может задаваться одним из двух способов:

```
select_item ::= value_expression [ [ AS ] column_name ]
              | [ correlation_name . ] *
```

Сначала обсудим первый вариант. В этом случае каждый элемент списка элементов выборки соответствует столбцу таблицы *T4*. Столбцу может быть явным образом приписано имя (когда и зачем могут использоваться имена таблицы *T4*, мы обсудим позже). Порядок формирования значения этого столбца для выделенных выше случаев (a) и (b) различается, и мы рассмотрим подобные случаи по отдельности.

В случае (a) выражение, содержащееся в элементе выборки, может содержать литеральные константы и вызовы функций со значениями соответствующих типов (в том числе ниладические). Кроме того, в выражении могут использоваться имена столбцов таблицы *T1***. Выражение вычисляется для каждой строки таблицы *T1*, и именам столбцов соответствуют значения этих столбцов в данной строке таблицы *T1*.

В случае (b), как и в случае (a), выражение, содержащееся в элементе выборки, может содержать литеральные константы и вызовы функций. Но, в отличие от случая (a), в выражение могут входить непосредственно имена только тех столбцов таблицы *T3*, которые входили в список столбцов группировки раздела GROUP BY оператора выборки. (Если сгруппированная таблица *T3* была образована за счет наличия раздела HAVING без

---

* Заметим, что эта форма раздела HAVING в языке SQL не допускается, поскольку после ключевого слова HAVING должно следовать булевское выражение, а true булевским выражением не является.

** Обратите внимание, что в выражении элемента выборки *не обязательно* должно содержаться хотя бы одно имя столбца. Допускается наличие чисто контантного выражения, значение которого будет повторяться в данном столбце всех строк таблицы TF. Кроме того, заметим, что в соответствии с определением value_expression элемент списка выборки может быть запросом, возвращающим таблицу из одной строки с одним столбцом.

присутствия раздела GROUP BY, то в выражении элемента выборки вообще нельзя непосредственно использовать имена столбцов таблицы *T3*.) Имена других столбцов таблицы *T3* могут использоваться только в конструкциях вызова агрегатных функций COUNT, SUM, MIN, MAX, AVG. Выражение вычисляется для каждой группы строк таблицы *T3*. Именам столбцов, входящих в выражение непосредственно, сопоставляются значения этих столбцов, которые соответствуют данной группе строк таблицы *T3*.

Во втором варианте спецификация элемента списка выборки вида [*Z*.]* является сокращенной формой записи списка $Z.a_1$, $Z.a_2$, ..., $Z.a_n$, где a_1, a_2, ..., a_n представляет собой полный список имен столбцов таблицы, псевдоним которой *Z*.* Следует сделать три замечания. Во-первых, для именованной таблицы, входящей в список раздела FROM только один раз, можно использовать имя таблицы вместо псевдонима. Во-вторых, во втором варианте спецификации элемента списка выборки можно опустить псевдоним только в том случае, если в разделе FROM указана только одна таблица. В-третьих, в случае (b) второй вариант спецификации элемента выборки допустим только тогда, когда все столбцы таблицы с псевдонимом *Z* входят в список столбцов группировки раздела GROUP BY.

Итак, мы получили таблицу *T4*. Если в спецификации раздела SELECT отсутствует ключевое слово DISTINCT, или присутствует ключевое слово ALL, либо отсутствуют и ALL, и DISTINCT, то *T4* является результатом выполнения раздела SELECT. В противном случае на завершающей стадии выполнения раздела SELECT в таблице *T4* удаляются строки-дубликаты.

Если в операторе выборки не содержится раздел ORDER BY, то таблица *T4* является результирующей таблицей запроса. Иначе на завершающей стадии выполнения запроса производится сортировка строк таблицы *T4* в соответствии со списком элементов сортировки (order_item_commalist) раздела ORDER BY. В стандарте SQL:1999 элемент списка элементов сортировки имеет следующую синтаксическую форму:

```
order_item ::= value_expression [ collate_clause ]
   [ { ASC | DESC } ]
```

Выполнение раздела ORDER BY производится следующим образом.** Выбирается первый элемент списка сортировки, и строки таблицы T4 расставляются в порядке возрастания (если в элементе присутствует спецификация ASC; при отсутствии спецификации ASC/DESC предполагается нали-

* Заметим, что любой элемент $Z.a_i$ этого неявно заданного списка может быть явно включен в список элементов выборки. Кстати, если в списке выборке присутствует явно или неявно заданный элемент вида *Z*.*a*, то в пределах запроса соответствующий столбец таблицы *T4* получает то же имя.

** Мы снова проигнорируем спецификацию раздела collate, связанную с использованием национальных наборов символов.

чие ASC) или в порядке убывания (при наличии спецификации DESC) в соответствии со значениями выражения, содержащегося в данном элементе, которые вычисляются для каждой строки таблицы *T4*. Далее выбирается второй элемент списка сортировки, и в соответствии со значениями заданного в нем выражения и порядка сортировки расставляются строки, которые после первого шага сортировки образовали группы с одинаковым значением выражения первого элемента списка сортировки. Операция продолжается до исчерпания списка элементов сортировки. Результирующий отсортированный список строк является окончательным результатом запроса.

В общем случае выражение, входящее в элемент списка сортировки, основывается на именах столбцов таблицы *T4* и именах столбцов таблицы, над которой вычислялся раздел SELECT (*T1* или *T3*). Идея состоит в том, что если некоторое выражение могло бы быть использовано в элементе списка выборки, то его можно использовать в элементе списка сортировки. В стандарте SQL:1999 присутствует ряд чисто технических ограничений на вид выражений, допустимых в элементах списка сортировки, если в запросе присутствуют разделы GROUP BY и/или HAVING и если в разделе SELECT присутствует спецификация DISTINCT. Но в любом случае это выражение может иметь вид *a*, где *a* – имя столбца таблицы *T4*.

Заметим, что в предыдущих версиях стандарта языка SQL, включая SQL/92, элемент списка сортировки определялся следующим синтаксическим правилом:

```
order_item ::= { column_name | unsigned_integer }
   [ { ASC | DESC } ]
```

В качестве имени столбца (column_name) можно было использовать любое имя, вводимое для столбца таблицы *T4* в элементе списка выборки. Вместо имени столбца можно было использовать его порядковый номер (unsigned_integer) в списке элементов выборки раздела SELECT. Как мы видели, в новом стандарте вторая возможность исключена. Доводом является не тот факт, что использование номеров столбцов противоречит реляционной модели. Использование номеров столбцов запрещено, поскольку не давало возможности применять в элементах списка сортировки выражения. Тем не менее, по нашему мнению, возможность использования номеров столбцов в течение долгого времени будет продолжать поддерживаться в коммерческих реализациях SQL, поскольку она применяется во многих существующих приложениях.

Ссылки на таблицы раздела FROM

Напомним, что раздел FROM оператора выборки определяется синтаксическим правилом

```
FROM table_reference_commalist
```

Рассмотрим более подробно, какой вид могут иметь элементы этого списка. Для начала приведем полный набор синтаксических правил SQL:1999, определяющий table_reference.*

```
table_reference ::= table_primary | joined_table
table_primary ::= table_or_query_name [ [ AS ] correlation_name
          [ (derived_column_list) ] ]
        | derived_table [ [ AS ] correlation_name
          [ (derived_column_list) ] ]
        | lateral_derived_table [ [ AS ] correlation_name
          [ (derived_column_list) ] ]
        | collection_derived_table [ [ AS ] correlation_name
          [ (derived_column_list) ] ]
        | ONLY (table_or_query_name)[ [ AS ] correlation_name
          [ (derived_column_list) ] ] | (joined_table)
table_or_query_name ::= { table_name | query_name }
derived_table ::= (query_expression)
lateral_derived_table ::= LATERAL (query_expression)
collection_derived_table ::= UNNEST
        (collection_value_exression) [ WITH ORDINALITY ]
```

Мы отложим до следующих лекций обсуждение порождаемых таблиц с горизонтальной связью (lateral_derived_table) и «соединенных таблиц» (joined_table). Кроме того, мы не будем рассматривать в этом курсе конструкции collection_derived_table и ONLY (table_or_query_name), поскольку они относятся к объектным расширениям языка SQL, которые в данном курсе подробно не рассматриваются (на неформальном уровне объектно-реляционный подход обсуждается в последней лекции этого курса). Но даже при таких самоограничениях для дальнейшего продвижения нам придется определить несколько дополнительных синтаксических конструкций языка SQL.

Табличное выражение, спецификация запроса и выражение запросов

Табличным выражением (table_expression) называется конструкция

```
table_expression ::= FROM table_reference_commalist
     [ WHERE conditional_expression ]
       [ GROUP BY column_name_commalist ]
         [ HAVING conditional_expression ]
```

* В связи с введением в стандарте SQL:2003 конструктора типов мультимножеств, в качестве элемента списка ссылок на таблицы раздела FROM теперь можно использовать и выражение со значением-мультимножеством. Однако в этом курсе мы не будем подробно рассматривать эту возможность.

Спецификацией запроса (query_specification) называется конструкция

```
query_specification SELECT [ ALL | DISTINCT ]
     select_item_commalist table_expression
```

Наконец, выражением запросов (query_expression) называется конструкция

```
query_expression ::= [ with_clause ] query_expression_body
query_expression_body ::= { non_join_query_expression
                          | joined_table }
non_join_query_expression ::= non_join_query_term
                    | query_expression_body
                      { UNION | EXCEPT }[ ALL | DISTINCT ]
                        [ corresponding_spec ] query_term
query_term ::= non_join_query_term | joined_table
non_join_query_term ::= non_join_query_primary
                  | query_term INTERSECT [ ALL | DISTINCT ]
                    [ corresponding_spec ] query_primary
query_primary ::= non_join_query_primary | joined_table
non_join_query_primary ::= simple_table
                          | (non_join_query_expression)
simple_table ::= query_specification
             | table_value_constructor
             | TABLE table_name
corresponding_spec ::= CORRESPONDING
                       [ BY column_name_comma_list ]
```

Если не обращать внимания на не обсуждавшиеся пока конструкции joined_table и table_value_constructor, синтаксические правила показывают, что выражение запросов строится из выражений, значениями которых являются таблицы, с использованием «теоретико-множественных»* операций UNION (объединение), EXCEPT (вычитание) и INTERSECT (пересечение). Операция пересечения является «мультипликативной» и обладает более высоким приоритетом, чем «аддитивные» операции объединения и вычитания. Вычисление выражения производится слева направо с учетом приоритетов операций и круглых скобок. При этом действуют следующие правила.

- Если выражение запросов не включает ни одной теоретико-множественной операции, то результатом вычисления выражения запросов яв-

* Мы использовали кавычки, поскольку таблицы, к которым применяются операции, в общем случае могут содержать строки-дубликаты, т.е. являться *мультимножествами*.

ляется результат вычисления простой или соединенной таблицы.

- Если в терме (non_join_query_term) или выражении запросов (non_join_query_expression) без соединения присутствует теоретико-множественная операция, то пусть *T1*, *T2* и *TR* обозначают соответственно первый операнд, второй операнд и результат терма или выражения соответственно, а OP — используемую теоретико-множественную операцию.
- Если в операции присутствует спецификация CORRESPONDING, то:
  (a) если присутствует конструкция BY column_name_comma_list, то все имена в этом списке должны быть различны, и каждое имя должно являться одновременно именем некоторого столбца таблицы *T1* и именем некоторого столбца таблицы *T2*, причем типы этих столбцов должны быть совместимыми; обозначим данный список имен через *SL*;
  (b) если список соответствия столбцов не задан, пусть *SL* обозначает список имен столбцов, являющихся именами столбцов и в *T1*, и в *T2*, в том порядке, в котором эти имена фигурируют в *T1*;
  (c) вычисляемые терм или выражение запросов без соединения эквивалентны выражению (SELECT *SL* FROM *T1*) OP (SELECT *SL* FROM *T2*), не включающему спецификацию CORRESPONDING.
- При отсутствии в операции спецификации CORRESPONDING операция выполняется таким образом, как если бы эта спецификация присутствовала и включала конструкцию BY column_name_comma_list, в которой были бы перечислены все столбцы таблицы *T1*.*
- При выполнении операции OP две строки *s1* с именами столбцов *c1*, *c2*, ..., *cn* и *s2* с именами столбцов *d1*, *d2*, ..., *dn* считаются строками-дубликатами, если для каждого *I* (*i* = 1, 2, ..., *n*) либо *ci* и *di* не содержат NULL, и (*ci* = *di*) = ***true***, либо и *ci*, и *di* содержат NULL.
- Если в операции OP не задана спецификация ALL, то в *TR* строки-дубликаты удаляются.
- Если спецификация ALL задана, то пусть *s* — строка, являющаяся дубликатом некоторой строки *T1*, или некоторой строки *T2*, или обеих; пусть m — число дубликатов *s* в *T1*, а *n* — число дубликатов *s* в *T2*. Тогда:
  - если указана операция UNION, то число дубликатов s в TR равно *m* + *n*;
  - если указана операция EXCEPT, то число дубликатов s в TR равно *max* ((*m*-*n*),0);
  - если указана операция INTERSECT, то число дубликатов *s* в *TR* равно *min* (*m*,*n*).

* Другими словами, при отсутствии спецификации CORRESPONDING требуется, чтобы заголовки таблиц-операндов совпадали за исключением, возможно, порядка следования столбцов.

** С учетом возможности неявного приведения типов.

Раздел WITH выражения запросов

Как видно из синтаксиса выражения запросов, в этом выражении может присутствовать раздел WITH. Он задается в следующем синтаксисе:

```
with_clause ::= WITH [ RECURSIVE ] with_element_comma_list
with_element ::= query_name [ (column_name_list) ]
            AS (query_expression) [ search_or_cycle_clause ]
```

Общую форму раздела WITH мы обсудим в следующих лекциях, когда будем рассматривать средства формулировки рекурсивных запросов. Пока ограничимся случаем, когда в разделе WITH отсутствуют спецификация RECURSIVE и search_or_cycle_clause. Тогда конструкция

```
WITH query_name (c1, c2, … cn) AS (query_exp_1) query_exp_2
```

означает, что в любом месте выражения запросов query_exp_2, где допускается появление ссылки на таблицу, можно использовать имя query_name. Можно считать, что перед выполнением query_exp_2 происходит выполнение query_exp_1, и результирующая таблица с именами столбцов *c1*, *c2*, … *cn* сохраняется под именем query_name. Как мы увидим позже, в этом случае раздел WITH фактически служит для локального определения представляемой таблицы (VIEW).

Конструкторы значения строки и таблицы

Чтобы завершить обсуждение выражений запросов (с учетом того, что конструкция соединенных таблиц (joined_table) отложена на следующие лекции), нам осталось рассмотреть конструкции table_value_constructor и TABLE table_name.

В определении конструктора значения-таблицы используется конструктор значения-строки, который строит упорядоченный набор скалярных значений, представляющий строку (возможно и использование подзапроса*):

```
row_value_constructor ::= row_value_constructor_element
      | [ ROW ] (row_value_constructor_element_comma_list)
      | row_subquery
row_value_constructor_element ::= value_expression | NULL | DEFAULT
```

Заметим, что значение элемента по умолчанию можно использовать только в том случае, когда конструктор значения-строки применяется в

* В следующей лекции мы более подробно обсудим подзапросы. Пока заметим, что row_subquery – это запрос, результирующая таблица которого состоит из одной строки.

операторе INSERT (тогда этим значением будет значение по умолчанию соответствующего столбца).

Конструктор значения-таблицы производит таблицу на основе заданного набора конструкторов значений-строк:

```
table_value_constructor ::= VALUES
   row_value_constructor_comma_list
```

Конечно, для того чтобы корректно построить таблицу, требуется, чтобы строки, производимые всеми конструкторами строк, были одной и той же степени и чтобы типы (или домены) соответствующих столбцов являлись приводимыми.

Наконец, конструкция TABLE table_name является сокращенной формой записи выражения SELECT * FROM table_name.

Ссылки на базовые, представляемые и порождаемые таблицы

Теперь мы можем завершить обсуждение разновидностей ссылок на таблицу в разделе FROM. Для удобства повторим синтаксические правила (опустив конструкции, рассмотрение которых отложено на следующие лекции или выходит за пределы материала данного курса):

```
table_reference ::= table_primary
table_primary ::= table_or_query_name [ [ AS ] correlation_name
                  [ (derived_column_list) ] ]
          | derived_table [ AS ] correlation_name
                  [ (derived_column_list) ]
table_or_query_name ::= { table_name | query_name }
derived_table ::= (query_expression)
```

Итак, в самом простом случае в качестве ссылки на таблицу используется имя таблицы (базовой или представляемой) или имя запроса, присоединенного к данному запросу с помощью раздела WITH. В другом случае (derived_table) порождаемая таблица задается выражением запроса, заключенным в круглые скобки. Явное указание имен столбцов результата запроса из раздела WITH или порождаемой таблицы требуется в том случае, когда эти имена не выводятся явно из соответствующего выражения запроса. Обратите внимание, что в таких случаях в соответствующем элементе списка раздела FROM должен указываться псевдоним (correlation_name), потому что иначе таблица была бы вообще лишена имени. Можно считать, что выражение запроса вычисляется и сохраняется во временной таблице при обработке раздела FROM.

Возможно, некоторых читателей смутила рекурсивная природа синтаксических определений, приведенных в этом подразделе. Чтобы определить понятие ссылки на таблицу в разделе FROM оператора выборки, который опирается на спецификацию запроса, нам пришлось ввести более общее понятие выражения запросов, в определении которого используется спецификация запроса. Да, действительно, многие синтаксические конструкции SQL определяются рекурсивно. Но эта рекурсия никогда не приводит к зацикливанию. В частности, раскрутка рекурсии операторов выборки основывается на базовой, не выделяемой отдельными синтаксическими правилами форме, в которой в разделе FROM указываются только имена базовых таблиц.

Представляемые таблицы, или представления (VIEW)

Еще одним примером рекурсивности спецификаций языка SQL является то, что в конце этой лекции мы вынуждены прервать обсуждение оператора выборки и ввести понятие представляемой таблицы, или представления, которую можно использовать в операторе выборки наряду с базовыми таблицами. Только после этого можно будет считать обсуждение ссылок на таблицы в разделе FROM условно завершенным. Итак, оператор создания представления в общем случае определяется следующими синтаксическими правилами:

```
create_view ::= CREATE [ RECURSIVE ] VIEW table_name
      [ column_name_comma_list ]
   AS query_expression
   [ WITH [ CASCADED | LOCAL ] CHECK OPTION ]
```

Рекурсивные представления (такие, в определении которых присутствует ключевое слово RECURSIVE) и необязательный раздел WITH CHECK OPTION мы обсудим в следующих лекциях (пока лишь заметим, что этот раздел связан с особенностями выполнения операций обновления базы данных через представления). Здесь мы кратко рассмотрим только простую форму представлений, определяемых по следующим правилам:

```
create_view ::= CREATE VIEW table_name
      [ column_name_comma_list ]
   AS query_expression
```

Имя таблицы, задаваемое в определении представления, существует в том же пространстве имен, что и имена базовых таблиц, и, следовательно, должно отличаться от всех имен таблиц (базовых и представляемых), созданных тем же пользователем. Если имя представления встречается в

разделе FROM какого-либо оператора выборки, то вычисляется выражение запроса, указанное в разделе AS, и оператор выборки работает с результирующей таблицей этого выражения запроса.* Явное указание имен столбцов представляемой таблицы требуется в том случае, когда эти имена не выводятся из соответствующего выражения запроса.

Как и для всех других вариантов оператора CREATE, для CREATE VIEW имеется обратный оператор DROP VIEW table_name, выполнение которого приводит к отмене определения представления (реально это выражается в удалении данных о представлении из таблиц-каталогов базы данных). После выполнения операции пользоваться представлением с данным именем становится невозможно.**

Заключение

В ходе чтения лекций, посвященных оператору SELECT языка SQL, мне неоднократно случалось слышать жалобы студентов на сухость начального материала и отсутствие иллюстрирующих примеров. Однако я не встречал ни одного учебного пособия по языку SQL, основанного на примерах (среди многочисленных изданий типа «SQL за 24 часа», «SQL для чайников» и даже «SQL для идиотов»), который действительно давал бы представление об SQL как языке, а не служил инструкцией армейского типа.

Сложность организации оператора выборки не позволяет сразу начинать с полноценных примеров, а для демонстрации примеров промежуточных конструкций требуется создание неприемлемо громоздкого контекста. Поэтому могу лишь принести извинения за некоторую сухость этой лекции.

С другой стороны, теперь мы уже вплотную подошли к тому этапу, на котором возможно использование иллюстраций, и в следующих лекциях их будет достаточно, хотя проиллюстрировать все интересные разновидности оператора SELECT все равно не представляется возможным, поскольку число вариантов близко к астрономическому.

* По крайней мере, так это следует понимать в соответствии с семантикой представлений в языке SQL. При реальной обработке запросов над представлениями такая явная «материализация» представления выполняется крайне редко. Вместо этого используется техника подстановки тела представления в тело запроса с гарантией того, что результат модифицированного запроса будет в точности таким же, что и результат исходного запроса над материализованным представлением. Но это уже относится к тематике оптимизации SQL-запросов, выходящей за пределы этого курса.

** Конструкция ALTER VIEW в языке SQL не поддерживается.

Лекция 14. Язык баз данных SQL: предикаты раздела WHERE оператора SELECT

В этой лекции мы продолжим рассматривать механизм выборки данных языка SQL – оператора SELECT. Лекция целиком посвящена видам условных выражений, которые могут содержаться в разделе WHERE оператора выборки. Определяются и иллюстрируются на примерах запросов все виды предикатов, специфицированных в стандарте SQL:1999.

Ключевые слова: раздел WHERE оператора выборки, логическое выражение раздела WHERE, предикат, предикат сравнения, сравнение символьных строк, сравнение битовых строк, сравнение значений типа дата-время, сравнение значений анонимного строкового типа, спецификация DISTINCT, корреляционные подзапросы, полусоединение (semijoin), уточненная семантика оператора выборки, предикат between, предикат null, семантика предиката null, предикат in, предикат like, раздел ESCAPE, предикат similar, регулярные выражения предиката similar, набор символов (regular character set), идентификатор набора символов (regular charset id), функция SUBSTRING ... SIMILAR ... ESCAPE, предикат exists, предикат unique, предикат overlaps, предикат сравнения с квантором, предикат сравнения с квантором всеобщности, предикат сравнения с квантором существования, предикат match, тип сопоставления SIMPLE, тип сопоставления PARTIAL, тип сопоставления FULL, предикат distinct.

Введение

Конструкции оператора SELECT языка SQL в значительной степени ортогональны. В частности, выбор способа указания ссылки на таблицы в разделе FROM никак не влияет на выбор варианта формирования условия выборки в разделе WHERE. Это полезное свойство языка позволяет нам абстрагироваться от обсуждавшегося в предыдущей лекции многообразия способов указания ссылки на таблицу и сосредоточиться на возможностях формирования запросов при использовании различных предикатов, допускаемых стандартом SQL:1999 в логических выражениях раздела WHERE.

В стандарте SQL:1999 специфицированы 12 разновидностей предикатов, причем некоторые из них в действительности представляют собой семейства (например, под общим названием *предиката сравнения* скрываются шесть видов предикатов). Набор допустимых предикатов в SQL явно избыточен, но тем не менее в языке SQL имеется явная тенденция расширения этого набора. В частности, в SQL:2003 в связи с введением генератора типов мультимножеств в дополнение ко всем разновидностям предикатов

SQL:1999 появилось три новых вида предикатов: предикаты для проверки того, что заданное значение является элементом мультимножества (MEMBER); что одно мультимножество входит в другое мультимножество (SUBMULTISET) и что мультимножество не содержит дубликаты (IS A SET). В этом курсе мы не приводим подробного описания этих видов предикатов по нескольким причинам:

- введение конструктора типов мультимножеств в стандарте SQL:2003 не означает, что достигнута общая цель разработчиков стандарта SQL по обеспечению полного набора типов коллекций; по всей видимости, в будущих версиях стандарта появятся дополнительные конструкторы типов коллекций, и набор видов предикатов изменится;
- предикаты с мультимножествами трудно пояснять и иллюстрировать в отрыве от других объектно-реляционных средств языка SQL;
- включение подобного материала в данную лекцию заметно увеличило бы ее объем и затруднило понимание более традиционных конструкций.

В лекции содержится много примеров запросов с использованием различных видов предикатов. Для полного усвоения материала требуется тщательно проанализировать эти примеры.

Логические выражения раздела WHERE

Синтаксически логическое выражение раздела WHERE определяется как булевское выражение (boolean_value_expression), правила построения которого обсуждались в предыдущей лекции. Основой логического выражения являются предикаты. Предикат позволяет специфицировать условие, результатом вычисления которого может быть *true*, *false* или *unknown*. В языке SQL:1999 допустимы следующие предикаты:*

```
predicate ::= comparison_predicate
            | between_predicate
            | null_predicate
            | in_predicate
            | like_predicate
            | similar_predicate
            | exists_predicate
            | unique_predicate
            | overlaps_predicate
            | quantified_comparison_predicate
            | match_predicate
            | distinct_predicate
```

* Мы не обсуждаем в этом курсе предикаты, основанные на использовании выражений типа мультимножества, которые были введены в стандарте SQL:2003.

Далее мы будем последовательно обсуждать разные виды предикатов и приводить примеры запросов с использованием базы данных СЛУЖАЩИЕ-ОТДЕЛЫ-ПРОЕКТЫ, определения таблиц которой на языке SQL были приведены в лекции 12. Для удобства повторим здесь структуру таблиц.

EMP:

EMP_NO : EMP_NO
EMP_NAME : VARCHAR
EMP_BDATE : DATE
EMP_SAL : SALARY
DEPT_NO : DEPT_NO
PRO_NO : PRO_NO

DEPT:

DEPT_NO : DEPT_NO
DEPT_NAME : VARCHAR
DEPT_EMP_NO : INTEGER
DEPT_TOTAL_SAL : SALARY
DEPT_MNG : EMP_NO

PRO:

PRO_NO : PRO_NO
PRO_TITLE : VARCHAR
PRO_SDATE : DATEP
RO_DURAT : INTERVAL
PRO_MNG : EMP_NO
PRO_DESC : CLOB

Столбцы EMP_NO, DEPT_NO и PRO_NO являются первичными ключами таблиц EMP, DEPT и PRO соответственно. Столбцы DEPT_NO и PRO_NO таблицы EMP являются внешними ключами, ссылающимися на таблицы DEPT и PRO соответственно (DEPT_NO указывает на отделы, в которых работают служащие, а PRO_NO – на проекты, в которых они участвуют; оба столбца могут принимать неопределенные значения). Столбец DEPT_MNG является внешним ключом таблицы DEPT (DEPT_MNG указывает на служащих, которые исполняют обязанности руководителей отделов; у отдела может не быть руководителя, и один служащий не может быть руководителем

двух или более отделов). Столбец PRO_MNG является внешним ключом таблицы PRO (PRO_MNG указывает на служащих, которые являются менеджерами проектов, у проекта всегда есть менеджер, и один служащий не может быть менеджером двух или более проектов).

Предикат сравнения

Этот предикат предназначен для спецификации сравнения двух строчных значений. Синтаксис предиката следующий:

```
comparison_predicate ::=
     row_value_constructor comp_op row_value_constructor
comp_op ::= = | <> ("неравно")| < | >
     | <= "меньше или равно"| >= "больше или равно"
```

Строки, являющиеся операндами операции сравнения, должны быть одинаковой степени. Типы данных соответствующих значений строк-операндов должны быть совместимы.

Пусть *X* и *Y* обозначают соответствующие элементы строк-операндов, а *xv* и *yv* – их значения. Тогда:

(a) если *xv* и/или *yv* являются неопределенными значениями, то значение условия *X* comp_op *Y* —***unknown***;

(b) в противном случае значением условия *X* comp_op *Y* является ***true*** или ***false*** в соответствии с естественными правилами применения операции сравнения.

При этом:

- Числа сравниваются в соответствии с правилами алгебры.
- Сравнение двух символьных строк производится следующим образом:
  - если длина строки *X* не равна длине строки *Y*, то для выравнивания длин строк более короткая строка расширяется ***символами набивки (pad symbol)***; если для используемого набора символов порядок сортировки явным образом не специфицирован, то в качестве символа набивки используется пробел;
  - далее производится лексикографическое сравнение строк в соответствии с предопределенным или явно определенным порядком сортировки символов.
- Сравнение двух битовых строк *X* и *Y* основано на сравнении соответствующих бит. Если X_i и Y_i – значения *i*-тых бит *X* и *Y* соответственно и если *lx* и *ly* обозначает длину в битах *X* и *Y* соответственно, то:
  - *X* равно *Y* тогда и только тогда, когда *lx* = *ly* и $X_i = Y_i$ для всех *i*;
  - *X* меньше *Y* тогда и только тогда, когда (a) *lx* < *ly* и $X_i = Y_i$ для всех *i* меньших или равных *lx*, или (b) $X_i = Y_i$ для всех $i < n$ и $X_n = 0$, а $Y_n = 1$ для некоторого *n* меньшего или равного min (*lx*, *ly*).

- Сравнение двух значений типа дата-время производится в соответствии с видом интервала, который получается при вычитании второго значения из первого. Пусть *X* и *Y* – сравниваемые значения, а *H* – наименее значимое поле даты-времени *X* и *Y*. Результат сравнения *X* comp_op *Y* определяется как (*X* - *Y*) *H* comp_ op INTERVAL (0) *H*. (Два значения типа дата-время сравнимы только в том случае, если они содержат одинаковый набор полей даты-времени.)
- Сравнение двух значений анонимного строкового типа производится следующим образом. Пусть *Rx* и *Ry* обозначают строки-операнды, а Rx_i и Ry_i – *i*-тые элементы *Rx* и *Ry* соответственно. Вот как определяется результат сравнения *Rx* comp_op *Ry*:
  - *Rx* = *Ry* есть *true* тогда и только тогда, когда Rx_i = Ry_i есть *true* для всех *i*;
  - *Rx* <> *Ry* есть *true* тогда и только тогда, когда Rx_i <> Ry_i есть *true* для некоторого *i*;
  - *Rx* < *Ry* есть *true* тогда и только тогда, когда Rx_i = Ry_i есть *true* для всех *i* < *n*, и Rx_n < Ry_n есть *true* для некоторого *n*;
  - *Rx* > *Ry* есть *true* тогда и только тогда, когда Rx_i = Ry_i есть *true* для всех *i* < *n*, и Rx_n > Ry_n есть *true* для некоторого n;
  - *Rx* <= *Ry* есть *true* тогда и только тогда, когда *Rx* = *Ry* есть *true* или *Rx* < *Ry* есть *true*;
  - *Rx* >= *Ry* есть *true* тогда и только тогда, когда *Rx* = *Ry* есть *true* или *Rx* > *Ry* есть *true*;
  - *Rx* = *Ry* есть *false* тогда и только тогда, когда *Rx* <> *Ry* есть *true*;
  - *Rx* <> *Ry* есть *false* тогда и только тогда, когда *Rx* = *Ry* есть *true*;
  - *Rx* < *Ry* есть *false* тогда и только тогда, когда *Rx* >= *Ry* есть *true*;
  - *Rx* > *Ry* есть *false* тогда и только тогда, когда *Rx* <= *Ry* есть *true*;
  - *Rx* <= *Ry* есть *false* тогда и только тогда, когда *Rx* > *Ry* есть *true*;
  - *Rx* >= *Ry* есть *false* тогда и только тогда, когда *Rx* < *Ry* есть *true*;
  - *Rx* comp_op *Ry* есть *unknown* тогда и только тогда, когда *Rx* comp_op *Ry* не есть *true* или *false*.

Примеры запросов с использованием предиката сравнения

Пример 14.1. Найти номера отделов, в которых работают служащие с фамилией 'Smith'.

```
SELECT DISTINCT EMP.DEPT_NO
FROM EMP
WHERE EMP.EMP_NAME = 'Smith';
```

Мы добавили спецификацию DISTINCT в раздел SELECT, потому что в одном отделе могут работать несколько служащих с фамилией 'Smith', а их число нас в данном случае не интересует. Кстати, если бы нас интересовало

число служащих с фамилией 'Smith' в каждом отделе, где такие служащие работают, то следовало бы, например, написать такой запрос (**пример 14.1а**):

```
SELECT EMP.DEPT_NO, COUNT(*)
FROM EMP
WHERE EMP.NAME = 'Smith'
GROUP BY EMP.DEPT_NO;
```

В этом варианте запроса спецификация DISTINCT не требуется, поскольку в запросе содержится раздел GROUP BY, группировка производится в соответствии со значениями столбца EMP.DEPT_NO, и строка результата соответствует одной группе.

Пример 14.2. Найти номера, имена и номера отделов служащих, родившихся после 15 апреля 1965 г.

```
SELECT EMP.EMP_NO, EMP.EMP_NAME, EMP.DEPT_NO
FROM EMP
WHERE EMP.EMP_BDATE > DATE '1965-04-15';
```

В результате этого запроса дубликатов быть не может, поскольку в список выборки включен столбец, являющийся первичным ключом таблицы EMP. Должно быть ясно, что по этой причине все строки результата будут различными.

Пример 14.3. Найти номера, имена и номера отделов служащих, размер заработной платы которых составляет больше одной десятой объема фонда заработной платы их отделов.

```
SELECT EMP.EMP_NO, EMP.EMP_NAME, EMP.DEPT_NO
FROM EMP
WHERE EMP.EMP_SAL > 0.1 *
   (SELECT DEPT_TOTAL_SAL
   FROM DEPT
   WHERE DEPT.DEPT_NO = EMP.DEPT_NO);
```

В этом SQL-запросе имеются две интересные особенности, которые мы до сих пор не обсуждали. Во-первых, второй операнд операции сравнения содержит подзапрос, возвращающий единственное значение, поскольку логическое выражение раздела WHERE этого подзапроса состоит из условия, однозначно определяющего значение первичного ключа таблицы DEPT. Во-вторых, в условии раздела WHERE подзапроса используется ссылка на столбец таблицы EMP, указанной в разделе FROM «внешнего» запроса. Подобные подзапросы в терминологии SQL традиционно называ-

ются *корреляционными*, и их следует понимать следующим образом*.

При выполнении внешнего запроса последовательно, строка за строкой, в некотором порядке, определяемом системой, производится проверка соответствия строк результирующей таблицы раздела FROM условию раздела WHERE. Если это условие включает *корреляционные* подзапросы, то внутри каждого из этих подзапросов ссылка на столбец внешней таблицы трактуется как ссылка на столбец текущей строки данной таблицы во внешнем цикле. Естественно, условие WHERE любого подзапроса может включать более глубоко вложенные подзапросы, на которые распространяется то же правило корреляции с внешними таблицами.

Кстати, эквивалентная формулировка на языке SQL примера 14.3 выглядит следующим образом (**пример 14.3а**):

```
SELECT EMP.EMP_NO, EMP.EMP_NAME, EMP.DEPT_NO
FROM EMP, DEPT
WHERE EMP.DEPT_NO = DEPT.DEPT_NO AND
   EMP.EMP_SAL > 0.1 * DEPT.TOTAL_SAL;
```

Мы видим, что в терминах реляционной алгебры этот запрос представляет собой ограничение (по условию EMP.EMP_SAL > 0.1 * DEPT.TOTAL_SAL) *эквисоединения* таблиц EMP и DEPT (по условию EMP.DEPT_NO = DEPT.DEPT_NO). Подобную операцию часто называют *полусоединением* (*semi-join*), поскольку в результирующей таблице используются столбцы только одного из операндов операции эквисоединения. Мы привели вторую формулировку запроса, преследуя две цели: (1) продемонстрировать, каким образом предикат сравнения можно использовать для задания условия соединения, и (2) показать, что запросы, содержащие вложенные запросы, часто могут быть переформулированы в запросы с соединениями.

Пример 14.4. Найти номера, имена, номера отделов и имена руководителей отделов служащих, размер заработной платы которых меньше 15000 руб.

```
SELECT EMP1.EMP_NO, EMP1.EMP_NAME, EMP1.DEPT_NO, EMP2.EMP_NAME
FROM EMP AS EMP1, EMP AS EMP2, DEPT
WHERE EMP1.EMP_SAL < 15000.00 AND
   EMP1.DEPT_NO = DEPT.DEPT_NO AND
   DEPT.DEPT_MNG = EMP2.EMP_NO;
```

* Здесь снова идет речь о *семантике* выполнения оператора SELECT. В стандарте, естественно, не требуется, чтобы в реализации языка запросы с корреляционными подзапросами выполнялись в точности так, как описывается ниже. Суть в том, что какой бы реальный алгоритм выполнения такого запроса не использовался, результат выполнения должен быть точно таким же, как если бы запрос выполнялся по описываемой схеме.

Этот запрос представляет собой эквисоединение ограничения таблицы EMP (по условию EMP_SAL < 15000.00) с таблицами DEPT и EMP (по условиям EMP.DEPT_NO = DEPT.DEPT_NO и DEPT.DEPT_MNG = EMP2.EMP_NO соответственно). Таблица EMP участвует в качестве операнда операции эквисоединения два раза. Поэтому в разделе FROM ей присвоены два псевдонима – EMP1 и EMP2. Следуя предписанному стандартом порядку выполнения запроса, можно считать, что введение этих псевдонимов обеспечивает переименование столбцов таблицы EMP, требуемое для выполнения раздела FROM с образованием расширенного декартова произведения таблиц-операндов.* Заметим также, что в данном случае мы имеем дело с *полным* эквисоединением трех таблиц (а не с *полу*соединением), поскольку в списке выборки присутствуют имена столбцов каждой из них.

Покажем способ формулировки этого запроса с использованием вложенного подзапроса в качестве элемента списка выборки (**пример 14.4а**):

```
SELECT EMP.EMP_NO, EMP.EMP_NAME, EMP.DEPT_NO,
   (SELECT EMP_NAME
      FROM EMP
      WHERE EMP_NO = DEPT_MNG)
FROM EMP, DEPT
WHERE EMP.EMP_SAL < 15000.00 AND
   EMP.DEPT_NO = DEPT.DEPT_NO;
```

Как показывает последний пример, в условии выборки подзапроса, участвующего в списке выборки, можно использовать имена столбов таблиц внешнего запроса. Из этой возможности языка SQL видно, что в разделе «Общие синтаксические правила построения скалярных выражений» предыдущей лекции для облегчения понимания материала мы немного исказили семантику оператора выборки. Там было сказано следующее: «После выполнения раздела WHERE (если в запросе отсутствуют разделы GROUP BY и HAVING, случай (a)) или выполнения явно или неявно заданного раздела HAVING (случай (b)) выполняется раздел SELECT. При выполнении этого раздела на основе таблицы *T1* в случае (a) или на основе сгруппированной таблицы *T3* в случае (b) строится таблица *T4*, содержащая столько строк, сколько строк или групп строк содержится в таблицах *T1* или *T3* соответственно». В действительности, в общем случае очередная строка таблицы *T4* должна строиться в тот момент, когда очередная строка или группа строк заносится в таблицу *T1* или *T3* соответственно.

* Кстати, в этом случае можно было бы обойтись введением одного псевдонима, оставив в качестве неявного второго псевдонима имя таблицы – EMP.

Предикат `between`

Предикат позволяет специфицировать условие вхождения в диапазон значений. Операндами являются строки:

```
between_predicate ::=
     row_value_constructor [ NOT ] BETWEEN
     row_value_constructor AND row_value_constructor
```

Все три строки-операнды должны иметь одну и ту же степень. Типы данных соответствующих значений строк-операндов должны быть совместимыми.

Пусть *X*, *Y* и *Z* обозначают первый, второй и третий операнды. Тогда по определению выражение *X* `NOT BETWEEN` *Y* `AND` *Z* эквивалентно выражению `NOT (`*X* `BETWEEN` *Y* `AND` *Z*`)`. Выражение *X* `BETWEEN` *Y* `AND` *Z* по определению эквивалентно булевскому выражению *X* `>=` *Y* `AND` *X* `<=` *Z*.

Примеры запросов с использованием предиката `between`

Пример 14.5. Найти номера, имена и размер зарплаты служащих, получающих зарплату в размере от 12000 до 15000 руб.

```
SELECT EMP_NO, EMP_NAME, EMP_SAL
FROM EMP
WHERE EMP_SAL BETWEEN 12000.00 AND 15000.00;
```

Пример 14.6. Найти номера, имена и размер зарплаты служащих, получающих зарплату, размер которой не меньше средней зарплаты служащих своего отдела и не больше зарплаты руководителя отдела.

```
SELECT EMP_NO, EMP_NAME, EMP_SAL
FROM EMP
WHERE EMP_SAL BETWEEN
   (SELECT AVG(EMP1.EMP_SAL)
      FROM EMP EMP1
      WHERE EMP.DEPT_NO = EMP1.DEPT_NO)
         AND
   (SELECT EMP1.EMP_SAL
      FROM EMP EMP1
      WHERE EMP1.EMP_NO =
         (SELECT DEPT.DEPT_MNG
            FROM DEPT
            WHERE DEPT.DEPT_NO = EMP.DEPT_NO));
```

В этом запросе можно выделить три интересных момента. Во-первых, диапазон значений предиката BETWEEN задан двумя подзапросами, результатом каждого из которых является единственное значение. Первый подзапрос выдает единственное значение, поскольку в списке выборки содержится агрегатная функция (AVG) и отсутствует раздел GROUP BY, а второй – потому что в его разделе WHERE присутствует условие, задающее единственное значение первичного ключа. Во-вторых, в обоих подзапросах таблица EMP получает псевдоним EMP1 (в формулировке этого запроса мы старались использовать как можно меньше вспомогательных идентификаторов). Поскольку подзапросы выполняются независимо один от другого, использование общего имени не вызывает проблем. Наконец, в условии второго подзапроса присутствует более глубоко вложенный подзапрос, и в условии его раздела WHERE используется ссылка на столбец таблицы из самого внешнего раздела FROM.

Предикат is null

Предикат is null позволяет проверить, являются ли неопределенными значения всех элементов строки-операнда:

```
null_predicate ::= row_value_constructor IS [ NOT ] NULL
```

Пусть *X* обозначает строку-операнд. Если значения всех элементов *X* являются неопределенными, то значением условия *X* IS NULL является *true*; иначе – *false*. Если ни у одного элемента *X* значение не является неопределенным, то значением условия *X* IS NOT NULL является *true*; иначе – *false*.

Замечание: условие *X* IS NOT NULL имеет то же значение, что условие NOT *X* IS NULL для любого *X* в том и только в том случае, когда степень *X* равна 1. Полная семантика предиката null приведена в таблице 14.1.

Табл.14.1.

Вид условия / Вид операнда	X IS X NULL	IS NOT NULL	NOT X IS NULL	NOT X IS NOT NULL
Степень 1: значение NULL	*true*	*false*	*false*	*true*
Степень 1: значение отлично от NULL	*false*	*true*	*true*	*false*
Степень > 1: у всех элементов значение NULL	*true*	*false*	*false*	*true*
Степень > 1: у некоторых (не у всех) элементов значение NULL	*false*	*false*	*true*	*true*
Степень > 1: ни у одного элемента нет значения NULL	*false*	*true*	*true*	*false*

Примеры запросов с использованием предиката `is null`

Пример 14.7. На самом деле, в нашей формулировке запроса из примера 14.6 есть одна неточность. Если у некоторого служащего номер отдела неизвестен (значение столбца `EMP.DEPT_NO` у соответствующей строки таблицы служащих является неопределенным), то бессмысленно вычислять средний размер зарплаты отдела этого служащего и находить размер зарплаты руководителя отдела. Формулировка из примера 14.6 приведет к правильному результату, но это неочевидно.* Чтобы сделать формулировку более понятной (и, возможно, помочь системе выполнить запрос более эффективно), нужно воспользоваться предикатом `IS NOT NULL` и переписать запрос следующим образом:

```
SELECT EMP_NO, EMP_NAME, EMP_SAL
FROM EMP
WHERE DEPT_NO IS NOT NULL AND
   EMP_SAL BETWEEN
   (SELECT AVG(EMP1.EMP_SAL)
      FROM EMP EMP1
      WHERE EMP.DEPT_NO = EMP1.DEPT_NO)
         AND
   (SELECT EMP1.EMP_SAL
      FROM EMP EMP1
      WHERE EMP1.EMP_NO =
         ( SELECT DEPT.DEPT_MNG
         FROM DEPT
         WHERE DEPT.DEPT_NO = EMP.DEPT_NO ) );
```

Пример 14.8. Найти номера и имена служащих, номер отдела которых неизвестен.

```
SELECT EMP_NO, EMP_NAME
FROM EMP
WHERE DEPT_NO IS NULL;
```

* Покажем это в развернутой форме. Пусть *s* — текущая строка таблицы `EMP`, просматриваемой в цикле внешнего запроса, и пусть *s*.`DEPT_NO` содержит неопределенное значение. Тогда для строки *s* условие первого подзапроса будет иметь вид `NULL = EMP1.DEPT_NO`, и значением этого условия будет *uknown* для любой строки таблицы `EMP` (`EMP1`), просматриваемой в цикле этого подзапроса. Поскольку *uknown* не является разрешающим условием, результирующая таблица подзапроса будет пуста, и агрегатная функция `AVG` выдаст значение `NULL`. По этому поводу значением условия внешнего запроса будет *uknown*, и строка *s* не войдет в его результирующую таблицу.

Предикат `in`

Предикат позволяет специфицировать условие вхождения строчного значения в указанное множество значений. Синтаксические правила следующие:

```
in_predicate ::= row_value_constructor [ NOT ]
                 IN in_predicate_value
in_predicate_value ::= table_subquery
             | (value_expression_comma_list)
```

Строка, являющаяся первым операндом, и таблица-второй операнд должны быть одинаковой степени. В частности, если второй операнд представляет собой список значений, то первый операнд должен иметь степень 1. Типы данных соответствующих столбцов операндов должны быть совместимы.

Пусть `X` обозначает строку-первый операнд, а S — множество строк второго операнда. Обозначим через s строку-элемент этого множества. Тогда по определению условие `X IN S` эквивалентно булевскому выражению $\mathrm{OR}_{s \in S}$ `(X = s)`. Другими словами, `X IN S` принимает значение *true* в том и только в том случае, когда во множестве S существует хотя бы один элемент s, такой, что значением предиката `X = s` является *true*. `X IN S` принимает значение *false* в том и только том случае, когда для всех элементов s множества S значением операции сравнения `X = s` является *false*. Иначе значением условия `X IN S` является *unknown*. Заметим, что для пустого множества S значением `X IN S` является *false*.

По определению условие `X NOT IN S` эквивалентно `NOT (X IN S)`.

Примеры запросов с использованием предиката `in`

Пример 14.9. Найти номера, имена и номера отделов служащих, работающих в отделах 15, 17 и 19.

```
SELECT EMP_NO, EMP_NAME, DEPT_NO
FROM EMP
WHERE DEPT_NO IN (15, 17, 19);
```

Конечно, эта формулировка запроса эквивалентна следующей формулировке (**пример 14.9а**):

```
SELECT EMP_NO, EMP_NAME, DEPT_NO
FROM EMP
WHERE DEPT_NO = 15
  OR DEPT_NO = 17
  OR DEPT_NO = 19;
```

Пример 14.10. Найти номера служащих, не являющихся руководителями отделов и получающих заплату, размер которой равен размеру зарплаты какого-либо руководителя отдела.

```
SELECT EMP_NO
FROM EMP
WHERE EMP_NO NOT IN (SELECT DEPT_MNG FROM DEPT)
   AND EMP_SAL IN (SELECT EMP_SAL FROM EMP,
      DEPT WHERE EMP_NO = DEPT_MNG);
```

Запросы, содержащие предикат IN с подзапросом, легко переформулировать в запросы с соединениями. Например, запрос из примера 14.10 эквивалентен следующему запросу с соединениями (**пример 14.10а**):

```
SELECT DISTINCT EMP_NO
FROM EMP, EMP EMP1, DEPT
WHERE EMP_NO NOT IN (SELECT DEPT_MNG FROM DEPT)
 AND EMP_SAL = EMP1_SAL
 AND EMP1.EMP_NO = DEPT.DEPT_MNG;
```

По поводу этой второй формулировки следует сделать два замечания. Во-первых, как видно, мы изменили только ту часть условия, в которой использовался предикат IN, и не затронули предикат NOT IN. Запросы с предикатами NOT IN запросами с соединениями так просто не заменяются. Во-вторых, в разделе SELECT было добавлено ключевое слово DISTINCT, потому что в результате запроса во второй формулировке для каждого сотрудника будет содержаться столько строк, сколько существует руководителей отделов, получающих такую же зарплату, что и данный служащий.

Предикат like

Формально предикат like определяется следующими синтаксическими правилами:

```
like_predicate ::= source_value [ NOT ] LIKE pattern_value
        [ ESCAPE escape_value ]
source_value ::= value_expression
pattern_value ::= value_expression
escape_value ::= value_expression
```

Все три операнда (source_value, pattern_value и escape_value) должны быть одного типа: либо типа символьных строк, либо типа бито-

вых строк*. В первом случае значением последнего операнда должна быть строка из одного символа, во втором – строка из 8 бит. Второй операнд, как правило, задается литералом соответствующего типа. В обоих случаях значение предиката равняется *true* в том и только в том случае, когда исходная строка (source_value) может быть сопоставлена с заданным шаблоном (pattern_value).

Если обрабатываются символьные строки, и если раздел ESCAPE условия отсутствует, то при сопоставлении шаблона со строкой производится специальная интерпретация двух символов шаблона: символ подчеркивания ('_') обозначает любой одиночный символ; символ процента ('%') обозначает последовательность произвольных символов произвольной длины (длина последовательности может быть нулевой). Если же раздел ESCAPE присутствует и специфицирует некоторый одиночный символ x, то пары символов «x_» и «x%» представляют одиночные символы «_» и «%» соответственно.

В случае обработки битовых строк сопоставление шаблона со строкой производится восьмерками соседних бит (*октетами*). В соответствии со стандартом SQL:1999, при сопоставлении шаблона со строкой производится специальная интерпретация октетов со значениями X'25' и X'5F' (коды символов подчеркивания и процента в кодировке ASCII). Первый октет обозначает любой одиночный октет, а второй – последовательность произвольной длины произвольных октетов (длина может быть нулевой). В разделе ESCAPE указывается октет, отменяющий специальную интерпретацию октетов X'25' и X'5F'.

Значение предиката like есть *unknown*, если значение первого или второго операндов является неопределенным. Условие *x* NOT LIKE *y* ESCAPE *z* эквивалентно условию NOT *x* LIKE *y* ESCAPE *z*.

Примеры запросов с использованием предиката like

Пример 14.11. Найти номера проектов, в названии которых присутствуют слова 'next' и 'step'. Слова должны следовать именно в такой последовательности, но слово 'next' может быть первым в названии проекта.

```
SELECT PRO_TITLE
FROM PRO
WHERE PRO_TITLE LIKE '%next%step%'
   OR PRO_TITLE LIKE 'Next%step%';
```

Это очень неудачный запрос, потому что его выполнение, скорее всего, вынудит СУБД просмотреть все строки таблицы PRO и для каждой

* В стандарте SQL:1999 разрешается применять предикат LIKE только для битовых строк типа BLOB. Битовые строки типов BIT и BIT VARYING не допускаются.

строки выполнить две проверки столбца PRO_TITLE. Можно немного улучшить формулировку с небольшим риском получить неверный ответ (**пример 14.11а**):

```
SELECT PRO_TITLE
FROM PRO
WHERE PRO_TITLE LIKE '%ext%step%';
```

Пример 14.12. Найти номера отделов, служащие которых являются менеджерами проектов, и название каждого из этих проектов начинается с названия отдела.

```
SELECT DISTINCT DEPT.DEPT_NO
FROM EMP, DEPT, PRO
WHERE EMP.EMP_NO = PRO.PRO_MNG
 AND EMP.DEPT_NO = DEPT.DEPT_NO
 AND PRO.PRO_TITLE LIKE DEPT.DEPT_NAME || '%';
```

Вот как может выглядеть формулировка этого запроса, если использовать вложенные подзапросы (**пример 14.12а**):

```
SELECT DEPT.DEPT_NO
FROM DEPT
WHERE DEPT.DEPT_NO IN
   (SELECT EMP.DEPT_NO
   FROM EMP
   WHERE EMP.EMP_NO IN
      (SELECT PRO.PRO_MNG FROM PRO
   WHERE PRO.PRO_TITLE LIKE DEPT.DEPT_NAME || '%'));
```

Пример 14.13. Найти номера отделов, названия которых не начинаются со слова 'Software'.

```
SELECT DEPT_NO
FROM DEPT
WHERE DEPT_NAME NOT LIKE 'Software%';
```

Предикат similar

Формально предикат similar определяется следующими синтаксическими правилами:

```
similar_predicate ::= source_value [ NOT ]
     SIMILAR TO pattern_value [ ESCAPE escape_value ]
```

```
source_value ::= character_expression
pattern_value ::= character_expression
escape_value ::= character_expression
```

Все три операнда (source_value, pattern_value и escape_value) должны иметь тип символьных строк. Значением последнего операнда должна быть строка из одного символа. Второй операнд, как правило, задается литералом соответствующего типа. В обоих случаях значение предиката равняется *true* в том и только в том случае, когда шаблон (pattern_value) должным образом сопоставляется с исходной строкой (source_value).

Основное отличие предиката similar от рассмотренного ранее предиката like состоит в существенно расширенных возможностях задания шаблона, основанных на использовании правил построения *регулярных выражений*. Регулярные выражения предиката similar определяются следующими синтаксическими правилами:

```
regular_expression ::= regular_term
          | regular_expression vertical_bar regular_term
regular_term ::= regular_factor | regular_term regular_factor
regular_factor ::= regular_primary
          | regular_primary *
          | regular_primary +
regular_primary ::= character_specifier
          | %
          | regular_character_set
          | ( regular_expression )
character_specifier ::= non_escape_character | escape_character
regular_character_set ::= _
   | left_bracket character_enumeration_list right_bracket
   | left_bracket ^ character_enumeration_list right_bracket
   | left_bracket : regular_charset_id : right_bracket
character_enumeration ::= character_specifier
          | character_specifier - character_specifier
regular_charset_id ::= ALPHA | UPPER | LOWER | DIGIT | ALNUM
```

Поскольку в синтаксических правилах регулярных выражений символы «|», «[» и «]», используемые нами в качестве метасимволов в BNF, являются терминальными символами, они изображены как vertical_bar, left_bracket и right_bracket соответственно.

Создаваемое по приведенным правилам регулярное выражение представляет собой символьную строку, содержащую все символы, кото-

рые требуется явно сопоставлять с символами строки-источника. В строке могут находиться специальные символы, представляющие собой заменители обычных символов («%» и «_»), обозначения операций («|»), показатели числа возможных повторений («*» и «+») и т. д. При вычислении регулярного выражения образуются все возможные символьные строки, не содержащие специальных символов и соответствующие исходному шаблону. Тем самым, значением предиката `similar` является *true* в том и только в том случае, когда среди всех символьных строк, генерируемых по регулярному выражению `pattern_value`, найдется символьная строка, совпадающая с `source_value`.

Рассмотрим несколько примеров регулярных выражений.

Выражение `'(This is string1)|(This is string2)'` производит две символьные строки: `'(This is string1)'` и `'(This is string2)'`. В общем случае в круглых скобках могут находиться произвольные регулярные выражения `rexp1` и `rexp2`. Результатом вычисления `'(rexp1)|(rexp2)'` является множество символьных строк, генерируемых выражением `rexp1`, объединенное с множеством символьных строк, генерируемых выражением `rexp2`.

Выражение `'This is string [12]*'` генерирует символьные строки `'This is string '`, `'This is string 1'`, `'This is string 2'`, `'This is string 11'`, `'This is string 22'`, `'This is string 12'`, `'This is string 22'`, `'This is string 111'` и т. д. Конструкция в квадратных скобках представляет собой один из вариантов определения набора символов (`regular_character_set`). В данном случае символы, входящие в определяемый набор, просто перечисляются. При вычислении регулярного выражения в каждой из генерируемых символьных строк конструкция в квадратных скобках заменяется одним из символов соответствующего набора.

Специальный символ «*», стоящий после закрывающей квадратной скобки, является показателем числа повторений. «Звездочка» означает, что в генерируемых символьных строках элемент регулярного выражения, непосредственно предшествующий «звездочке», может появляться ноль или более раз. Использование в такой же ситуации специального символа «+» означает, что в генерируемых символьных строках элемент регулярного выражения, непосредственно предшествующий символу «плюс», может появляться один или более раз.

Другая форма определения набора символов иллюстрируется регулярным выражением `'This is string [:DIGIT:]'`. В этом случае конструкция в квадратных скобках представляет любой одиночный символ, изображающий десятичную цифру. Другими допустимыми в SQL идентификаторами наборов символов (`regular_charset_id`) являются `ALPHA` (любой символ алфавита), `UPPER` (любой символ верхнего регистра), `LOWER` (любой символ нижнего регистра) и `ALNUM` (любой алфавитно-цифровой символ).

Определяемый набор символов может задаваться нижней и верхней границей диапазона значений кодов допустимых символов. Например, в регулярном выражении 'This is string [3-8]' конструкция в квадратных скобках представляет собой любой одиночный символ, изображающий цифры от 3 до 8 включительно. Заметим, что при задании диапазона можно использовать любые символы, но требуется, чтобы значение кода символа левой границы диапазона было не больше значения кода символа правой границы.

Наконец, имеется еще одна возможность определения набора символов. Соответствующая конструкция позволяет указать, какие символы из общего набора символов SQL не входят в определяемый набор символов. Например, регулярное выражение '_S[^t]*ing%' генерирует все символьные строки, у которых вторым символом является «S», за которым (не обязательно непосредственно) следует подстрока «ing», но между «S» и «ing» отсутствуют вхождения символа «t».

Как и в предикате like, символ, определенный в разделе ESCAPE, поставленный перед любым специальным символом, отменяет специальную интерпретацию этого символа.

В заключение данного пункта вернемся к отложенному в разделе «Скалярные выражения» лекции 13 обсуждению функции SUBSTRING ... SIMILAR ... ESCAPE. Напомним, что вызов этой функции определяется следующим синтаксисом:

```
SUBSTRING (character_value_expression
   SIMILAR character_value_expression
      ESCAPE character_value_expression)
```

Предположим, что в разделе ESCAPE (который должен присутствовать обязательно) задан символ «x». Тогда символьная строка, задаваемая во втором операнде, должна иметь вид 'rexp1x"rexp2x"rexp3', где rexp1, rexp2 и rexp3 являются регулярными выражениями. Функция пытается разделить символьную строку первого операнда на три раздела, первый из которых определяется путем сопоставления начала строки со строками, генерируемыми rexp1, второй – путем сопоставления оставшейся части строки первого операнда с rexp2 и третий – путем сопоставления конца этой строки с rexp3. Возвращаемым значением функции является средняя часть символьной строки первого операнда.

Вот пример вызова функции:

```
SUBSTRING ( 'This is string22'
   SIMILAR 'This is\"[:ALPHA:]+\"[:DIGIT:]+'
      ESCAPE '\' )
```

Результатом будет строка 'string'.

Примеры запросов с использованием предиката similar

Пример 14.14. Найти номера и названия отделов, название которых начинается со слов 'Hardware' или 'Software', а за ними (не обязательно непосредственно) следует последовательность десятичных цифр, предваряемых символом подчеркивания.

```
SELECT DEPT_NAME, DEPT_NO
FROM DEPT
WHERE DEPT_NAME SIMILAR TO '(HARD|SOFT)WARE%\_[:DIGIT:]+' ESCAPE '\';
```

Пример 14.15. Найти номера и названия проектов, название которых не начинается с последовательности цифр.

```
SELECT DEPT_NAME, DEPT_NO
FROM DEPT
WHERE DEPT_NAME SIMILAR TO '[^1-9]+%';
```

Предикат exists

Предикат exists определяется следующим синтаксическим правилом:

```
exists_predicate ::= EXISTS (query_expression)
```

Значением условия EXISTS (query_expression) является *true* в том и только в том случае, когда мощность таблицы-результата выражения запросов больше нуля, иначе значением условия является *false*.

Примеры запросов с использованием предиката exists

Пример 14.16. Найти номера отделов, среди служащих которых имеются менеджеры проектов.

```
SELECT DEPT.DEPT_NO
FROM DEPT
WHERE EXISTS
   (SELECT EMP.EMP_NO
   FROM EMP
   WHERE EMP.DEPT_NO = DEPT.DEPT_NO
      AND EXISTS
         (SELECT PRO.PRO_MNG
         FROM PRO
         WHERE PRO.PRO_MNG = EMP.EMP_NO));
```

Эту формулировку можно упростить, избавившись от самого вложенного запроса (**пример 14.16a**):

```
SELECT DEPT.DEPT_NO
FROM DEPT
WHERE EXISTS
   (SELECT EMP.EMP_NO
      FROM EMP, PRO
      WHERE EMP.DEPT_NO = DEPT.DEPT_NO
         AND PRO.PRO_MNG = EMP.EMP_NO);
```

Далее заметим, что по смыслу предиката EXISTS список выборки во вложенном подзапросе является несущественным, и формулировку запроса можно изменить, например, следующим образом (**пример 14.16b**):

```
SELECT DEPT.DEPT_NO
FROM DEPT
WHERE EXISTS
   (SELECT *
      FROM EMP, DEPT
      WHERE EMP.DEPT_NO = DEPT.DEPT_NO
         AND PRO.PRO_MNG = EMP.EMP_NO);
```

Запросы с предикатом EXISTS можно также переформулировать в виде запросов с предикатом сравнения (**пример 14.16c**):

```
SELECT DEPT.DEPT_NO
FROM DEPT
WHERE (SELECT COUNT(*)
   FROM EMP, DEPT
   WHERE EMP.DEPT_NO = DEPT.DEPT_NO
      AND PRO.PRO_MNG = EMP.EMP_NO ) >= 1;
```

Пример 14.17. Найти номера отделов, размер заработной платы сотрудников которых не превышает размер заработной платы руководителя отдела.

```
SELECT DEPT.DEPT_NO
FROM DEPT
WHERE NOT EXISTS
   (SELECT *
      FROM EMP EMP1, EMP EMP2
```

```
WHERE EMP1.EMP_NO = DEPT.DEPT_MNG AND
   EMP2.DEPT_NO = DEPT.DEPT_NO AND
   EMP2.EMP_SAL > EMP1.EMP_SAL);
```

Предикат `unique`

Этот предикат позволяет сформулировать условие отсутствия дубликатов в результате запроса:

```
unique_predicate ::= UNIQUE (query_expression)
```

Результатом вычисления условия `UNIQUE (query_expression)` является *true* в том и только в том случае, когда в таблице-результате выражения запросов отсутствуют какие-либо две строки, одна из которых является дубликатом другой. В противном случае значение условия есть *false*.

Примеры запросов с использованием предиката `unique`

Пример 14.18. Найти номера отделов, служащих которых можно различить по имени и дате рождения.

```
SELECT DEPT_NO
FROM DEPT
WHERE UNIQUE
   (SELECT EMP_NAME, EMP_BDATE
      FROM EMP
      WHERE EMP.DEPT_NO = DEPT.DEPT_NO);
```

Возможна альтернативная, но более сложная формулировка этого запроса с использованием предиката `NOT EXISTS` **(пример 14.18а)**:

```
SELECT DEPT_NO
FROM DEPT
WHERE NOT ESISTS
   (SELECT *
      FROM EMP, EMP EMP1
      WHERE EMP1.EMP_NO <> EMP.EMP_NO
         AND EMP.DEPT_NO = DEPT.DEPT_NO
         AND EMP1.DEPT_NO = DEPT.DEPT_NO
         AND EMP1.EMP_NAME = EMP.EMP_NAME
         AND(EMP1.EMP_BDATE = EMP.EMP_BDATE
            OR (EMP.EMP_BDATE IS NULL
            AND EMP1.EMP_BDATE IS NULL)));
```

Если же ограничиться требованием уникальности имен служащих, то возможна следующая формулировка (**пример 14.18b**):

```
SELECT DEPT_NO
FROM DEPT
WHERE (SELECT COUNT (EMP_NAME)
      FROM EMP
      WHERE EMP.DEPT_NO = DEPT.DEPT_NO) =
   (SELECT COUNT (DISTINCT EMP_NAME)
      FROM EMP
      WHERE EMP.DEPT_NO = DEPT.DEPT_NO);
```

Предикат `overlaps`

Этот предикат служит для проверки перекрытия во времени двух событий. Условие определяется следующим синтаксисом:

```
overlaps_predicate ::= row_value_constructor OVERLAPS
        row_value_constructor
```

Степень каждой из строк-операндов должна быть равна 2. Тип данных первого столбца каждого из операндов должен быть типом даты-времени, и типы данных первых столбцов должны быть совместимы. Тип данных второго столбца каждого из операндов должен быть типом даты-времени или интервала. При этом:

- если это тип интервала, то точность типа должна быть такой, чтобы интервал можно было прибавить к значению типа дата-время первого столбца;
- если это тип дата-время, то он должен быть совместим с типом данных дата-время первого столбца.

Пусть `D1` и `D2` — значения первого столбца первого и второго операндов соответственно. Если второй столбец первого операнда имеет тип дата-время, то пусть `E1` обозначает его значение. Если второй столбец первого операнда имеет тип `INTERVAL`, то пусть `I1` —его значение, а `E1 = D1 + I1`. Если `D1` является неопределенным значением или если `E1 < D1`, то пусть `S1 = E1` и `T1 = D1`. В противном случае, пусть `S1 = D1` и `T1 = E1`. Аналогично определяются `S2` и `T2` применительно ко второму операнду. Результат условия совпадает с результатом вычисления следующего булевского выражения:

```
(S1 > S2 AND NOT (S1 >= T2 AND T1 >= T2))
OR
(S2 > S1 AND NOT (S2 >= T1 AND T2 >= T1))
```

```
OR
(S1 = S2 AND (T1 <> T2 OR T1 = T2))
```

Примеры запросов с использованием предиката `overlaps`

Пример 14.19. Найти номера проектов, которые выполнялись в период с 15 января 2000 г. по 31 декабря 2002 г.

```
SELECT PRO_NO
FROM PRO
WHERE (PRO_SDATE, PRO_DURAT) OVERLAPS
   (DATE '2000-01-15', DATE '2002-12-31');
```

Пример 14.20. Найти названия проектов, которые будут выполняться в течение следующего года.

```
SELECT PRO_TITLE
FROM PRO
WHERE (PRO_SDATE, PRO_DURAT) OVERLAPS
    (CURRENT_DATE, INTERVAL '1' YEAR);
```

Предикат сравнения с квантором

Этот предикат позволяет специфицировать квантифицированное сравнение строчного значения и определяется следующим синтаксическим правилом:

```
quantified_comparison_predicate ::= row_value_constructor
        comp_op { ALL | SOME | ANY } query_expression
```

Степень первого операнда должна быть такой же, как и степень таблицы-результата выражения запросов. Типы данных значений строки-операнда должны быть совместимы с типами данных соответствующих столбцов выражения запроса. Сравнение строк производится по тем же правилам, что и для предиката сравнения.

Обозначим через x строку-первый операнд, а через S – результат вычисления выражения запроса. Пусть s обозначает произвольную строку таблицы S. Тогда:

- условие x `comp_op ALL` S имеет значение *true* в том и только в том случае, когда S пусто, или значение условия x `comp_op` s равно *true* для каждой строки s, входящей в S. Условие x `comp_op ALL` S имеет значение *false* в том и только в том случае, когда значение предиката x `comp_op` s равно *false* хотя бы для одной строки s, входящей в S. В остальных случаях значение условия x `comp_op ALL` S равно *unknown*;

- условие *x* comp_op SOME *S* имеет значение *false* в том и только в том случае, когда *S* пусто, или значение условия *x* comp_op *s* равно *false* для каждой строки *s*, входящей в *S*. Условие *x* comp_op SOME *S* имеет значение *true* в том и только в том случае, когда значение предиката *x* comp_op *s* равно *true* хотя бы для одной строки *s*, входящей в *S*. В остальных случаях значение условия *x* comp_op SOME *S* равно *unknown*;
- условие *x* comp_op ANY *S* эквивалентно условию *x* comp_op SOME *S*.

Примеры запросов с использованием предиката сравнения с квантором

Пример 14.21. Найти номера служащих отдела номер 65, зарплата которых в этом отделе не является минимальной.

```
SELECT EMP_NO
FROM EMP
WHERE DEPT_NO = 65
   AND EMP_SAL > SOME (SELECT EMP1.EMP_SAL
                       FROM EMP EMP1
                       WHERE EMP.DEPT_NO = EMP1.DEPT_NO);
```

Одна из возможных альтернативных формулировок этого запроса может основываться на использовании предиката EXISTS **(пример 14.21a)**:

```
SELECT EMP_NO
FROM EMP
WHERE DEPT_NO = 65
   AND EXISTS(SELECT *
              FROM EMP EMP1
              WHERE EMP.DEPT_NO = EMP1.DEPT_NO
                 AND EMP.EMP_SAL > EMP1.EMP_SAL);
```

Вот альтернативная формулировка этого запроса, основанная на использовании агрегатной функции MIN **(пример 14.21b)**:

```
SELECT EMP_NO
FROM EMP
WHERE DEPT_NO = 65 AND
   EMP_SAL > (SELECT MIN(EMP1.EMP_SAL)
              FROM EMP EMP1
              WHERE EMP.DEPT_NO = EMP1.DEPT_NO);
```

Пример 14.22. Найти номера и имена служащих отдела 65, однофамильцы которых работают в этом же отделе.

```
SELECT EMP_NO, EMP_NAME
FROM EMP
WHERE DEPT_NO = 65 AND
   EMP_NAME = SOME (SELECT EMP1.EMP_NAME
                    FROM EMP EMP1
                    WHERE EMP.DEPT_NO = EMP1.DEPT_NO
                       AND EMP.EMP_NO <> EMP1.EMP_NO);
```

Заметим, что эта формулировка эквивалентна следующей формулировке **(пример 14.22a)**:

```
SELECT EMP_NO, EMP_NAME
FROM EMP
WHERE DEPT_NO = 65 AND
   EMP_NAME IN (SELECT EMP1.EMP_NAME
                FROM EMP EMP1
                WHERE EMP.DEPT_NO = EMP1.DEPT_NO
                   AND EMP.EMP_NO <> EMP1.EMP_NO);
```

Возможна формулировка с использованием агрегатной функции COUNT **(пример 14.22b)**:

```
SELECT EMP_NO, EMP_NAME
FROM EMP
WHERE DEPT_NO = 65 AND
   (SELECT COUNT(*)
      FROM EMP EMP1
      WHERE EMP.DEPT_NO = EMP1.DEPT_NO
         AND EMP.EMP_NO <> EMP1.EMP_NO ) >= 1;
```

Наиболее лаконичным образом этот запрос можно сформулировать с использованием соединения **(пример 14.22c)**:

```
SELECT DISTINCT EMP.EMP_NO, EMP.EMP_NAME
FROM EMP, EMP EMP1
WHERE EMP.DEPT_NO = 65
   AND EMP.EMP_NAME = EMP1.EMP_NAME
   AND EMP.DEPT_NO = EMP1.DEPT_NO
   AND EMP.EMP_NO <> EMP1.EMP_NO;
```

В последней формулировке мы вынуждены везде использовать уточненные имена столбцов, потому что на одном уровне используются два вхождения таблицы EMP.

Пример 14.23. Найти номера служащих отдела номер 65, зарплата которых в этом отделе является максимальной.

```
SELECT EMP_NO
FROM EMP
WHERE DEPT_NO = 65
 AND EMP_SAL >= ALL(SELECT EMP1.EMP_SAL
                    FROM EMP EMP1
                    WHERE EMP.DEPT_NO = EMP1.DEPT_NO);
```

Одна из возможных альтернативных формулировок этого запроса может основываться на использовании предиката NOT EXISTS **(пример 14.23a)**:

```
SELECT EMP_NO
FROM EMP
WHERE DEPT_NO = 65
 AND NOT EXISTS (SELECT *
                 FROM EMP EMP1
                 WHERE EMP.DEPT_NO = EMP1.DEPT_NO
                   AND EMP.EMP_SAL < EMP1.EMP_SAL);
```

Можно сформулировать этот же запрос с использованием агрегатной функции MAX **(пример 14.23.b)**:

```
SELECT EMP_NO
FROM EMP
WHERE DEPT_NO = 65
 AND EMP_SAL = (SELECT MAX(EMP1.EMP_SAL)
                FROM EMP EMP1
                WHERE EMP.DEPT_NO = EMP1.DEPT_NO);
```

Пример 14.24. Найти номера и имена служащих, не имеющих однофамильцев.

```
SELECT EMP_NO, EMP_NAME
FROM EMP
WHERE EMP_NAME <> ALL (SELECT EMP1.EMP_NAME
                       FROM EMP EMP1
                       WHERE EMP1.EMP_NO <> EMP.EMP_NO);
```

Этот запрос можно переформулировать на основе использования

предиката NOT EXISTS или агрегатной функции COUNT (по причине очевидности мы не приводим эти формулировки), но, в отличие от случая в примере 14.22c, формулировка в виде запроса с соединением здесь не проходит. Формулировка запроса

```
SELECT DISTINCT EMP_NO, EMP_NAME
FROM EMP, EMP EMP1
WHERE EMP.EMP_NAME <> EMP1.EMP_NAME
   AND EMP1.EMP_NO <> EMP.EMP_NO);
```

эквивалентна формулировке

```
SELECT EMP_NO, EMP_NAME
FROM EMP
WHERE EMP_NAME <> SOME (SELECT EMP1.EMP_NAME
                        FROM EMP EMP1
                        WHERE EMP1.EMP_NO <> EMP.EMP_NO);
```

Очевидно, что этот запрос является бессмысленным («Найти сотрудников, для которых имеется хотя бы один не однофамилец»).

Предикат match

Предикат позволяет сформулировать условие соответствия строчного значения результату табличного подзапроса. Синтаксис определяется следующим правилом:

```
match_predicate ::= row_value_constructor
          MATCH [ UNIQUE ] [ SIMPLE | PARTIAL | FULL ]
             query_expression
```

Степень первого операнда должна совпадать со степенью таблицы-результата выражения запроса. Типы данных столбцов первого операнда должны быть совместимы с типами соответствующих столбцов табличного подзапроса. Сравнение пар соответствующих значений производится аналогично тому, как это специфицировалось для предиката сравнения.

Пусть *x* обозначает строку-первый операнд. Тогда:

- Если отсутствует спецификация вида сопоставления или специфицирован тип сопоставления SIMPLE, то:
  - если значение некоторого столбца *x* является неопределенным, то значением условия является *true*;

- если в x нет неопределенных значений, то:
  - если не указано UNIQUE, и в результате выражения запроса существует (возможно, не уникальная) строка s в такая, что $x = s$, то значением условия является *true*;
  - если указано UNIQUE, и в результате выражения запроса существует уникальная строка s, такая, что $x = s$, то значением условия является *true*;
  - в противном случае значением условия является *false*.

- Если в условии присутствует спецификация PARTIAL, то:
  - если все значения в x являются неопределенными, то значение условия есть *true*;
  - иначе:
    - если не указано UNIQUE, и в результате выражения запроса существует (возможно, не уникальная) строка s, такая, что каждое отличное от неопределенного значение x равно соответствующему значению s, то значение условия есть *true*;
    - если указано UNIQUE, и в результате выражения запроса существует уникальная строка s, такая, что каждое отличное от неопределенного значение x равно соответствующему значению s, то значение условия есть *true*;
    - в противном случае значение условия есть *false*.
- Если в условии присутствует спецификация FULL, то:
  - если все значения в x неопределенные, то значение условия есть *true*;
  - если ни одно значение в x не является неопределенным, то:
    - если не указано UNIQUE, и в результате выражения запроса существует (возможно, не уникальная) строка s, такая, что $x = s$, то значение условия есть *true*;
    - если указано UNIQUE, и в результате выражения запроса существует уникальная строка s, такая, что $x = s$, то значение условия есть *true*;
    - в противном случае значение условия есть *false*.
  - в противном случае значение условия есть *false*.

Примеры запросов с использованием предиката `match`

Все примеры этого пункта основаны на запросе «Найти номера служащих и номера их отделов для служащих, для которых в отделе со «схожим» номером работает служащий со «схожей» датой рождения» с некоторыми уточнениями.

Пример 14.25.

```
SELECT EMP_NO, DEPT_NO
FROM EMP
WHERE (DEPT_NO, EMP_BDATE) MATCH SIMPLE
   (SELECT EMP1.DEPT_NO, EMP1.EMP_BDATE
      FROM EMP EMP1
   WHERE EMP1.EMP_NO <> EMP.EMP_NO);
```

Этот запрос вернет данные о служащих, про которых:

- либо неизвестны номер отдела *или* дата рождения (или и то, и другое);
- либо в отделе данного служащего работает по крайней мере еще один человек с той же датой рождения.

Если использовать предикат `MATCH UNIQUE SIMPLE`, то мы получим данные о служащих, про которых:

- либо неизвестны номер отдела *или* дата рождения (или и то, и другое);
- либо в отделе данного служащего работает еще один человек с той же датой рождения.

Пример 14.26.

```
SELECT EMP_NO, DEPT_NO
FROM EMP
WHERE (DEPT_NO, EMP_BDATE) MATCH PARTIAL
   (SELECT EMP1.DEPT_NO, EMP1.EMP_BDATE
      FROM EMP EMP1
      WHERE EMP1.EMP_NO <> EMP.EMP_NO);
```

Этот запрос вернет данные о служащих, про которых:

- либо неизвестны номер отдела *и* дата рождения;
- либо неизвестен номер отдела, но имеется по крайней мере еще один человек с той же датой рождения;
- либо неизвестна дата рождения, но в отделе данного служащего работает по крайней мере еще один человек;
- либо известны и номер отдела, и дата рождения, и в отделе данного служащего работает по крайней мере еще один человек с той же датой рождения.

Если использовать предикат `MATCH UNIQUE PARTIAL`, то мы получим данные о служащих, про которых:

- либо неизвестны номер отдела *и* дата рождения;
- либо неизвестен номер отдела, но имеется еще один человек с той же датой рождения;

- либо неизвестна дата рождения, но в отделе данного служащего работает еще один человек;
- либо известны и номер отдела, и дата рождения, и в отделе данного служащего работает еще один человек с той же датой рождения.

Пример 14.27.

```
SELECT EMP_NO, DEPT_NO
FROM EMP
WHERE (DEPT_NO, EMP_BDATE) MATCH FULL
   (SELECT EMP1.DEPT_NO, EMP1.EMP_BDATE
     FROM EMP EMP1
     WHERE EMP1.EMP_NO <> EMP.EMP_NO);
```

Этот запрос вернет данные о служащих, о которых:

- либо неизвестны номер отдела *и* дата рождения;
- либо в отделе данного служащего работает по крайней мере еще один человек с той же датой рождения.

Если использовать предикат `MATCH UNIQUE FULL`, то мы получим данные о служащих, о которых:

- либо неизвестны номер отдела *и* дата рождения;
- либо в отделе данного служащего работает еще один человек с той же датой рождения.

Предикат `distinct`

Предикат позволяет проверить, являются ли две строки дубликатами. Условие определяется следующим синтаксическим правилом:

```
distinct_predicate ::= row_value_constructor IS DISTINCT FROM
     row_value_constructor
```

Строки-операнды должны быть одинаковой степени. Типы данных соответствующих значений строк-операндов должны быть совместимы.

Напомним, что две строки *s1* с именами столбцов *c1*, *c2*, ..., *cn* и *s2* с именами столбцов *d1*, *d2*, ..., *dn* считаются строками-дубликатами, если для каждого *i* (*i* = *1*, *2*, ..., *n*) либо *ci* и *di* не содержат NULL, и (*ci* = *di*) = ***true***, либо и *ci*, и *di* содержат NULL. Значением условия *s1* IS DISTINCT FROM *s2* является ***true*** в том и только в том случае, когда строки *s1* и *s2* не являются дубликатами. В противном случае значением условия является ***false***.

Заметим, что отрицательная форма условия — `IS NOT DISTINCT FROM` — в стандарте SQL не поддерживается. Вместо этого можно воспользоваться выражением `NOT` *s1* `IS DISTINCT FROM` *s2*.

Примеры запросов с использованием предиката `distinct`

Пример 14.28. Найти номера и имена служащих отдела 65, которых можно отличить по данным об имени и дате рождения от руководителя отдела 65.

```
SELECT EMP_NO, EMP_NAME
FROM EMP
WHERE DEPT_NO = 65
   AND (EMP_NAME, EMP_BDATE) IS DISTINCT FROM
      (SELECT EMP1.EMP_NAME, EMP1.EMP_BDATE
         FROM EMP EMP1, DEPT
         WHERE EMP1.DEPT_NO = EMP.DEPT_NO
            AND DEPT.DEPT_MNG = EMP1.EMP_NO);
```

Пример 14.29. Найти все пары номеров таких служащих отдела 65, которых нельзя различить по данным об имени и дате рождения.

```
SELECT EMP1.EMP_NO, EMP2.EMP_NO
FROM EMP EMP1, EMP EMP2
WHERE EMP1.EMP_NO <> EMP2.EMP_NO
   AND NOT ((EMP1.EMP_NAME, EMP1.EMP_BDATE) IS DISTINCT FROM
         (EMP2.EMP_NAME, EMP2.EMP_BDATE));
```

Заключение

В этой лекции мы обсудили наиболее важные возможности языка SQL, связанные с выборкой данных. Даже простые примеры, приводившиеся в лекции, показывают исключительную избыточность языка SQL. Еще в то время, когда действующим стандартом языка был SQL/92, была опубликована любопытная статья, в которой приводилось 25 формулировок одного и того же несложного запроса. При использовании всех возможностей SQL:1999 этих формулировок было бы гораздо больше.

Можно спорить, хорошо или плохо иметь возможность формулировать один и тот же запрос десятками разных способов. На мой взгляд, это не очень хорошо, поскольку увеличивает вероятность появления ошибок в запросах (особенно в сложных запросах). С другой стороны, таково объективное состояние дел, и мы стремились обеспечить в этой лекции материал, достаточный для того, чтобы прочувствовать различные возможности формулировки запросов. Как показывают следующие две лекции, возможности, предоставляемые оператором `SELECT`, в действительности гораздо шире.

Лекция 15. Язык баз данных SQL: группировка и условия раздела HAVING, порождаемые и соединенные таблицы

В этой лекции мы завершаем обсуждение основных (традиционных) конструкций оператора SELECT языка SQL. В разделе «Агрегатные функции, группировка и условия раздела HAVING» обсуждаются разделы GROUP BY и HAVING. Основной акцент делается на способах конструирования условий раздела HAVING. На примерах демонстрируется, что разделы GROUP BY и HAVING действительно полезны, а иногда и необходимы при формулировке запросов с вызовами агрегатных функций. В разделах «Ссылки на порождаемые таблицы в разделе FROM» и «Более сложные конструкции оператора выборки» мы возвращаемся к разновидностям ссылок на таблицу в разделе FROM и последовательно обсуждаем порождаемые таблицы, соединенные таблицы и порождаемые таблицы с горизонтальной связью.

Ключевые слова: агрегатные функции, семантика агрегатных функций, типы значений агрегатных функций, использование агрегатных функций в разделах HAVING и SELECT оператора выборки, логические выражения раздела HAVING, предикаты сравнения, предикат between, предикат null, предикат in, предикат like, предикат exists, предикат unique, предикаты сравнения с квантором, предикат distinct, ссылки на порождаемые таблицы в разделе FROM, соединенная таблица, соответствующие столбцы соединения, список выборки соответствующих столбцов соединения, прямое соединение (CROSS JOIN), внутреннее соединение по условию (INNER JOIN ... ON), внутреннее соединение по совпадению значений указанных одноименных столбцов (INNER JOIN ... USING), естественное внутреннее соединение (NATURAL INNER JOIN), левое внешнее соединение по условию (LEFT OUTER JOIN ... ON), правое внешнее соединение по условию (RIGHT OUTER JOIN ... ON), полное внешнее соединение по условию (FULL OUTER JOIN ... ON), симметричное внешнее соединение, левое внешнее соединение по совпадению значений указанных одноименных столбцов (LEFT OUTER JOIN ... USING), правое внешнее соединение по совпадению значений указанных одноименных столбцов (RIGHT OUTER JOIN ... USING), полное внешнее соединение по совпадению значений указанных одноименных столбцов (FULL OUTER JOIN ... USING), естественное левое внешнее соединение (NATURAL LEFT OUTER JOIN), естественное правое внешнее соединение (NATURAL RIGHT OUTER JOIN), естественное полное внешнее соединение (NATURAL FULL OUTER JOIN), соединение объединением

(UNION JOIN), порождаемые таблицы с горизонтальной связью (lateral derived table).

Введение

В предыдущих двух лекциях мы обсудили допускаемые в стандарте SQL виды ссылок на таблицы в разделе FROM оператора SELECT и подробно, с многочисленными примерами, рассмотрели возможные способы построения условных выражений раздела WHERE. Данную лекцию мы начинаем с анализа возможностей и целесообразности использования в запросах разделов GROUP BY и HAVING. Соответствующий раздел «Агрегатные функции, группировка и условия раздела HAVING» формально похож на раздел «Логические выражения раздела WHERE» лекции 14: обсуждаются виды предикатов, которые можно использовать в условных выражениях раздела HAVING, и приводятся иллюстрирующие примеры. Но в действительности мы преследуем бóльшую цель: показать, что во многих случаях разделы GROUP BY и HAVING являются избыточными; запрос можно сформулировать более понятным образом без их использования. Применение разделов GROUP BY и HAVING оказывается действительно полезным, а иногда и необходимым, в тех случаях, когда в запросе присутствует несколько вызовов агрегатных функций на группах строк.

После обсуждения разделов GROUP BY и HAVING можно будет считать, что мы полностью рассмотрели базовые конструкции оператора выборки (раздел ORDER BY не заслуживает дополнительного обсуждения). Поэтому в разделах «Ссылки на порождаемые таблицы в разделе FROM» и «Более сложные конструкции оператора выборки» мы возвращаемся к отложенным в лекции 13 темам порождаемых таблиц, соединенных таблиц и порождаемых таблиц с горизонтальной связью.

В обычных порождаемых таблицах SQL нет ничего особенного. По всей видимости, возможность указывать в разделе FROM выражения запросов, а не только ссылки на базовые или представляемые таблицы, была введена в SQL на основе следующих естественных соображений. Результатом вычисления выражения запросов в SQL является таблица. Следовательно, в любой конструкции языка, где может присутствовать ссылка на таблицу SQL, следует допустить присутствие выражения запросов. Одновременное наличие возможностей определения представляемых таблиц, указания именованного выражения запросов в разделе WITH и указания выражения запросов порождаемой таблицы непосредственно в списке раздела FROM, очевидно, является избыточным.

Соединенные таблицы появились еще в стандарте SQL/92, и внедрение в стандарт SQL этой возможности было действительно обоснованным. В соответствии с традиционной общей семантикой оператора

SELECT в нем вообще не предусматривали явные средства для выражения потребности в соединении двух или более таблиц. Наличие возможности указывать несколько ссылок на таблицы в разделе FROM и спецификации произвольного логического выражения в разделе WHERE для ограничения расширенного декартова произведения этих таблиц позволяет выражать с помощью традиционных средств SQL соединение общего вида в смысле Кодда, и до поры до времени это считалось достаточным.

Внешние соединения

Но имеются два важных частных случая соединений, которые выражаются с помощью традиционных средств SQL излишне громоздко,— это естественные и внешние соединения. При наличии возможности определения внешних ключей таблицы кажется достаточно странной потребность всякий раз явно указывать в запросах условие естественного соединения. Например, во многих примерах запросов в лекции 14 присутствует условие соединения EMP.DEPT_NO = DEPT.DEPT_NO в тех случаях, когда в действительности нам требовался результат операции EMP NATURAL JOIN DEPT.

Внешние соединения были введены еще Эдгаром Коддом в 1979 г. В целом, основная идея этой разновидности операции соединения состояла в том, что, с одной стороны, результат операции обычного соединения двух отношений повышает информационный уровень данных, поскольку в результате операции мы имеем информационно связанные данные. Но, с другой стороны, в результирующем отношении мы теряем информацию об исходных объектах, которые оказались несвязанными и не вошли в результат соединения. Кодд придумал, как, используя неопределенные значения, определить обобщенную операцию, которая будет обладать достоинствами обычной операции соединения, не приводя к потере исходной информации. Вернее, он предложил три операции: левое внешнее соединение, правое внешнее соединение и полное (симметричное) внешнее соединение. Приведем их определения (в реляционных терминах данного курса).

Пусть имеются отношения *r1* и *r2*, совместимые относительно операции взятия расширенного декартова произведения. Пусть *s* является результатом операции *r1* LEFT OUTER JOUN *r2* WHERE *comp* (левое внешнее соединение *r1* и *r2* по условию *comp*). Тогда Hs = Hr1 union Hr2. Пусть *tr1* ∈ *Br1* и *tr2* ∈ *Br2*. Тогда *tr1* union *tr2* ∈ *Bs* в том и только в том случае, когда *comp* (*tr1* union *tr2*) = *true*. Если имеется кортеж *tr1* ∈ *Br1*, для которого нет ни одного кортежа *tr2* ∈ *r2*, такого, что *comp* (*tr1* union *tr2*) = *true*, то *tr1* union $tr2_{null}$ ∈ *Bs*, где $tr2_{null}$ — кортеж, соответствующий *Hr2*, все значения которого являются неопределенными*.

* Здесь мы прибегаем к компромиссу между реляционной терминологией и моделью данных SQL: конечно, в реляционной модели кортеж из неопределенных значений не может соответствовать заголовку отношения, поскольку NULL не является значением ни одного типа данных.

Пусть s является результатом операции *r1* RIGHT OUTER JOUN *r2* WHERE *comp* (правое внешнее соединение *r1* и *r2* по условию *comp*). Тогда Hs = Hr1 union Hr2. Пусть *tr1* ∈ *Br1* и *tr2* ∈ *Br2*. Тогда *tr1* union *tr2* ∈ *Bs* в том и только в том случае, когда *comp* (*tr1* union *tr2*) = *true*. Если имеется кортеж *tr2* ∈ *Br2*, для которого нет ни одного такого кортежа *tr1* ∈ *Br1*, что *comp* (*tr1* union *tr2*) = *true*, то $tr1_{null}$ union *tr2* ∈ Bs, где $tr1_{null}$ – кортеж, соответствующий Hr1, все значения которого являются неопределенными.

Наконец, пусть *s* является результатом операции *r1* FULL OUTER JOUN *r2* WHERE *comp* (полное внешнее соединение *r1* и *r2* по условию *comp*). Тогда Hs = Hr1 union Hr2. Пусть *tr1* ∈ *Br1* и *tr2* ∈ *Br2*. Тогда *tr1* union *tr2* ∈ *Bs* в том и только в том случае, когда *comp* (*tr1* union *tr2*) = *true*. Если имеется кортеж *tr1* ∈ *Br1*, для которого нет ни одного кортежа *tr2* ∈ *Br2*, такого, что *comp* (*tr1* union *tr2*) = *true*, то *tr1* union $tr2_{null}$ ∈ *Bs*, где $tr2_{null}$ – кортеж, соответствующий Hr2, все значения которого являются неопределенными. Если имеется кортеж *tr2* ∈ *Br2*, для которого нет ни одного кортежа *tr1* ∈ *Br1*, такого, что *comp* (*tr1* union *tr2*) = *true*, то $tr1_{null}$ union *tr2* ∈ Bs, где $tr1_{null}$ – кортеж, соответствующий Hr1, все значения которого являются неопределенными.

Понятно, что традиционными средствами SQL можно выразить все виды внешних соединений (например, с использованием переключателей), но такие запросы будут очень громоздкими. Компании-производители SQL-ориентированных СУБД пытались обеспечивать выразительные средства внешних соединений путем расширения системы обозначений для операций сравнения. Этот подход был не слишком удачным и не обеспечивал общего решения.

В стандарте языка SQL специфицирован отдельный специализированный подъязык для формирования выражений соединения таблиц. Такие выражения называются соединенными таблицами, и их можно использовать в качестве ссылок на таблицы в списке раздела FROM. Разработчики стандарта SQL не любят мельчить — в языке допускается 14 видов соединений:

- прямое соединение;
- внутреннее соединение по условию;
- внутреннее соединение по совпадению значений указанных одноименных столбцов;
- естественное внутреннее соединение;
- левое внешнее соединение по условию;
- правое внешнее соединение по условию;
- полное внешнее соединение по условию;
- левое внешнее соединение по совпадению значений указанных одноименных столбцов;
- правое внешнее соединение по совпадению значений указанных одноименных столбцов;
- полное внешнее соединение по совпадению значений указанных одно-

именных столбцов;
- естественное левое внешнее соединение;
- естественное правое внешнее соединение;
- естественное полное внешнее соединение;
- соединение объединением.

Во всех этих операциях нет ничего сложного, но их неформальное описание исключительно громоздко. Поэтому в разделе «Более сложные конструкции оператора выборки» мы определяем операции на формальном уровне, а потом иллюстрируем их на примерах.

Наконец, последняя тема этой лекции относится к еще одному типу ссылок на таблицу, допускаемых в разделе FROM: *порождаемым таблицам с горизонтальной связью*. Фактически порождаемая таблица с горизонтальной связью представляет собой выражение запросов, в котором может присутствовать корреляция со строками таблиц, специфицированных в списке раздела FROM *слева* от данной порождаемой таблицы с горизонтальной связью. Наличие порождаемых таблиц с горизонтальной связью требует некоторого уточнения семантики выполнения раздела FROM оператора SELECT. По нашему мнению, это средство является полностью избыточным, хотя и не вредным, поскольку его реализация не должна вызывать затруднений и/или снижать эффективность системы.

Агрегатные функции, группировка и условия раздела *HAVING*

В этом разделе мы систематически обсудим все аспекты группировки таблиц и вычисления агрегатных функций. Некоторые темы уже затрагивались на неформальном уровне в предыдущих лекциях.

Семантика агрегатных функций

Агрегатные функции (в стандарте SQL они называются функциями над множествами)* определяются следующими синтаксическими правилами:

```
<set_function_specification> ::=
     COUNT(*)
   | set_function_type ([DISTINCT | ALL ] value_expression)
   | GROUPING (column_reference)
<set_function_type> ::=
   { AVG | MAX | MIN | SUM | EVERY | ANY | SOME | COUNT }
```

* Оба термина являются приемлемыми. Речь идет об *агрегатных функциях*, поскольку аргументом функции является агрегатное (составное) значение. Речь идет о *функциях над множествами*, поскольку аргументом функции является множество (в общем случае, мультимножество) значений. Но более правильно было бы говорить о *групповых функциях*, поскольку в большинстве случаев такие функции работают на значениях столбцов групп строк.

Как видно из этих правил, в стандарте SQL:1999 определены пять стандартных агрегатных функций: COUNT – число строк или значений, MAX – максимальное значение, MIN – минимальное значение, SUM – суммарное значение и AVG – среднее значение, а также две «кванторные» функции EVERY и SOME (ANY). В последних двух случаях выражение должно иметь булевский тип. Обсуждение функции GROUPING мы отложим до следующей лекции.

Агрегатные функции предназначены для того, чтобы вычислять некоторое значение для заданного мультимножества строк. Таким мультимножеством строк может быть группа строк, если агрегатная функция применяется к сгруппированной таблице, или (в вырожденных случаях) вся таблица. Для всех агрегатных функций, кроме COUNT(*), фактический (т. е. требуемый семантикой) порядок вычислений состоит в следующем. На основании параметров агрегатной функции из заданного мультимножества строк производится список значений. Затем по этому списку значений производится вычисление функции. Если список оказался пустым, то значением функции COUNT для него является 0, значением функции SOME – *false*, значением функции ALL – *true*, а значением всех остальных функций – NULL.

Пусть *T* обозначает тип значений из этого списка (вернее, «наименьший общий» тип, см. раздел «Скалярные выражения» лекции 13). Типы значений агрегатных функций определяются следующими правилами.

- Результат вычисления функции COUNT – это точное число с точностью и шкалой, которые определяются в реализации.
- Тип результата значений функций MAX и MIN совпадает с *T*. При вычислении функций SUM и AVG тип *T* не должен быть типом символьных строк.
  - Если *T* представляет собой тип точных чисел, то и типом результата функции является тип точных чисел с определяемыми в реализации точностью и шкалой.
  - Если *T* представляет собой тип приблизительных чисел, то и типом результата функции является тип приблизительных чисел с определяемой в реализации точностью.
- Для функций EVERY и SOME *T* является булевским типом.*
  - Первая функция принимает значение *true* в том и только в том случае, когда вычисление выражения-аргумента дает значение *true* для каждой строки из заданного набора строк. и *false* – когда значение выражения-аргумента есть *false* хотя бы для одной строки из заданного набора строк.
  - Функция SOME принимает значение *false* в том и только в том случае, когда значение выражения-аргумента есть *false* для каждой строки из заданного набора строк, и *true* – когда значение выра-

* Поскольку, как отмечалось в Лекции 11, в SQL к булевскому значению *uknown* принято относиться точно так же, как и к неопределенному значению, в списке значений для вычисления этих функций не останутся значения uknown.

жения-аргумента есть *true* хотя бы для одной строки из заданного набора строк.

Вычисление функции COUNT(*) производится путем подсчета числа строк в заданном мультимножестве. Все строки считаются различными, даже если они состоят из одного столбца со значением null во всех строках.*

Если «арифметическая» (AVG, MAX, MIN, SUM, COUNT) агрегатная функция специфицирована с ключевым словом DISTINCT, то множество значений, на котором она вычисляется, строится из значений указанного выражения, вычисляемого для каждой строки заданной группы строк. Затем из этого мультимножества удаляются неопределенные значения, и в нем устраняются значения-дубликаты (т. е. образуется множество). После этого вычисляется указанная функция.

Если агрегатная функция специфицирована без ключевого слова DISTINCT (или с ключевым словом ALL), то мультимножество значений формируется из значений выражения, вычисляемого для каждой строки заданной группы строк. Затем из этого мультимножества удаляются неопределенные значения, и производится вычисление агрегатной функции.

Результаты запросов и агрегатные функции

Об использовании агрегатных функций в разделах HAVING и SELECT оператора выборки упоминалось в разделе «Общие синтаксические правила построения скалярных выражений» лекции 13. В данном подразделе уместно повторить и уточнить этот материал.

Агрегатные функции можно разумным образом использовать в списке выборки (при построении выражений, являющихся элементами выборки) и в логическом выражении раздела HAVING (вернее, в выражениях, входящих в простые условия). Рассмотрим разные случаи применения агрегатных функций в списке выборки в зависимости от вида табличного выражения.

Если результат табличного выражения *R* не является сгруппированной таблицей (т. е. в табличном выражении отсутствуют разделы GROUP BY и HAVING), то появление в списке выборки хотя бы одного вызова агрегатной функции от (мульти) множества строк *R* приводит к тому, что *R* неявно рассматривается как сгруппированная таблица, состоящая из одной (или нуля, если *R* пусто) групп с отсутствующими столбцами группи-

* Обратите внимание на то, что это еще один вид различения строк в SQL и еще одна скрытая интерпретация неопределенного значения. COUNT(*) работает так, как если бы выполнялось соотношение (NULL = NULL)≡*false*. Тем самым, в SQL применяются все три возможные интерпретации NULL. При вычислении логических выражений полагается (NULL = NULL)≡*uknown*; при определении строк-дубликатов неявно считается, что (NULL = NULL) ≡*true*; наконец, при вычислении агрегатной функции COUNT(*) неявно полагается, что (NULL = NULL)≡*false*. Конечно, в такой «тройственности» нет ничего хорошего, но в контексте языка SQL приходится мириться с этим и другими негативными последствиями наличия неопределенных значений.

рования. Поэтому в данном случае в выражениях списка выборки не допускается прямое использование имен столбцов *R*: все они должны находиться внутри спецификаций вызова агрегатных функций. Результатом запроса является таблица, состоящая не более чем из одной строки, значения столбцов которой получены путем применения агрегатных функций к *R*.

Аналогично обстоит дело в том случае, когда *R* представляет собой сгруппированную таблицу, но табличное выражение не содержит раздела GROUP BY (и, следовательно, содержит раздел HAVING). В этом случае считается, что результат табличного выражения явно объявлен сгруппированной таблицей, состоящей из одной группы, и результат запроса можно формировать только путем применения агрегатных функций к данной группе строк. Опять результатом запроса является таблица, состоящая не более чем из одной строки, значения столбцов которой получены путем применения агрегатных функций к *R*.

Наконец, рассмотрим случай, когда *R* представляет собой «настоящую» сгруппированную таблицу, т. е. табличное выражение содержит раздел GROUP BY, и, следовательно, определен по крайней мере один столбец группирования (т. е. имеется хотя бы один такой столбец, что для любой группы его значения одинаковы во всех строках группы). В этом случае правила формирования списка выборки полностью соответствуют правилам формирования условия выборки раздела HAVING. Другими словами, в выражениях, являющихся элементами списка выборки, допускается прямое использование имен столбцов группирования, а спецификации остальных столбцов *R* могут появляться только внутри спецификаций агрегатных функций. Результатом запроса является таблица, число строк в которой равно числу групп в *R*. Значения столбцов каждой строки формируются на основе значений столбцов группирования и вызовов агрегатных функций для соответствующей группы.

Логические выражения раздела HAVING

Приведем примеры использования в логических выражениях раздела HAVING некоторых предикатов, обсуждавшихся в предыдущей лекции. Теоретически в этих логических выражениях можно использовать все предикаты, но применение тех предикатов, которые мы проиллюстрируем, является более естественным.

Предикаты сравнения

Пример 15.1. Найти номера отделов, в которых работает ровно 30 сотрудников.

```
SELECT DEPT_NO
FROM EMP
WHERE DEPT_NO IS NOT NULL
GROUP BY DEPT_NO
HAVING COUNT(*) = 30;
```

Конечно, этот запрос можно сформулировать и без использования разделов GROUP BY и HAVING. Например, возможна следующая формулировка (**пример 15.1а**):

```
SELECT DISTINCT DEPT_NO
FROM EMP
WHERE (SELECT COUNT (*)
        FROM EMP EMP1
        WHERE EMP1.DEPT_NO = EMP.DEPT_NO) = 30;
```

Обратите внимание, что в формулировке 15.1а отдельная проверка условия DEPT_NO IS NOT NULL не требуется.

Пример 15.2. Найти номера всех отделов, в которых средний размер зарплаты служащих превосходит 12000 руб.

```
SELECT DEPT_NO
FROM EMP
WHERE DEPT_NO IS NOT NULL
GROUP BY DEPT_NO
HAVING AVG(EMP_SAL) > 12000.00;
```

Очевидно, что и в этом случае возможна формулировка запроса без использования разделов GROUP BY и HAVING (**пример 15.2а**):

```
SELECT DISTINCT DEPT_NO
FROM EMP
WHERE (SELECT AVG(EMP1.EMP_SAL)
        FROM EMP EMP1
        WHERE EMP1.DEPT_NO = EMP.DEPT_NO) > 12000.00;
```

Немного задержимся на этих примерах и обсудим, что означает различие в формулировках запросов. В соответствии с семантикой оператора SELECT, при выполнении запросов 15.1а и 15.2а для каждой строки таблицы EMP в цикле просмотра внешнего запроса будет выполняться подзапрос, который в случае наших примеров выберет из таблицы EMP (EMP1)

все строки со значением столбца DEPT_NO, равным значению этого столбца в текущей строке внешнего цикла. Другими словами, для каждой строки внешнего цикла образуется группа, для нее проверяется условие выборки, и в списке выборки используется имя столбца этой неявной группировки. Из-за того, что группа образуется и оценивается для каждой строки таблицы EMP, мы вынуждены указать в разделе SELECT спецификацию DISTINCT.

Формулировки 15.1 и 15.2 обеспечивают более четкие указания для выполнения запроса. Нужно сразу сгруппировать таблицу EMP в соответствии со значениями столбца DEPT_NO, отобрать нужные группы, и для каждой отобранной группы вычислить значения выражений списка выборки. В этом случае семантика выполнения запроса не предписывает выполнения лишних действий. Конечно, в развитой реализации SQL компилятор должен суметь понять, что формулировки 15.1а и 15.2а эквивалентны формулировкам 15.1 и 15.2 соответственно, и избежать выполнения лишних действий.

Пример 15.3. Найти номера всех отделов, в которых суммарный объем зарплаты служащих меньше суммарного объема зарплаты всех руководителей отделов.

```
SELECT DEPT_NO
FROM EMP
WHERE DEPT_NO IS NOT NULL
GROUP BY DEPT_NO
HAVING SUM(EMP_SAL) < (SELECT SUM(EMP1.EMP_SAL)
                       FROM EMP EMP1, DEPT
                       WHERE EMP1.EMP_NO = DEPT_MNG);
```

И в этом случае возможна формулировка без использования разделов GROUP BY и HAVING **(пример 15.3а)**. Эта формулировка является более сложной, чем в случае двух предыдущих примеров, но и к ней применимы приведенные выше замечания.

```
SELECT DISTINCT DEPT_NO
FROM EMP
WHERE (SELECT SUM(EMP1.EMP_SAL)
      FROM EMP EMP1
      WHERE EMP1.DEPT_NO = EMP.DEPT_NO) <
      (SELECT SUM(EMP1.EMP_SAL)
      FROM EMP EMP1, DEPT
      WHERE EMP1.EMP_NO = DEPT_MNG);
```

Пример 15.4. Для каждого отдела найти его номер, имя руководителя, число служащих, минимальный, максимальный и средний размеры зарплаты служащих.

```
SELECT DEPT.DEPT_NO, EMP.EMP_NAME, COUNT(*),
   MIN(EMP1.EMP_SAL), MAX(EMP1.EMP_SAL), AVG(EMP1.EMP_SAL)
FROM DEPT, EMP, EMP EMP1
WHERE DEPT.DEPT_NO = EMP1.DEPT_NO
GROUP BY DEPT.DEPT_NO, DEPT.DEPT_MNG, EMP.EMP_NO, EMP.EMP_NAME
HAVING DEPT.DEPT_MNG = EMP.EMP_NO;
```

Этот запрос иллюстрирует несколько интересных особенностей языка SQL. Во-первых, это первый пример запроса с соединениями, в котором присутствуют разделы `GROUP BY` и `HAVING`. Во-вторых, одно условие соединения находится в разделе `WHERE`, а другое – в разделе `HAVING`. На самом деле, можно было бы перенести в раздел `WHERE` и второе условие соединения, и, скорее всего, на практике использовалась бы формулировка, приведенная в **примере 15.4а**:

```
SELECT DEPT.DEPT_NO, EMP.EMP_NAME, COUNT(*),
   MIN(EMP1.EMP_SAL), MAX(EMP1.EMP_SAL), AVG(EMP1.EMP_SAL)
FROM DEPT, EMP, EMP EMP1
WHERE DEPT.DEPT_NO = EMP1.DEPT_NO
   AND DEPT.DEPT_MNG = EMP.EMP_NO
GROUP BY DEPT.DEPT_NO, EMP.EMP_NAME;
```

Но первая формулировка тоже верна, поскольку второе условие соединения определено на столбцах группировки.

Наконец, легко видеть, что по существу группировка производится по значениям столбца `DEPT.DEPT_NO`. Остальные столбцы, указанные в списке столбцов группировки, функционально определяются столбцом `DEPT.DEPT_NO`. Тем не менее, в первой формулировке мы включили в этот список столбцы `DEPT.DEPT_MNG` и `EMP.EMP_NO`, чтобы их имена можно было использовать в условии раздела `HAVING`, и столбец `EMP.EMP_NAME`, чтобы можно было использовать его имя в списке выборки раздела `SELECT`. Другими словами, мы вынуждены расширять запрос избыточными данными, чтобы выполнить формальные синтаксические требования языка. Как видно, во второй формулировке мы смогли удалить из списка группировки два столбца. Кстати, не следует думать, что многословие первой формулировки помешает СУБД выполнить запрос настолько же эффективно, как запрос во второй формулировке. Грамотно построенный оптимизатор SQL сам приведет первую формулировку ко второй.

Наконец, и этот запрос можно сформулировать без использования раздела GROUP BY за счет использования подзапросов в списке раздела SELECT (**пример 15.4b**):

```
SELECT DEPT.DEPT_NO, EMP.EMP_NAME,
      (SELECT COUNT(*)
      FROM EMP
      WHERE EMP.DEPT_NO = DEPT.DEPT_NO),
      (SELECT MIN(EMP_SAL)
      FROM EMP
      WHERE EMP.DEPT_NO = DEPT.DEPT_NO),
      (SELECT MAX(EMP_SAL)
      FROM EMP
      WHERE EMP.DEPT_NO = DEPT.DEPT_NO),
      (SELECT AVG(EMP_SAL)
      FROM EMP
      WHERE EMP.DEPT_NO = DEPT.DEPT_NO)
FROM DEPT, EMP
WHERE DEPT.DEPT_MNG = EMP.EMP_NO;
```

Здесь мы снова имеем замаскированную группировку строк по значениям столбца DEPT.DEPT_NO и вычисление агрегатных функций для каждой группы. Формально группа строится каждый раз заново при вызове каждой агрегатной функции. Хороший компилятор SQL должен привести формулировку 15.4b к виду 15.4a.

И последнее замечание. Во всех приведенных формулировках в результат не попадут данные об отделах, в которых отсутствует руководитель (столбец DEPT.DEPT_MNG может содержать неопределенное значение). Вообще говоря, это не противоречит условию запроса, но если бы мы хотели выдавать в результате NULL в качестве имени руководителя отдела с отсутствующим руководителем, то можно было немного усложнить формулировку запроса, например, следующим образом (**пример 15.4c**):

```
SELECT DEPT.DEPT_NO,
   CASE WHEN DEPT.DEPT_MNG IS NULL THEN NULL
      ELSE (SELECT EMP.EMP_NAME
            FROM EMP
            WHERE EMP.EMP_NO = DEPT.DEPT_MNG),
      COUNT(*), MIN(EMP1.EMP_SAL),
      MAX(EMP1.EMP_SAL), AVG(EMP1.EMP_SAL)
FROM DEPT, EMP, EMP EMP1
WHERE DEPT.DEPT_NO = EMP1.DEPT_NO
GROUP BY DEPT.DEPT_NO;
```

Предикат between

Пример 15.5. Найти номера отделов и минимальный и максимальный размер зарплаты служащих для отделов, в которых средний размер зарплаты служащих не меньше среднего размера зарплаты служащих во всей компании и не больше 30000 руб.

```
SELECT DEPT_NO, MIN(EMP_SAL), MAX(EMP_SAL)
FROM EMP
WHERE DEPT_NO IS NOT NULL
GROUP BY DEPT_NO
HAVING AVG(EMP_SAL) BETWEEN
   (SELECT AVG(EMP_SAL)
   FROM EMP) AND 30000.00;
```

Еще раз приведем возможную формулировку этого запроса без использования разделов GROUP BY и HAVING **(пример 15.5а)**:

```
SELECT DISTINCT DEPT_NO, (SELECT MIN(EMP1.EMP_SAL)
                          FROM EMP EMP1
                          WHERE EMP1.DEPT_NO = EMP.DEPT_NO),
                          (SELECT MAX(EMP1.EMP_SAL)
                          FROM EMP EMP1
                          WHERE EMP1.DEPT_NO = EMP.DEPT_NO)
FROM EMP
WHERE (SELECT AVG(EMP1.EMP_SAL)
      FROM EMP EMP1
      WHERE EMP1.DEPT_NO = EMP.DEPT_NO) BETWEEN
   (SELECT AVG(EMP_SAL)
      FROM EMP) AND 30000.00;
```

Как видно, отказ от использования раздела GROUP BY приводит к размножению однотипных подзапросов, строящих одну и ту же группу строк, над которой вычисляется агрегатная функция.

Предикат null

Пример 15.6. Найти номера и число служащих отделов, данные о руководителях которых не содержат номер отдела (конечно, в этом случае нас интересуют только те отделы, у которых имеется руководитель).

```
SELECT DEPT.DEPT_NO, COUNT(*)
FROM DEPT, EMP EMP1, EMP EMP2
WHERE DEPT.DEPT_NO = EMP2.DEPT_NO
```

```
 AND DEPT.DEPT_MNG = EMP1.EMP_NO
GROUP BY DEPT.DEPT_NO, EMP1.DEPT_NO
HAVING EMP1.DEPT_NO IS NULL;
```

Как и в примере 15.4, условие раздела HAVING можно переместить в раздел WHERE и получить вторую формулировку (**пример 15.6a**):

```
SELECT DEPT.DEPT_NO, COUNT(*)
FROM DEPT, EMP EMP1, EMP EMP2
WHERE DEPT.DEPT_NO = EMP2.DEPT_NO AND
   DEPT.DEPT_MNG = EMP1.EMP_NO AND
   EMP1.DEPT_NO IS NULL
GROUP BY DEPT.DEPT_NO;
```

Кстати, в этом случае, поскольку в запросе присутствует только один вызов агрегатной функции, формулировка без использования раздела GROUP BY оказывается более понятной и не менее эффективной (даже при следовании предписанной семантике выполнения оператора SELECT), что показывает **пример 15.6b**:

```
SELECT DEPT.DEPT_NO, (SELECT COUNT(*)
                      FROM EMP
                      WHERE DEPT.DEPT_NO = EMP.DEPT_NO)
FROM DEPT, EMP
WHERE DEPT.DEPT_MNG = EMP.EMP_NO AND
     EMP.DEPT_NO IS NULL;
```

Предикат in

Пример 15.7. Найти номера отделов, в которых средний размер зарплаты служащих равен максимальному размеру зарплаты служащих какого-либо другого отдела.

```
SELECT DEPT.DEPT_NO
FROM DEPT, EMP
WHERE DEPT.DEPT_NO = EMP.DEPT_NO
GROUP BY DEPT.DEPT_NO
HAVING AVG(EMP.EMP_SAL) IN
   (SELECT MAX(EMP1.EMP_SAL)
      FROM EMP, DEPT DEPT1
      WHERE EMP.DEPT_NO = DEPT1.DEPT_NO
      AND DEPT1.DEPT_NO <> DEPT.DEPT_NO
      GROUP BY DEPT.DEPT_NO);
```

Этот запрос, помимо прочего, демонстрирует наличие в условии раздела HAVING вложенного подзапроса с корреляцией. Как и раньше, можно избавиться от разделов GROUP BY и HAVING во внешнем запросе **(пример 15.7а)**:

```
SELECT DEPT.DEPT_NO
FROM DEPT
WHERE (SELECT AVG(EMP_SAL)
      FROM EMP
      WHERE EMP.DEPT_NO = DEPT.DEPT_NO) IN
      (SELECT MAX(EMP1.EMP_SAL)
      FROM EMP, DEPT DEPT1
      WHERE EMP.DEPT_NO = DEPT1.DEPT_NO
         AND DEPT1.DEPT_NO <> DEPT.DEPT_NO
      GROUP BY DEPT.DEPT_NO);
```

Но в данном случае мы не можем отказаться от раздела GROUP BY во втором вложенном запросе, поскольку без этого невозможно получить множество значений результатов вызова агрегатной функции.

Предикат like

Пример 15.8. Во всех отделах найти имена и число служащих, у которых в данном отделе имеются однофамильцы и фамилии которых начинаются со строки символов, изображающей фамилию руководителя отдела.

```
SELECT EMP_NAME, COUNT(*)
FROM EMP, DEPT
WHERE EMP.DEPT_NO = DEPT.DEPT_NO
GROUP BY DEPT.DEPT_NO, EMP_NAME
HAVING COUNT(*) > 1
   AND EMP.EMP_NAME LIKE (SELECT EMP1.EMP_NAME
              FROM EMP EMP1
              WHERE EMP1.EMP_NO = DEPT.DEPT_MNG) || '%';
```

Конечно, и в этом случае условие с предикатом LIKE можно переместить из раздела HAVING в раздел WHERE. Этот запрос можно переформулировать в виде, лишенном разделов GROUP BY и HAVING **(пример 15.8а)**, но вряд ли это разумно, поскольку формулировка является менее понятной и существенно более сложной.

```
SELECT EMP_NAME, (SELECT COUNT(*)
                  FROM EMP EMP1
                  WHERE EMP1.DEPT_NO = EMP.DEPT_NO
                     AND EMP1.EMP_NAME = EMP.EMP_NAME
```

```
                        AND EMP1.EMP_NO <> EMP.EMP_NO) + 1
FROM EMP
WHERE (SELECT COUNT(*)
        FROM EMP EMP1
        WHERE EMP1.DEPT_NO = EMP.DEPT_NO
           AND EMP1.EMP_NAME = EMP.EMP_NAME
           AND EMP1.EMP_NO <> EMP.EMP_NO) > 1
 AND EMP_NAME LIKE (SELECT EMP1.EMP_NAME
                      FROM EMP EMP1, DEPT
                      WHERE EMP.DEPT_NO = DEPT.DEPT_NO
                      AND EMP1.EMP_NO = DEPT.DEPT_MNG) || '%';
```

Предикат exists

Пример 15.9. Найти номера отделов, в которых средний размер зарплаты служащих равен максимальному размеру зарплаты служащих какого-либо другого отдела (другая формулировка для примера 15.7).

```
SELECT DEPT.DEPT_NO
FROM DEPT, EMP
WHERE DEPT.DEPT_NO = EMP.DEPT_NO
GROUP BY DEPT.DEPT_NO
HAVING EXISTS (SELECT *
          FROM EMP EMP1
          WHERE EMP1.DEPT_NO <> DEPT.DEPT_NO
          GROUP BY EMP1.DEPT_NO
          HAVING MAX (EMP1.EMP_SAL)= AVG (EMP.EMP_SAL));
```

В этой формулировке основной интерес представляет подзапрос, в котором корреляция с внешним запросом происходит через вызов агрегатной функции от группы строк внешнего запроса. Здесь также можно избавиться от разделов GROUP BY и HAVING во внешнем запросе (**пример 15.9а**):

```
SELECT DEPT.DEPT_NO
FROM DEPT
WHERE EXISTS (SELECT EMP.DEPT_NO
                FROM EMP
                WHERE EMP.DEPT_NO <> DEPT.DEPT_NO
                GROUP BY EMP.DEPT_NO
                HAVING MAX (EMP.EMP_SAL)=
                   (SELECT AVG (EMP1.EMP_SAL)
                      FROM EMP EMP1
                      WHERE EMP1.DEPT_NO = DEPT.DEPT_NO));
```

Предикат unique

Пример 15.10. Найти номера отделов и средний размер зарплаты сотрудников для таких отделов, где средний размер зарплаты служащих отличается от среднего размера зарплаты всех других отделов.

```
SELECT DEPT.DEPT_NO, AVG (EMP.EMP_SAL)
FROM DEPT, EMP
WHERE DEPT.DEPT_NO = EMP.DEPT_NO
GROUP BY DEPT.DEPT_NO
HAVING UNIQIUE (SELECT AVG (EMP1.EMP_SAL)
          FROM EMP EMP1
          WHERE EMP1.DEPT_NO IS NOT NULL
          GROUP BY EMP1.DEPT_NO
          HAVING AVG (EMP1.EMP_SAL) = AVG (EMP.EMP_SAL));
```

Вот альтернативная формулировка этого запроса с использованием предиката NOT EXISTS **(пример 15.10а)**:

```
SELECT DEPT.DEPT_NO, AVG (EMP.EMP_SAL)
FROM DEPT, EMP
WHERE DEPT.DEPT_NO = EMP.DEPT_NO
GROUP BY DEPT.DEPT_NO
HAVING NOT EXISTS (SELECT EMP1.DEPT_NO
          FROM EMP EMP1
          WHERE EMP1.DEPT_NO <> DEPT.DEPT_NO
          GROUP BY EMP1.DEPT_NO
          HAVING AVG (EMP1.EMP_SAL)= AVG (EMP.EMP_SAL));
```

Предикаты сравнения с квантором

Пример 15.11. Найти номера отделов и средний возраст служащих для таких отделов, что найдется хотя бы один другой отдел, средний возраст служащих которого больше, чем в данном.

```
SELECT DEPT_NO, AVG (CURRENT_DATE - EMP_BDATE)
FROM EMP
WHERE DEPT_NO IS NOT NULL
GROUP BY DEPT_NO
HAVING AVG (CURRENT_DATE - EMP_BDATE)< SOME
   (SELECT AVG (CURRENT_DATE - EMP1.EMP_BDATE)
     FROM EMP EMP1
     WHERE EMP1.DEPT_NO IS NOT NULL
     GROUP BY EMP1.DEPT_NO);
```

Напомним, что «ниладическая» функция CURRENT_DATE выдает текущую дату, и, следовательно, значением выражения CURRENT_DATE – EMP_BDATE является интервал, представляющий текущий возраст служащего. На наш взгляд, формулировка этого запроса несколько упрощается, если пользоваться предикатом EXISTS **(пример 15.11а)**:

```
SELECT DEPT_NO, AVG (CURRENT_DATE - EMP_BDATE)
FROM EMP
WHERE DEPT_NO IS NOT NULL
GROUP BY DEPT_NO
HAVING EXISTS (SELECT EMP1.DEPT_NO
               FROM EMP EMP1
               WHERE EMP1.DEPT_NO IS NOT NULL
               GROUP BY EMP1.DEPT_NO
               HAVING AVG (CURRENT_DATE - EMP1.EMP_BDATE) >
                  AVG (CURRENT_DATE - EMP.EMP_BDATE));
```

Пример 15.12. Найти номера отделов и средний возраст служащих для отделов с минимальным средним возрастом служащих.

```
SELECT DEPT_NO, AVG (CURRENT_DATE - EMP_BDATE)
FROM EMP
WHERE DEPT_NO IS NOT NULL
GROUP BY DEPT_NO
HAVING AVG (CURRENT_DATE - EMP_BDATE) <= ALL
   (SELECT AVG (CURRENT_DATE - EMP_BDATE)
      FROM EMP
      WHERE DEPT_NO IS NOT NULL
      GROUP BY DEPT_NO);
```

Этот запрос легко формулируется в более понятном виде с использованием предиката NOT EXISTS **(пример 15.12а)**:

```
SELECT DEPT_NO, AVG (CURRENT_DATE - EMP_BDATE)
FROM EMP
WHERE DEPT_NO IS NOT NULL
GROUP BY DEPT_NO
HAVING NOT EXISTS (SELECT EMP1.DEPT_NO
                   FROM EMP EMP1
                   WHERE EMP1.DEPT_NO IS NOT NULL
                   GROUP BY EMP1.DEPT_NO
                   HAVING AVG (CURRENT_DATE - EMP1.EMP_BDATE) <
                        AVG (CURRENT_DATE - EMP.EMP_BDATE));
```

Предикат `distinct`

Пример 15.13. Найти номера отделов, которые можно отличить от любого другого отдела по дате рождения руководителя и среднему размеру зарплаты.

```
SELECT DEPT.DEPT_NO
FROM DEPT, EMP EMP1, EMP EMP2
WHERE DEPT.DEPT_NO = EMP1.DEPT_NO AND
   DEPT.DEPT_MNG = EMP2.EMP_NO
GROUP BY DEPT.DEPT_NO, EMP2.EMP_BDATE
HAVING (EMP2.EMP_BDATE, AVG (EMP1.EMP_SAL)) DISTINCT FROM
   (SELECT EMP2.EMP_BDATE, AVG (EMP1.EMP_SAL)
      FROM DEPT DEPT1, EMP EMP1, EMP EMP2
      WHERE DEPT1.DEPT_NO = EMP1.DEPT_NO AND
      DEPT1.DEPT_MNG = EMP2.EMP_NO AND
      DEPT1.DEPT_NO <> DEPT.DEPT_NO
      GROUP BY DEPT.DEPT_NO, EMP2.EMP_BDATE);
```

Ссылки на порождаемые таблицы в разделе `FROM`

В этом разделе мы приведем несколько примеров запросов, в разделе `FROM` которых содержатся выражения запросов (ссылки на порождаемые таблицы, см. раздел «Общие синтаксические правила построения скалярных выражений» лекции 13).

Еще один способ формулировки запросов

Прежде всего, на простом примере покажем, как использование ссылок на порождаемые таблицы расширяет возможности формулировки запросов.

Пример 15.14. Найти номера отделов и имена руководителей отделов, которые числятся в тех же отделах, которыми руководят, и получают зарплату, размер которой является максимальным для служащих данного отдела.

```
SELECT MNG.DEPT_NO, MNG.MNG_NAME
FROM (SELECT DEPT.DEPT_NO, EMP.DEPT_NO, EMP_NAME, EMP_SAL
        FROM DEPT, EMP
        WHERE DEPT.DEPT_MNG = EMP.EMP_NO)
        AS MNG (DEPT_NO_1, DEPT_NO_2, MNG_NAME, MNG_SAL)
WHERE DEPT_NO_1 = DEPT_NO_2
   AND MNG_SAL = (SELECT MAX (EMP_SAL)
                    FROM EMP
                    WHERE EMP.DEPT_NO = DEPT_NO_1);
```

В этом запросе порождаемая таблица MNG содержит по одной строке для каждого служащего, являющегося руководителем отдела. Первый столбец этой таблицы – DEPT_NO_1 – содержит номер отдела, которым руководит данный служащий. В столбце DEPT_NO_1 хранятся номера отделов, в которых числятся руководители отделов, а в столбцах EMP_NAME и EMP_SAL содержатся имя служащего-руководителя отдела и размер его заработной платы соответственно.

Конечно, этот запрос можно сформулировать и без использования ссылки на порождаемую таблицу в разделе FROM, например, следующим образом **(пример 15.14a)**:

```
SELECT DEPT.DEPT_NO, EMP.EMP_NAME
FROM DEPT, EMP
   WHERE DEPT.DEPT_MNG = EMP.EMP_NO
      AND DEPT.DEPT_NO = EMP.DEPT_NO
      AND EMP.EMP_SAL = (SELECT MAX(EMP_SAL)
                         FROM EMP
                         WHERE EMP.DEPT_NO = DEPT.DEPT_NO);
```

А вот как можно сформулировать тот же запрос с использованием раздела WITH **(пример 15.14b)**:

```
WITH MNG (DEPT_NO_1, DEPT_NO_2, MNG_NAME, MNG_SAL) AS
   (SELECT DEPT.DEPT_NO, EMP.DEPT_NO, EMP_NAME, EMP_SAL
   FROM DEPT, EMP
   WHERE DEPT.MNG_NO = EMP.EMP_NO),
   MAX_DEPT_SAL (MAX_SAL, DEPT_NO) AS
   (SELECT MAX (EMP_SAL), DEPT_NO
   FROM EMP
   WHERE DEPT_NO IS NOT NULL
   GROUP BY DEPT_NO)
SELECT DEPT_NO_1, MNG_NAME
FROM MNG
WHERE DEPT_NO_1 = DEPT_NO_2
   AND MNG_SAL = (SELECT MAX_SAL
      FROM MAX_DEPT_SAL
      WHERE MAX_DEPT_SAL.DEPT_NO = DEPT_NO_1);
```

Случаи, в которых без порождаемых таблиц обойтись невозможно

На самом деле, пример 15.14 демонстрирует лишь возможность альтернативных формулировок запросов с использованием ссылок на порождаемые таблицы в разделе FROM. Но в некоторых случаях без подобных конструкций просто невозможно обойтись. Вот простой пример.

Пример 15.15. Найти общее число служащих и максимальный размер зарплаты в отделах с одинаковым максимальным размером зарплаты.

```
SELECT SUM (TOTAL_EMP), MAX_SAL
FROM (SELECT MAX (EMP_SAL), COUNT (*)
   FROM EMP
   WHERE DEPT_NO IS NOT NULL
   GROUP BY DEPT_NO ) AS DEPT_MAX_SAL (MAX_SAL, TOTAL_EMP)
GROUP BY MAX_SAL;
```

И в этом случае выражение запросов, содержащееся в разделе FROM, можно перенести в раздел WITH **(пример 15.15а)**:

```
WITH DEPT_MAX_SAL (MAX_SAL, TOTAL_EMP) AS
   (SELECT MAX (EMP_SAL), COUNT (*)
      FROM EMP
      WHERE DEPT_NO IS NOT NULL
      GROUP BY DEPT_NO)
SELECT SUM (TOTAL_EMP), MAX_SAL
FROM DEPT_MAX_SAL
GROUP BY MAX_SAL;
```

Здесь мы не можем обойтись «одноуровневой» конструкцией запроса, поскольку требуется двойная группировка, причем вторая группировка должна быть получена в соответствии с результатами первой. Еще один пример.

Пример 15.16. Найти число проектов, дату их завершения и средний размер зарплаты служащих, участвующих в проекте, для проектов с одной и той же датой завершения и одним и тем же средним размером зарплаты служащих, участвующих в проекте.

```
SELECT COUNT (*), PRO_EDATE, AVG_SAL
FROM (SELECT PRO_EDATE, AVG (EMP_SAL)
   FROM (SELECT PRO_SDATE + PRO_DURAT, PRO_NO
      FROM PRO) AS PRO1 (PRO_EDATE, PRO_NO), EMP
   WHERE PRO1.PRO_NO = EMP.PRO_NO
   GROUP BY PRO1.PRO_NO ) AS PRO_AVG_SAL (PRO_EDATE, AVG_SAL)
GROUP BY PRO_EDATE, AVG_SAL;
```

Заметим, что выражение запросов на третьей и четвертой строках примера необходимо только по той причине, что нам требуется группировка по дате окончания проектов, соответствующий столбец в таблице PRO отсутствует, а в списке группировки можно использовать только имена столбцов. Для упрощения вида формулировки это выражение разумно вынести в раздел WITH **(пример 15.16а)**:

```
WITH PRO1 (PRO_EDATE, PRO_NO) AS
   (SELECT PRO_SDATE + PRO_DURAT, PRO_NO
      FROM PRO)
SELECT COUNT (*), PRO_EDATE, AVG_SAL
FROM (SELECT PRO_EDATE, AVG (EMP_SAL)
      FROM PRO1, EMP
      WHERE PRO1.PRO_NO = EMP.PRO_NO
      GROUP BY PRO1.PRO_NO) AS PRO_AVG_SAL (PRO_EDATE, AVG_SAL)
GROUP BY PRO_EDATE, AVG_SAL;
```

Более сложные конструкции оператора выборки

В этом разделе мы обсудим возможности языка SQL, касающиеся явного задания выражений с соединениями и порождаемых таблиц с горизонтальной связью (lateral_derived_table). Начнем с соединений.

Соединенные таблицы

В примерах предыдущей и данной лекций присутствовало много запросов с соединениями двух или более таблиц. Условия соединения задавались предикатами сравнения столбцов таблиц, специфицированных в разделе FROM, и входили в состав логических выражений раздела WHERE (или, реже, раздела HAVING). Поскольку на практике требуются разные виды соединений, в стандарте SQL/92 появилась альтернативная возможность спецификации соединений – соединенная таблица (joined table). Соответствующая конструкция может использоваться в разделе FROM выражения запросов и фактически позволяет строить выражения соединений таблиц. Синтаксические правила построения таких выражений выглядят следующим образом:

```
joined_table ::= cross_join
               | qualified_join
               | natural_join
               | union_join
cross_join ::= table_reference CROSS JOIN table_primary
qualified_join ::= table_reference [ join_type ] JOIN
                   table_primary join_specification
natural_join ::= table_reference NATURAL [ join_type ]
                   JOIN table_primary
union_join ::= table_reference UNION JOIN table_primary
join_type ::= INNER | { LEFT | RIGHT | FULL } [ OUTER ]
join_specification ::= ON conditional_expression
                     | USING (column_comma_list)
```

Напомним, что синтаксические правила для table_reference и table_primary были показаны в лекции 13.

Как показывает сводка синтаксических правил, в SQL поддерживается много вариантов соединений. Чтобы объяснить особенности разных видов соединений на неформальном уровне, требуется очень большой объем текста с большим числом повторений. Поэтому сначала мы приведем достаточно формальное описание порядка определения заголовка и тела результирующей таблицы для всех разновидностей соединений. Фактически это описание напрямую позаимствовано из стандарта SQL:1999 с некоторыми незначительными упрощениями. Затем мы представим ряд иллюстрирующих примеров.

Формальные определения

Пусть требуется выполнить некоторую операцию соединения над таблицами *table1* и *table2*. Тогда:

- Обозначим через CP результат выполнения запроса

```
SELECT *
FROM table1, table2*
```

- Если задается операция JOIN (или NATURAL JOIN) без явного указания типа соединения (join_type), то по умолчанию имеется в виду INNER JOIN (или NATURAL INNER JOIN).
- Если в спецификации соединения (join_specification) указано ключевое слово ON, то все ссылки на столбцы, встречающиеся в условном выражении (conditional_expression), должны указывать на столбцы таблиц *table1* и *table2* или на столбцы таблиц внешнего запроса. Если в этом условном выражении присутствует вызов агрегатной функции, то соединенная таблица может фигурировать только в подзапросах, используемых в разделах HAVING или SELECT внешнего запроса, и ссылка на столбец в вызове функции должна указывать на столбец таблицы внешнего запроса.
- Для прямых соединений (CROSS JOIN) и всех других видов соединения, включающих раздел ON, заголовок результата операции совпадает с заголовком таблицы *CP*.
- Если в спецификации вида соединения присутствуют ключевые слова NATURAL или USING, то заголовок результата операции определяется следующим образом:
  - если в спецификации вида соединения присутствует ключевое слово NATURAL, то будем называть *соответствующими столбцами соединения* (corresponding join column) все столбцы таблиц

* Интересно, что для этого запроса возможна альтернативная формулировка с использованием операции CROSS JOIN: SELECT * FROM table1 CROSS JOIN table2. Может возникнуть естественный вопрос: зачем вводить специальную конструкцию для декартова произведения? По мнению автора, эта конструкция была введена, главным образом, для повышения уровня общности языка SQL. Кроме того, использование явного ключевого слова CROSS JOIN является подтверждением того, что пользователь действительно желает получить декартово произведение, а не опустил по ошибке раздел WHERE.

table1 и *table2*, которые имеют в заголовках этих таблиц одинаковые имена. Если в спецификации вида соединения присутствует ключевое слово USING, то будем называть *соответствующими столбцами соединения (corresponding join column)* все столбцы таблиц *table1* и *table2*, имена которых входят в список имен столбцов раздела USING (эти столбцы должны быть одноименными в заголовках обеих таблиц). В обоих случаях типы данных каждой пары соответствующих столбцов должны быть совместимыми;

- будем называть *списком выборки соответствующих столбцов соединения* (select_list of corresponding join columns – *SLCC*) список элементов вида COALESCE (table1.*c*, table2.*c*) AS *c**, где *c* является именем соответствующего столбца соединения. Элементы располагаются в том порядке, в котором они появляются в заголовке таблицы *table1*. Обозначим через *SLT1* (*SLT2*) список имен столбцов таблицы *table1* (*table2*), которые не являются соответствующими столбцами соединения. Имена располагаются в том же порядке, в котором они появляются в заголовке соответствующей таблицы;
- заголовок результата совпадает с заголовком результата запроса

```
SELECT SLCC, SLT1, SLT2
FROM table1, table2;
```

- Набор строк результата (множество или мультимножество) определяется по следующим правилам. Обозначим через *T* следующие наборы строк:
  - если видом соединения является UNION JOIN, то *T* – пусто;
  - если видом соединения является CROSS JOIN, то *T* включает все строки, входящие в *CP*;
  - если в спецификацию вида соединения входит раздел ON, то *T* включает все строки *CP*, для которых результатом вычисления условного выражения является *true*;
  - если в спецификацию вида соединения входят разделы NATURAL или USING, и список *SLCC* не является пустым, то *T* включает все строки *CP*, для которых значения соответствующих столбцов соединения совпадают;**
  - если в спецификацию вида соединения входят разделы NATURAL или USING, и список *SLCC* является пустым, то *T* включает все строки *CP*.
- Обозначим через *P1* (*P2*) набор (множество или мультимножество) всех строк таблицы *table1* (*table2*), каждая из которых участвует в образовании некой строки *T*.

* Для удобства читателей напомним, что по определению выражение COALESCE (V1, V2) эквивалентно следующему выражению с переключателем: CASE WHEN V1 IS NOT NULL THEN V1 ELSE V2 END.

** Совпадают в строгом смысле, т.е. значение столбца table1.*c* совпадает со значением столбца table2.*c* тогда и только тогда, когда значение операции сравнения table1.*c* = table2.*c* является *true*.

- Обозначим через *U1* (*U2*) набор (множество или мультимножество) всех строк таблицы *table1* (*table2*), ни одна из которых не участвует в образовании какой-либо строки *T*.
- Обозначим через *X1* набор (множество или мультимножество) всех строк, образуемых из строк набора *U1* путем добавления справа подстроки из неопределенных значений, содержащей столько неопределенных значений, сколько столбцов содержит таблица *table2*. Обозначим через *X2* набор (множество или мультимножество) всех строк, образуемых из строк набора *U2* путем добавления слева подстроки из неопределенных значений, содержащей столько неопределенных значений, сколько столбцов содержит таблица *table1*.
- Для соединений вида CROSS JOIN и INNER JOIN пусть *S* обозначает тот же набор строк, что и *T*.
- Для соединений вида LEFT OUTER JOIN пусть *S* обозначает набор строк, являющийся результатом выражения запросов

```
SELECT * FROM T
UNION ALL
SELECT * FROM X1;
```

- Для соединений вида RIGHT OUTER JOIN пусть *S* обозначает набор строк, являющийся результатом выражения запросов

```
SELECT * FROM T
UNION ALL
SELECT * FROM X2;
```

- Для соединений вида FULL OUTER JOIN пусть *S* обозначает набор строк, являющийся результатом выражения запросов

```
SELECT * FROM T
UNION ALL
SELECT * FROM X1
UNION ALL
SELECT * FROM X2;
```

- Для соединений вида UNION JOIN пусть *S* обозначает набор строк, являющийся результатом выражения запросов

```
SELECT * FROM X1
UNION ALL
SELECT * FROM X2;
```

- Если в спецификации вида соединения присутствуют ключевые слова NATURAL или USING, то результат операции совпадает с результатом выражения запросов

```
SELECT SLCC, SLT1, SLT2
FROM S;
```

- Во всех остальных случаях результат операции совпадает с *S*.

Примеры соединений разного вида

Основное назначение приводимых ниже примеров состоит не в том, чтобы продемонстрировать практическую значимость разнообразных соединений, а лишь в том, чтобы помочь в них разобраться.* Поэтому мы будем использовать упрощенные и формальные таблицы и показывать заголовки и тела результирующих таблиц.

Итак, пусть имеются таблицы *table1 (a1, a2, c1, c2)* и *table2 (b1, b2, c1, c2)* со следующими телами:

table1

a1	*a2*	*c1*	*c2*
1	1	1	1
1	1	2	3
1	1	2	3
2	3	4	NULL
3	NULL	NULL	5

table2

b1	*b2*	*c1*	*c2*
1	1	1	1
1	2	2	3
3	3	2	3
4	4	4	4
3	NULL	NULL	5
3	NULL	NULL	5

Обозначим через *JR* таблицу, являющуюся результатом соединения. Тогда для операции `table1 INNER JOIN table2 ON a1=b1 AND a2<b2` (*внутреннее соединение по условию*) тело *JR* будет следующим:

JR

a1	*a2*	*table1.c1*	*table1.c2*	*b1*	*b2*	*table2.c1*	*table2.c2*
1	1	1	1	1	2	2	3
1	1	2	3	1	2	2	3
1	1	2	3	1	2	2	3

Строки-дубликаты появились в *JR*, поскольку в первом операнде присутствовали строки-дубликаты, удовлетворяющие условию соединения.

Результатом операции *table1* `INNER JOIN` *table2* `USING (c2)` (*внутреннее соединение по совпадению значений указанных одноименных столбцов*) будет следующая таблица.

JR

a1	*a2*	*table1.c1*	*c2*	*b1*	*b2*	*table2.c1*
1	1	1	1	1	1	1
1	1	2	3	1	2	2
1	1	2	3	3	3	2
1	1	2	3	1	2	2
1	1	2	3	3	3	2
3	NULL	NULL	5	3	NULL	NULL
3	NULL	NULL	5	3	NULL	NULL

* За очевидностью мы опустим пример `CROSS JOIN`.

Результат операции *table1* INNER JOIN *table2* USING (*c1,c2*):

JR

a1	*a2*	*c1*	*c2*	*b1*	*b2*
1	1	1	1	1	1
1	1	2	3	1	2
1	1	2	3	3	3
1	1	2	3	1	2
1	1	2	3	3	3

Такой же результат будет получен при выполнении операции *table1* NATURAL INNER JOIN *table2* (***естественное внутреннее соединение***). Более того, для произвольных таблиц *table1* и *table2* результаты операций *table1* INNER JOIN *table2* USING (*c1, c2, ...cn*) и *table1* INNER NATURAL JOIN *table2* совпадают в том и только в том случае, когда список имен столбцов *c1, c2, ... cn* включает все имена столбцов, общие для таблиц *table1* и *table2*.

Результатом операции *table1* LEFT OUTER JOIN *table2* ON *a1=b1* AND *a2<b2* (***левое внешнее соединение по условию***) будет следующая таблица

JR

a1	*a2*	*table1.c1*	*table1.c2*	*b1*	*b2*	*table2.c1*	*table2.c2*
1	1	1	1	1	2	2	3
1	1	2	3	1	2	2	3
1	1	2	3	1	2	2	3
2	3	4	NULL	NULL	NULL	NULL	NULL
3	NULL	NULL	5	NULL	NULL	NULL	NULL

Как видно, в результате левого внешнего соединения сохраняются все данные первого (левого) операнда.

Результатом операции *table1* RIGHT OUTER JOIN *table2* ON *a1=b1* AND *a2<b2* (***правое внешнее соединение по условию***) будет следующая таблица

JR

a1	*a2*	*table1.c1*	*table1.c2*	*b1*	*b2*	*table2.c1*	*table2.c2*
1	1	1	1	1	2	2	3
1	1	2	3	1	2	2	3
1	1	2	3	1	2	2	3
NULL	NULL	NULL	NULL	1	1	1	1
NULL	NULL	NULL	NULL	3	3	2	3
NULL	NULL	NULL	NULL	4	4	4	4
NULL	NULL	NULL	NULL	3	NULL	NULL	5
NULL	NULL	NULL	NULL	3	NULL	NULL	5

Как видно, в результате правого внешнего соединения сохраняются все данные второго (правого) операнда.

Результатом операции *table1* FULL OUTER JOIN *table2* ON *a1=b1* AND *a2<b2* (***полное внешнее соединение по условию***) будет следующая таблица:

JR

a1	*a2*	*table1.c1*	*table1.c2*	*b1*	*b2*	*table2.c1*	*table2.c2*
1	1	1	1	1	2	2	3
1	1	2	3	1	2	2	3
1	1	2	3	1	2	2	3
2	3	4	NULL	NULL	NULL	NULL	NULL
3	NULL	NULL	5	NULL	NULL	NULL	NULL
NULL	NULL	NULL	NULL	1	1	1	1
NULL	NULL	NULL	NULL	3	3	2	3
NULL	NULL	NULL	NULL	4	4	4	4
NULL	NULL	NULL	NULL	3	NULL	NULL	5
NULL	NULL	NULL	NULL	3	NULL	NULL	5

Как видно, в результате полного внешнего соединения сохраняются данные обоих операндов. Кстати, полное внешнее соединение иногда называют еще ***симметричным внешним соединением***. Очевидно, что все операции внутреннего соединения и операция полного внешнего соединения коммутативны, а операции левого и правого соединения коммутативными не являются.

Результатом операции *table1* LEFT OUTER JOIN *table2* USING (*c2*) (***левое внешнее соединение по совпадению значений указанных одноименных столбцов***) будет следующая таблица:

JR

a1	*a2*	*table1.c1*	*c2*	*b1*	*b2*	*table2.c1*
1	1	1	1	1	1	1
1	1	2	3	1	2	2
1	1	2	3	3	3	2
1	1	2	3	1	2	2
1	1	2	3	3	3	2
3	NULL	NULL	5	3	NULL	NULL
3	NULL	NULL	5	3	NULL	NULL
2	3	4	NULL	NULL	NULL	NULL

Результатом операции *table1* RIGHT OUTER JOIN *table2* USING (*c2*) (***правое внешнее соединение по совпадению значений указанных одноименных столбцов***) будет следующая таблица:

JR

a1	*a2*	*table1.c1*	*c2*	*b1*	*b2*	*table2.c1*
1	1	1	1	1	1	1
1	1	2	3	1	2	2
1	1	2	3	3	3	2
1	1	2	3	1	2	2
1	1	2	3	3	3	2
3	NULL	NULL	5	3	NULL	NULL
3	NULL	NULL	5	3	NULL	NULL
NULL	NULL	NULL	4	4	4	4

Результатом операции *table1* FULL OUTER JOIN *table2* USING (*c2*) (***полное внешнее соединение по совпадению значений указанных одноименных столбцов***) будет следующая таблица:

JR

a1	*a2*	*table1.c1*	*c2*	*b1*	*b2*	*table2.c1*
1	1	1	1	1	1	1
1	1	2	3	1	2	2
1	1	2	3	3	3	2
1	1	2	3	1	2	2
1	1	2	3	3	3	2
3	NULL	NULL	5	3	NULL	NULL
3	NULL	NULL	5	3	NULL	NULL
2	3	4	NULL	NULL	NULL	NULL
NULL	NULL	NULL	4	4	4	4

Результатом операции *table1* LEFT OUTER JOIN *table2* USING (*c2*, *c1*) (и операции *table1* NATURAL LEFT OUTER JOIN *table2* — ***естественное левое внешнее соединение***) будет следующая таблица:

JR

a1	*a2*	*c1*	*c2*	*b1*	*b2*
1	1	1	1	1	1
1	1	2	3	1	2
1	1	2	3	3	3
1	1	2	3	1	2
1	1	2	3	3	3
2	3	4	NULL	NULL	NULL
3	NULL	NULL	5	NULL	NULL

Результатом операции *table1* RIGHT OUTER JOIN *table2* USING (*c2*, *c1*) (и операции *table1* NATURAL RIGHT OUTER JOIN *table2* — ***естественное правое внешнее соединение***) будет следующая таблица:

JR

a1	*a2*	*c1*	*c2*	*b1*	*b2*
1	1	1	1	1	1
1	1	2	3	1	2
1	1	2	3	3	3
1	1	2	3	1	2
1	1	2	3	3	3
NULL	NULL	4	4	4	4
NULL	NULL	NULL	5	3	NULL
NULL	NULL	NULL	5	3	NULL

Результатом операции *table1* FULL OUTER JOIN *table2* USING (*c2*, *c1*) (и операции *table1* NATURAL FULL OUTER JOIN *table2* – ***естественное полное внешнее соединение***) будет следующая таблица:

JR

a1	*a2*	*c1*	*c2*	*b1*	*b2*
1	1	1	1	1	1
1	1	2	3	1	2
1	1	2	3	3	3
1	1	2	3	1	2
1	1	2	3	3	3
2	3	4	NULL	NULL	NULL
3	NULL	NULL	5	NULL	NULL
NULL	NULL	4	4	4	4
NULL	NULL	NULL	5	3	NULL
NULL	NULL	NULL	5	3	NULL

Наконец, результатом операции *table1* UNION JOIN *table2* (***соединение объединением***) будет следующая таблица:

JR

a1	*a2*	*table1.c1*	*table1.c2*	*b1*	*b2*	*table2.c1*	*table2.c2*
1	1	1	1	NULL	NULL	NULL	NULL
1	1	2	3	NULL	NULL	NULL	NULL
1	1	2	3	NULL	NULL	NULL	NULL
2	3	4	NULL	NULL	NULL	NULL	NULL
3	NULL	NULL	5	NULL	NULL	NULL	NULL
NULL	NULL	NULL	NULL	1	1	1	1
NULL	NULL	NULL	NULL	1	2	2	3
NULL	NULL	NULL	NULL	3	3	2	3
NULL	NULL	NULL	NULL	4	4	4	4
NULL	NULL	NULL	NULL	3	NULL	NULL	5
NULL	NULL	NULL	NULL	3	NULL	NULL	5

Примеры запросов с использованием соединенных таблиц

Мы приведем всего пару примеров, чтобы проиллюстрировать формулировки запросов, в разделе FROM которых используются ссылки на соединенные таблицы, т. е. выражения соединений.

Пример 15.17. Для каждого отдела найти его номер, имя руководителя, число служащих, минимальный, максимальный и средний размеры зарплаты служащих (еще одна формулировка запроса из примера 15.4).

```
SELECT DEPT.DEPT_NO, EMP1.EMP_NAME, COUNT(*), MIN(EMP2.EMP_SAL),
     MAX(EMP2.EMP_SAL), AVG(EMP2.EMP_SAL)
FROM (DEPT NATURAL INNER JOIN EMP AS EMP2)
     INNER JOIN EMP AS EMP1 ON DEPT.DEPT_MNG = EMP1.EMP_NO
GROUP BY DEPT.DEPT_NO, EMP1.EMP_NAME;
```

Пример 15.18. Найти номера служащих и имена их начальников отделов для служащих, размер зарплаты которых больше 30000 руб.

```
SELECT EMP1.EMP_NO, EMP2.EMP_NAME
FROM (EMP AS EMP1 NATURAL INNER JOIN DEPT)
   INNER JOIN EMP AS EMP2 ON DEPT.DEPT_MNG = EMP2.EMP_NO
WHERE EMP1.EMP_SAL > 30000.00;
```

Можно обойтись вообще без раздела WHERE, если пожертвовать «естественностью» первого соединения (**пример 15.17а**):

```
SELECT EMP1.EMP_NO, EMP2.EMP_NAME
FROM (EMP AS EMP1 INNER JOIN DEPT
   ON EMP1.DEPT_NO = DEPT.DEPT_NO AND
      EMP1.EMP_SAL > 30000.00)
   INNER JOIN EMP AS EMP2 ON DEPT.MNG = EMP2.EMP_NO;
```

Возможности соединенных таблиц открывают широкий простор для воображения, но не будем увлекаться и ограничимся приведенными простыми примерами.

Порождаемые таблицы с горизонтальной связью (lateral_derived_table)

Во всех вариантах построения запросов, обсуждавшихся ранее в этой и предыдущей лекциях, оставалась действующей общая семантика выполнения запроса: на первом шаге вычисляется расширенное декартово произведение таблиц, специфицированных в списке раздела FROM. Это остается верным и для случаев порождаемых и соединенных таблиц – вычисление выражения запросов или выражения соединений соответственно производится как подшаг вычисления раздела FROM. Однако в SQL имеется один специальный случай спецификации ссылки на таблицу (table_reference), который, вообще говоря, изменяет семантику раздела FROM. В этом подразделе мы кратко рассмотрим этот специальный случай.

Как показывают синтаксические правила, приведенные в лекции 13, один из возможных способов спецификации ссылки на таблицу состоит в следующем:

```
table_reference ::= LATERAL (query_expression)
                      [ [ AS ] correlation_name
                        [ ( derived_column_list ) ] ]
```

Таблица, ссылка на которую специфицируется таким образом, называется *порождаемой таблицей с горизонтальной связью** (lateral_derived_table; для краткости будем называть такие таблицы LD-таблицами). Отличие LD-таблицы от обычной порождаемой таблицы состоит в том, что в выражении запросов LD-таблицы разрешается использовать ссылки на столбцы таблиц, специфицированных ранее в разделе FROM (т. е. таких таблиц, ссылки на которые содержатся в списке раздела FROM слева от ссылки на данную LD-таблицу).** Покажем на примере, каким образом наличие в списке раздела FROM ссылки на LD-таблицу меняет семантику этого раздела.

---

* Конечно, предлагаемый русский вариант термина *lateral* слишком громоздок. По всей видимости, если этот механизм войдет в практику пользователей SQL, нужно будет использовать качестве термина что-то вроде *латеральной порождаемой таблицы*. Но здесь для нас главным является не предложение хорошей новой терминологии, а обеспечение понимания материала.

** Тем самым, ссылка на LD-таблицу не может быть первой в списке раздела FROM. Кстати, может возникнуть естественный вопрос: почему разрешаются ссылки только на таблицы, находящиеся в списке раздела FROM только слева LD-таблицы? Стандарт отвечает на этот вопрос весьма просто и бесхитростно. Если разрешить использовать ссылки, находящиеся и слева, и справа от спецификации ссылки на LD-таблицу, то это может привести к зацикливанию при выполнении раздела FROM. Поэтому нужно было выбирать одно из направлений, и было выбрано направление слева направо.

Предположим, что раздел FROM имеет вид FROM *T1*, *T2*, причем таблица *T2* является LD-таблицей. Обозначим соответствующее выражение запросов через *Q2*. Тогда таблица *T*, являющаяся результатом раздела FROM, будет вычисляться следующим образом. Последовательно, строка за строкой просматривается таблица *T1*. Пусть *s1* является очередной строкой *T1*. Тогда в *Q2* все ссылки на столбцы вида *T1.ck*, где *ck* – имя некоторого столбца *T1*, заменяются значением *s1.ck*, и вычисляется полученное таким образом выражение запросов. Обозначим результирующую таблицу этого выражения через $T2_{s1}$. Обозначим через $T12_{s1}$ таблицу, являющуюся результатом расширенного декартова произведения *s1* CROSS JOIN $T2_{s1}$. Таблица *T* получается путем объединения с сохранением дубликатов таблиц $T12_{s1}$, полученных для всех строк *s1* таблицы *T1*.

Видимо, наиболее важным (хотя и не единственным) частным случаем применения LD-таблицы является тот случай, когда в результате выполнения раздела FROM формируется соединение таблиц. Многие из формулировок запросов, приводившихся в этой лекции в качестве примеров, можно переформулировать с использованием данного механизма. Приведем лишь один простой пример.

Пример 15.19. Найти номера служащих, не являющихся руководителями отделов и получающих заплату, размер которой равен размеру зарплаты какого-либо руководителя отдела (еще одна формулировка запроса из примера 14.10 из лекции 14).

```
SELECT EMP.EMP_NO
FROM DEPT, LATERAL
          (SELECT EMP1_SAL
             FROM EMP EMP1
             WHERE EMP1.EMP_NO = DEPT.DEPT_MNG),
          LATERAL
          (SELECT EMP_NO
             FROM EMP
             WHERE EMP_SAL = EMP1_SAL AND
             EMP.EMP_NO <> DEPT.DEPT_MNG);
```

Я не могу привести ни одного примера запроса, который было бы невозможно сформулировать без использования порождаемых таблиц с горизонтальной связью. Возникает впечатление (возможно, ошибочное), что эта конструкция была введена в язык по двум причинам – (a) из соображений общности и (b) по причине простоты реализации (в том смысле, что для реализации LD-таблиц не требуется изобретать какие-то новые технические приемы).

Заключение

Думаю, что теперь читатели в состоянии в полной мере оценить мощность, разнообразие и избыточность средств языка SQL, предназначенных для формулировки запросов на выборку данных. Конечно, язык SQL (по крайней мере, ту часть SQL, которая обсуждается в этом курсе) нельзя считать языком программирования, но написание сложных запросов сродни программированию. И нельзя сказать, что SQL каким-либо образом дисциплинирует это «программирование». По всей видимости, в общем случае никто не может сказать, какая из формулировок одного и того же запроса является более правильной, это дело вкуса.

Зачастую десять студентов, одновременно формулирующих на SQL один и тот же запрос к одной и той же базе данных, выдают десять разных правильных решений. Один человек предпочитает формулировки запросов в классическом стиле, другой использует выражения запросов в разделе `FROM`, третий пытается сосредоточить все условия выборки в разделе `HAVING`. Люди с алгебраическими наклонностями предпочитают использовать выражения соединений. Приходилось встречать и формулировки со сложными вложенными подзапросами в списке выборки раздела `SELECT`.

Конечно, теоретически компилятор SQL должен быть в состоянии распознать все эквивалентные формулировки одного и того же запроса и выработать для всех них один и тот же наиболее эффективный план выполнения. Но чем больше разнообразие возможных формулировок, тем сложнее эта задача. Отсюда практический совет: не злоупотребляйте сложностью формулировки запроса. Полагайтесь на интуицию (и имеющиеся представления об особенностях используемой системы) и формулируйте запрос как можно проще.

И еще один практический совет. При формулировке запроса никогда не пользуйтесь имеющимися у вас данными о текущем состоянии базы данных, полагайтесь только на метаданные схемы базы данных. В противном случае вы сможете сформулировать запрос, выдающий в данный момент правильный результат, но этот запрос не будет эквивалентен никакому запросу, выдающему правильный ответ при любом состоянии базы данных.

Лекция 16. Язык баз данных SQL: средства формулировки аналитических и рекурсивных запросов

В этой лекции мы завершаем обсуждение средств выборки данных языка SQL коротким описанием сравнительно недавно появившихся в языке SQL средств формулировки аналитических и рекурсивных запросов.

Ключевые слова: аналитические запросы к базе данных, оперативная аналитическая обработка баз данных, OLAP, раздел GROUP BY ROLLUP, агрегатная функция GROUPING, раздел GROUP BY CUBE, рекурсивные запросы, обход дерева в ширину, обход дерева в глубину, цикл в ориентированном графе, прямая рекурсия, линейная рекурсия, монотонная прогрессия, взаимная рекурсия, отрицание, начальный источник рекурсии, стратификация, семантика фиксированной точки, рекурсивные запросы с разделом WITH, конструкция SEARCH, раздел CYRCLE, рекурсивное представление.

Введение

Две темы, которым посвящается эта лекция, касаются сравнительно новых возможностей оператора SELECT языка SQL, впервые появившихся в стандарте SQL:1999 и открывающих возможность использования языка в приложениях, для которых ранее он не был приспособлен. Речь идет о возможностях аналитических и рекурсивных запросов. Эти темы логически не связаны, их объединяет лишь то, что соответствующие средства очень громоздки и не всегда легко понимаются. В данной краткой лекции мы не стремимся привести полное описание возможностей, специфицированных в стандарте SQL. Наша цель состоит лишь в том, чтобы в общих чертах описать подход SQL в указанных направлениях.

В аналитических приложениях обычно требуются не детальные данные, непосредственно хранящиеся в базе данных, а некоторые их обобщения, агрегаты. Например, аналитика интересует не заработная плата конкретного человека в конкретное время, а изменение заработной платы некоторой категории людей в течение определенного промежутка времени. Если пользоваться терминологией SQL, то типичный запрос к базе данных со стороны аналитического приложения содержит раздел GROUP BY и вызовы агрегатных функций. Хотя в этом курсе мы почти не касаемся вопросов реализации SQL-ориентированных СУБД, из общих соображений должно быть понятно, что запросы с разделом GROUP BY в общем случае являются «трудными» для СУБД, поскольку для группирования таблицы, вообще говоря, требуется внешняя сортировка.

В системах баз данных, специально спроектированных в расчете на аналитические приложения, проблему обычно решают за счет явного избыточного хранения агрегированных данных (т. е. результатов вызовов агрегатных функций). Конечно, для этого требуется динамическая корректировка хранимых агрегатных значений при изменении детальных данных, но для таких специализированных баз данных это не слишком обременительно, поскольку аналитические базы данных обновляются сравнительно редко.

Однако далеко не каждое предприятие может позволить себе одновременно поддерживать оперативную базу данных для работы обычных приложений оперативной обработки транзакций (OLTP), таких, как бухгалтерские, кадровые и другие приложения, и аналитическую базу данных для приложений оперативной аналитической обработки (OLAP). Приходится выполнять аналитические приложения над детальными оперативными базами данных, и эти приложения обращаются к СУБД с многочисленными трудоемкими запросами с разделами `GROUP BY` и вызовами агрегатных функций.

Разработчики стандарта языка SQL старались одновременно решить две задачи: сократить число запросов, требуемых в аналитических приложениях, и добиться снижения стоимости запросов с разделом `GROUP BY`, обеспечивающих требуемые суммарные данные. В этой лекции мы обсудим наиболее важные, с нашей точки зрения, конструкции языка SQL, облегчающие формулировку, выполнение и использование результатов аналитических запросов: разделы `GROUP BY ROLLUP` и `GROUP BY CUBE` и новую агрегатную функцию `GROUPING`, позволяющую правильно трактовать результаты аналитических запросов при наличии неопределенных значений.

Традиционно язык SQL никогда не обладал возможностью формулировки рекурсивных запросов, где под рекурсивным запросом (упрощенно говоря) мы понимаем запрос к таблице, которая сама каким-либо образом изменяется при выполнении этого запроса. Напомню, что это заложено в базовую семантику оператора SQL: до выполнения раздела `WHERE` результат раздела `FROM` должен быть полностью вычислен.

Однако разработчикам приложений часто приходится решать задачи, для которых недостаточно традиционных средств формулировки запросов языка SQL: например, нахождение маршрута движения между двумя заданными географическими точками, определения общего набора комплектующих для сбора некоторого агрегата и т. д. Компании-производители SQL-ориентированных СУБД пытались удовлетворять такие потребности за счет частных решений, обладающих ограниченными рекурсивными свойствами, но до появления стандарта SQL:1999 общие стандартизованные средства отсутствовали.

Следует отметить и некоторое давление на SQL-сообщество со стороны сообщества логических систем баз данных. На основе языка логического программирования Prolog был разработан язык реляционных баз данных Datalog, обеспечивающий все необходимые средства для обычной работы с базами данных наряду с развитыми возможностями рекурсивных запросов. Требовался адекватный ответ со стороны разработчиков стандарта SQL.

Компромиссное (не слишком красивое) решение для введения рекурсии в SQL было найдено на основе введения раздела `WITH` в выражение запроса. Только в этом разделе допускается как линейная, так и взаимная рекурсия между вводимыми порождаемыми таблицами. При этом только для линейной рекурсии обеспечиваются дополнительные возможности управления порядком вычисления рекурсивно определенной порождаемой таблицы и контроля отсутствия циклов. Следует заметить, что при чтении стандарта временами возникает впечатление, что его авторы сами не до конца еще осознали всех возможных последствий, к которым может привести использование введенных конструкций. Я думаю, что в следующих версиях стандарта следует ожидать уточнений и/или ограничений использования названных конструкций. В связи с этим в данной лекции мы ограничиваемся общими определениями рекурсивных конструкций языка SQL и обсуждением простого случая рекурсивного запроса.

Возможности формулирования аналитических запросов

Аналитическими запросами к базе данных принято называть запросы, сводные (агрегатные) результаты которых вычисляются над детальными данными, хранящимися в таблицах базы данных. В этом смысле любой запрос на языке SQL, результат которого основан на вычислении агрегатных функций, можно назвать аналитическим. Характерная особенность аналитических запросов состоит в том, что, как правило, они применяются к большим по объему базам данных, и выполнение таких запросов вызывает существенные накладные расходы СУБД.

В этом курсе мы не будем подробно обсуждать возможности языка SQL, предназначенные для поддержки оперативной аналитической обработки баз данных (`OLAP` – on-line analytical processing). Рассмотрим только самые основные средства, опираясь на простые примеры. Для этих примеров предположим, что таблица `EMP` содержит следующий набор строк (покажем содержимое только тех столбцов, которые потребуются в примерах, причем для простоты будем считать, что в столбце `EMP_DATE` содержится не полная дата, а только год рождения служащего):

EMP

EMP_NO	DEPT_NO	EMP_BDATE	EMP_SAL
2440	1	1950	15000.00
2441	1	1950	16000.00
2442	1	1960	14000.00
2443	1	1960	19000.00
2444	2	1950	17000.00
2445	2	1950	16000.00
2446	2	1960	14000.00
2447	2	1960	20000.00
2448	3	1950	18000.00
2449	3	1950	13000.00
2450	3	1960	21000.00
2451	3	1960	22000.00

Представим себе, что для проведения анализа требуется узнать максимальный размер зарплаты на всем предприятии, максимальный размер зарплаты в каждом отделе и максимальный размер зарплаты сотрудников каждой возрастной категории каждого отдела. Если пользоваться стандартными средствами языка SQL, обсуждавшимися ранее в предложенном курсе, то для получения этих данных потребуется три запроса:

```
SELECT MAX (EMP_SAL) AS MAX_ENT_SAL
FROM EMP;
SELECT DEPT_NO, MAX (EMP_SAL) AS MAX_DEP_SAL
FROM EMP
GROUP BY DEPT_NO;
SELECT DEPT_NO, EMP_BDATE, MAX (EMP_SAL) AS MAX_DEP_BDATE_SAL
FROM EMP
GROUP BY DEPT_NO, EMP_BDATE;
```

При выполнении запросов будут получены следующие результирующие таблицы:

MAX_ENT_SAL
22000.00

DEPT_NO	MAX_DEP_SAL
1	19000.00
2	20000.00
3	22000.00

DEPT_NO	EMP_BDATE	MAX_DEP_BDATE_SAL
1	1950	16000.00
1	1960	19000.00
2	1950	17000.00
2	1960	20000.00
3	1950	18000.00
3	1960	22000.00

Раздел GROUP BY ROLLUP

Эти же результаты можно получить при выполнении единственного запроса, если в его формулировке использовать специальный вид группировки ROLLUP **(пример 16.1)**:

```
SELECT DEPT_NO, EMP_BDATE, MAX (EMP_SAL) AS MAX_SAL
FROM EMP
GROUP BY ROLLUP (DEPT_NO, EMP_BDATE);
```

Сначала покажем, как будет выглядеть результирующая таблица этого запроса, а потом приведем развернутое пояснение действия новой конструкции. В результате выполнения запроса будет получена таблица, показанная на рис. 16.1.

Как видно, в столбце MAX_SAL первой строки* результирующей таблицы находится максимальное значение зарплаты служащих на всем предприятии. Столбцы DEPT_NO и EMP_BDATE в этой строке содержат неопределенное значение, поскольку значение MAX_SAL не привязано к каким-либо отделу и возрастной категории. В столбце MAX_SAL следующих трех строк находятся максимальные значения зарплаты служащих отделов с номерами 1, 2 и 3 соответственно, что показывают значения

* Конечно, мы показали строки результирующей таблицы, расположенные в удобном для нас порядке только для упрощения объяснений. В действительности, строки результирующей таблицы (как обычно) будут расположены в порядке, определяемом системой. Чтобы добиться в точности такого порядка расположения строк, как это показано на рис. 16.1, к формулировке запроса из примера 16.1 нужно добавить раздел ORDER BY DEPT_NO, EMP_BDATE.

столбца DEPT_NO. Столбец EMP_BDATE в этих строках содержит неопределенное значение, поскольку значение MAX_SAL не привязано к какой-либо возрастной категории. Наконец, в столбце MAX_SAL в последних шести строках содержатся максимальные значения зарплаты служащих каждой возрастной категории каждого отдела, что показывают значения столбцов DEPT_NO и EMP_BDATE, которые теперь содержат соответствующий номер отдела и год рождения служащих.

DEPT_NO	EMP_BDATE	MAX_SAL
NULL	NULL	22000.00
1	NULL	19000.00
2	NULL	20000.00
3	NULL	22000.00
1	1950	16000.00
1	1960	19000.00
2	1950	17000.00
2	1960	20000.00
3	1950	18000.00
3	1960	22000.00

Рис. 16.1. Результат запроса с разделом GROUP BY ROLLUP

В общем случае пусть раздел группировки запроса имеет вид GROUP BY ROLLUP ($cname_1$, $cname_2$,..., $cname_n$), где $cname_i$ ($i = 1, 2,..., n$) — имя столбца таблицы-результата раздела FROM запроса. Пусть в списке выборки используются вызовы агрегатных функций AGG_1, AGG_2,..., AGG_m над значениями столбцов, не входящих в список группировки, а также имена столбцов $cname_1$, $cname_2$,..., $cname_n$. Тогда запрос выполняется следующим образом. Первая строка результата (первый набор строк результирующей таблицы) производится таким образом, как если бы в запросе вообще отсутствовал раздел GROUP BY, т. е. агрегатные функции AGG_1, AGG_2,..., AGG_m вычисляются над значениями всех строк таблицы. Значением столбцов $cname_1$, $cname_2$,..., $cname_n$ в этой строке является NULL. ($i+1$)-й набор строк результата формируется так, как если бы раздел группировки запроса имел вид GROUP BY ($cname_1$, $cname_2$,..., $cname_i$) ($1 \leqslant i < n$). Во всех этих строках значением столбцов $cname_{(i+1)}$,..., $cname_n$ является NULL. Наконец, ($n+1$)-й набор строк результата формируется так, как если бы раздел группировки запроса имел вид GROUP BY ($cname_1$, $cname_2$,..., $cname_n$).

Может показаться, что запросы, содержащие раздел GROUP E
настолько сложны, что их выполнение будет занимать чрезме
шое время. Это ощущение является ложным. В действительн
выполнении запросов с обычной группировкой вида GROUP B
$cname_2$,..., $cname_n$, как правило, последовательно выполняется с
строк таблицы-результата раздела FROM в соответствии со зн
столбца $cname_1$, затем – в соответствии со значениями столбца cn
и в заключение – сортировка в соответствии со значениями столб
Во время выполнения каждой сортировки можно заодно вычи
чения агрегатных функций. Так что стоимость выполнения за
держащего раздел GROUP BY ROLLUP, лишь незначительно отл
стоимости выполнения запроса с обычной группировкой.

Агрегатная функция *GROUPING*

Обсудим теперь один более тонкий вопрос. Как говорилос
12, определение столбцов DEPT_NO и EMP_BDATE таблицы EMP
появление в этих столбцах неопределенных значений. Поэтому
лицы EMP могло бы иметь, например, следующий вид:

EMP

EMP_NO	DEPT_NO	EMP_BDATE	EMP_SAL
2440	1	1950	15000.00
2441	1	1950	16000.00
2442	1	1960	14000.00
2443	1	1960	19000.00
2452	1	NULL	15000.00
2453	1	NULL	17000.00
2444	2	1950	17000.00
2445	2	1950	16000.00
2446	2	1960	14000.00
2447	2	1960	20000.00
2448	3	1950	18000.00
2449	3	1950	13000.00
2450	3	1960	21000.00
2451	3	1960	22000.00
2454	NULL	1950	13000.00
2455	NULL	1950	14000.00
2456	NULL	NULL	19000.00

Тогда результат запроса из примера 16.1 имел бы следующий вид*:

DEPT_NO	EMP_BDATE	MAX_SAL
NULL	NULL	22000.00
NULL	NULL	19000.00
NULL	NULL	14000.00
1	NULL	19000.00
2	NULL	20000.00
3	NULL	22000.00
1	NULL	17000.00
1	1950	16000.00
1	1960	19000.00
2	1950	17000.00
2	1960	20000.00
3	1950	18000.00
3	1960	22000.00
NULL	1950	14000.00

Рис. 16.2. Результат запроса с разделом GROUP BY ROLLUP к таблице с неопределенными значениями столбцов группировки

Очевидно, что, просматривая строки таблицы, показанной на рис. 16.2, невозможно установить, в какой из первых трех строк неопределенное значение столбцов DEPT_NO и EMP_BDATE означает то, что эта строка является сводной для всего предприятия, а не то, что она является сводной для всех служащих с неизвестными номером отдела и годом рождения или просто для всех служащих с неизвестным номером отдела. Аналогичным образом невозможно понять, какая строка в следующей далее паре строк является сводной для всех служащих отдела номер 1, а не для всех служащих отдела номер 1 с неизвестным годом рождения.

Для того чтобы всегда можно было разобраться в результатах запросов, включающих раздел GROUP BY ROLLUP, в язык SQL была введена специальная агрегатная функция GROUPING. Эта функция применяется к столбцу, входящему в список столбцов раздела GROUP BY ROLLUP, и принимает целое значение 1 в тех строках результирующей таблицы, в которых соответствующий столбец имеет значение NULL по той причине, что стро-

* Мы опять искусственным образом упорядочили результат запроса для удобства пояснений.

ка является сводной для более обобщенной группы. В противном случае функция GROUPING принимает значение 0.

Уточним формулировку запроса из примера 16.1 (**пример 16.1а**):

```
SELECT DEPT_NO, EMP_BDATE, MAX (EMP_SAL) AS MAX_SAL,
   GROUPING (DEPT_NO) AS GDN, GROUPING (EMP_BDATE) AS GEB
FROM EMP
GROUP BY ROLLUP (DEPT_NO, EMP_BDATE);
```

Результирующая таблица для этого запроса будет иметь следующий вид:

DEPT_NO	EMP_BDATE	MAX_SAL	GDN	GEB
NULL	NULL	22000.00	1	1
NULL	NULL	19000.00	0	0
NULL	NULL	14000.00	1	0
1	NULL	19000.00	0	1
2	NULL	20000.00	0	1
3	NULL	22000.00	0	1
1	NULL	17000.00	0	0
1	1950	16000.00	0	0
1	1960	19000.00	0	0
2	1950	17000.00	0	0
2	1960	20000.00	0	0
3	1950	18000.00	0	0
3	1960	22000.00	0	0
NULL	1950	14000.00	0	0

Рис. 16.3. Результат запроса с разделом GROUP BY ROLLUP и вызовами агрегатной функции GROUPING к таблице с неопределенными значениями столбцов группировки

Анализируя значения столбцов GDN и GEB в строках таблицы, показанной на рис. 16.4, можно убедиться, что значение столбца MAX_SAL в первой строке является максимальным значением зарплаты всех служащих предприятия, во второй строке – максимальным значением зарплаты служащих с неизвестными номером отдела и годом рождения, а в третьей строке – максимальным значением зарплаты всех служащих с неизвест-

ным номером отдела. В следующих трех строках значения столбца MAX_SAL являются максимальными значениями зарплаты служащих с неизвестным годом рождения из отделов с номерами 1, 2 и 3 соответственно. Как видно, значения столбцов GDN и GEB являются своего рода индикаторами, указывающими на природу основных значений строки.

Раздел *GROUP BY CUBE*

Наконец, заметим, что, в отличие от запросов с традиционной группировкой, результат запроса, содержащего раздел GROUP BY ROLLUP, зависит от порядка столбцов в списке группировки. При выполнении запроса происходит движение по этому списку слева направо с повышением уровня детальности результирующих данных. Существует еще одна разновидность запроса с группировкой, основанная на использовании раздела GROUP BY CUBE.

Пусть раздел группировки запроса имеет вид GROUP BY CUBE ($cname_1$, $cname_2$,..., $cname_n$), где $cname_i$ (i = 1, 2,..., n) – имя столбца таблицы-результата раздела FROM запроса. Обозначим через S_{GBC} множество {$cname_1$, $cname_2$,..., $cname_n$}. Пусть S_1 является произвольным подмножеством S_{GBC}, т. е. S_1 представляет собой пустое множество или имеет вид {$cname_{i1}$, $cname_{i2}$,..., $cname_{im}$}, где $m \leqslant n$, и каждое имя столбца $cnamei_j$ совпадает с одним и только одним именем столбца из списка столбцов раздела GROUP BY CUBE. Очевидно, что у множества S_{GBC} существует 2^n подмножеств различных вида S_1. Тогда по определению результат этого запроса совпадает с объединением результатов 2^n запросов с теми же разделами SELECT, FROM и WHERE, что и у запроса с GROUP BY CUBE, и с разделом группировки вида GROUP BY S_1, причем во всех строках результата частичного запроса значением любого столбца $cname_j$ такого, что $cname_j \in S_{GBC}$ и $cname_j \notin S_1$, является NULL. Запрос с разделом группировки вида GROUP BY S, где S – пустое множество, трактуется как запрос без раздела GROUP BY. Вот пример запроса, содержащего раздел GROUP BY CUBE.

Пример 16.2. Найти максимальный размер зарплаты во всем предприятии, максимальный размер зарплаты в каждом отделе, максимальный размер зарплаты служащих в каждой возрастной категории и максимальный размер зарплаты служащих каждой возрастной категории каждого отдела.

```
SELECT DEPT_NO, EMP_BDATE, MAX (EMP_SAL)AS MAX_SAL,
   GROUPING (DEPT_NO) AS GDN, GROUPING (EMP_BDATE) AS GEB
FROM EMP
GROUP BY CUBE (DEPT_NO, EMP_BDATE);
```

Результирующая таблица для этого запроса будет иметь следующий вид:

DEPT_NO	EMP_BDATE	MAX_SAL	GDN	GEB
NULL	NULL	22000.00	1	1
NULL	NULL	19000.00	0	0
NULL	NULL	14000.00	1	0
1	NULL	19000.00	0	1
2	NULL	20000.00	0	1
3	NULL	22000.00	0	1
1	NULL	17000.00	0	0
1	1950	16000.00	0	0
1	1960	19000.00	0	0
2	1950	17000.00	0	0
2	1960	20000.00	0	0
3	1950	18000.00	0	0
3	1960	22000.00	0	0
NULL	1950	14000.00	0	0
NULL	1950	18000.00	1	0
NULL	1960	22000.00	1	0

Рис. 16.4. Результат запроса с разделом `GROUP BY CUBE` и вызовами агрегатной функции `GROUPING` к таблице с неопределенными значениями столбцов группировки

Как видно, результат запроса из примера 16.2 совсем немного отличается от результата запроса из примера 16.1а. Добавились две последние строки, показывающие максимальные значения зарплаты всех служащих предприятия, родившихся в 1950-м и 1960-м гг. соответственно.

Наш пример может навести на мысль, что и в общем случае запросы, содержащие раздел `GROUP BY CUBE`, не слишком отличаются от запросов с `GROUP BY ROLLUP`, и выполнение этих запросов тоже не слишком различается. Однако это совсем не так. Запрос, содержащий раздел `GROUP BY CUBE`, действительно вырождается в объединение результатов 2^n запросов с обычным разделом `GROUP BY`. Соответственно, сложность выполнения такого запроса несравненно выше сложности выполнения похожего запроса с `GROUP BY ROLLUP`. В нашем примере все получилось так просто только по той причине, что в запросе имеются всего два столбца группировки.

Рекурсивные запросы

Начнем этот раздел с нескольких определений, касающихся понятий, которые связаны с рекурсией. Эти понятия имеют общий характер, но в приведенных ниже определениях и комментариях к ним (там, где это уместно) подчеркивается контекст SQL.

Определения, относящиеся к рекурсии

Обход дерева в ширину. При этом способе обхода непосредственные потомки обходятся слева направо, до того как производится переход к потомкам следующего уровня родства.

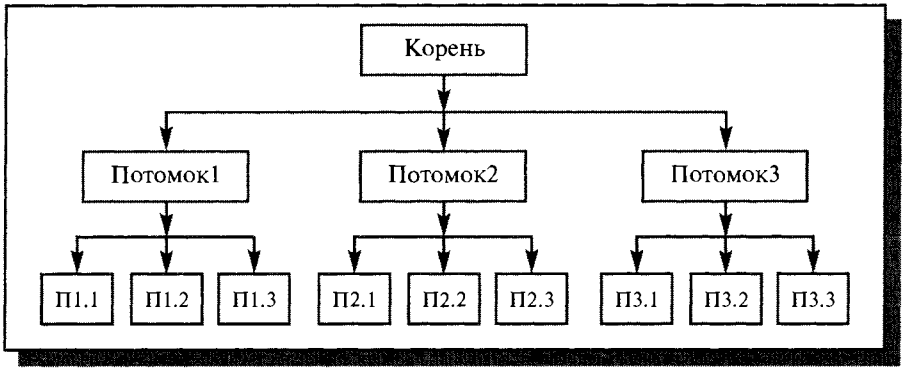


Рис. 16.5. Пример дерева

При обходе в ширину дерева, показанного на рис. 16.5, узлы будут обходиться в следующем порядке: `Корень-Потомок1-Потомок2-Потомок3-П1.1-П1.2-П1.3-П2.1-П2.2-П2.3-П3.1-П3.2-П3.3`.

Обход дерева в глубину. При этом способе обхода на каждом шаге производится переход к самому левому текущему потомку. При обходе в глубину дерева с рис. 16.5 порядок обхода узлов будет следующим: `Корень-Потомок1-П1.1-П1.2-П1.3-Потомок2-П2.1-П2.2-П2.3-Потомок3-П3.1-П3.2-П3.3`.

Цикл в ориентированном графе. В теории графов ориентированный граф называется циклическим в том и только в том случае, когда хотя бы один узел графа одновременно является и предком, и потомком (т. е. для этого узла имеется и выходящая, и входящая дуги). В SQL:1999 узлами графа рекурсии являются строки, входящие в результат рекурсивного запроса, а дуги соответствуют способам обработки текущих строк, которые ведут к добавлению к результату новых строк. На рис. 16.6 показан простейший пример ориентированного графа с циклом.

Рис. 16.6. Пример графа с циклом

Прямая рекурсия. По определению, некоторый элемент использует прямую рекурсию в том и только в том случае, когда он обращается сам к себе без посредников. Пример, приведенный на рис. 16.6, демонстрирует (в графовой форме) прямую рекурсию. На рис. 16.7 показан графовый пример непрямой рекурсии.

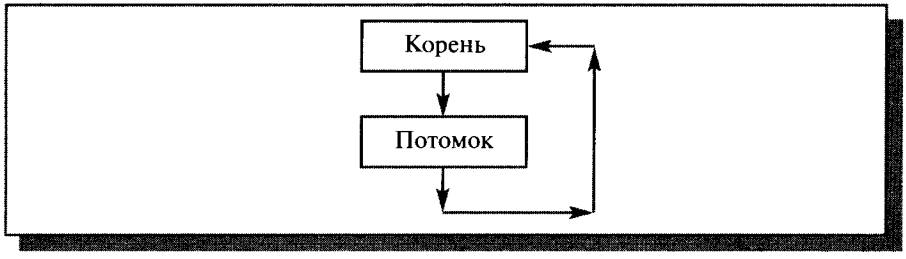


Рис. 16.7. Графовый пример непрямой рекурсии

Линейная рекурсия. При линейно рекурсивном вызове элемент прямо рекурсивно обращается сам к себе не более одного раза. В SQL:1999 в определении любой виртуальной таблицы с рекурсией допускается не более одной ссылки на саму себя (в разделе `FROM` и/или в подзапросах). На рис. 16.8 показан графовый пример рекурсии, не являющейся линейной.

Монотонность. Монотонной прогрессией называется последовательность неубывающих или невозрастающих значений. Например, последовательность натуральных чисел `{1, 2, ..., n, ...}` является монотонной. В SQL:1999 свойство монотонности поддерживается в том смысле, что число строк результата рекурсивного запроса не уменьшается на каждом шаге рекурсии.

Взаимная рекурсия. Элементы *A* и *B* связаны отношением взаимной рекурсии, если *A* прямо или косвенно вызывает *B*, и *B* прямо или косвенно вызывает *A*. На рис. 16.9 показан графовый пример взаимной рекурсии (элемент *A* вызывает элемент *B* через элемент *C*, а элемент *B* вызывает элемент *A* через элемент *D*).

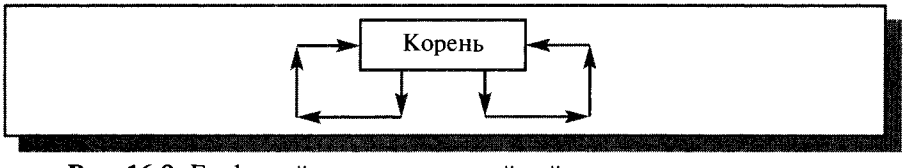


Рис. 16.8. Графовый пример нелинейной рекурсии

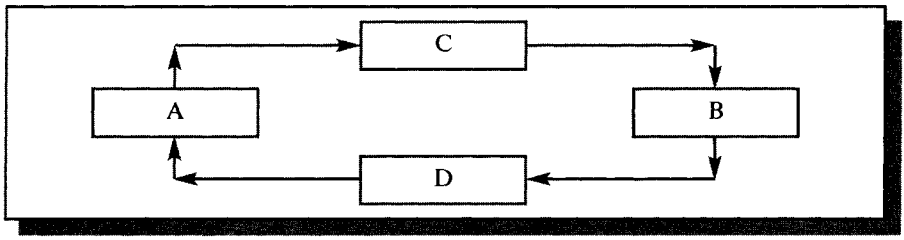


Рис. 16.9. Графовый пример взаимной рекурсии

Отрицание. В контексте SQL отрицанием называется любое действие, приводящее к уменьшению числа строк в результате запроса. Свойствами отрицания обладают операции над (мульти)множествами `EXCEPT` и `INTERSECT`, спецификация `DISTINCT`, условие `NOT EXISTS` и т. д. В стандарте SQL не запрещается использование отрицания в рекурсивных запросах. Возможной проблемы нарушения монотонности удается избежать за счет того, что отрицание разрешается применять только к тем таблицам, которые являются полностью известными (или вычисленными) к моменту применения отрицания. В процессе вычисления таблицы применение к ней отрицания не допускается.

Начальный источник рекурсии. При выполнении рекурсивных вычислений обычно (хотя и не всегда) имеется некоторое начальное значение. В SQL этим начальным источником рекурсии является одна или несколько строк, удовлетворяющих некоторым начальным условиям. На основе этих строк в процессе рекурсивного вычисления производятся дополнительные строки, образующие окончательный результат.

Стратификация. В SQL рекурсивный запрос обычно состоит из «рекурсивной» и «нерекурсивной» частей. В процессе стратификации («расслоения») запроса выполнение этих двух частей разделяется. В более сложных рекурсивных запросах может содержаться несколько рекурсивных частей и более одной нерекурсивной части. В этом случае в процессе стратификации будет обнаружено большее число слоев.

Семантика фиксированной точки. В контексте SQL:1999 семантика фиксированной точки означает, что решение о завершении рекурсивного запроса принимается тогда, когда становится невозможно добавить к результату какие-либо дополнительные строки.

Рекурсивные запросы с разделом *WITH*

В предыдущих лекциях мы уже говорили о разновидности спецификации ссылки на таблицу с использованием раздела `WITH`. Однако мы умышленно отложили обсуждение рекурсивных возможностей. Полный синтаксис раздела `WITH` выглядит следующим образом:

```
with_clause ::= WITH [ RECURSIVE ]
with_element_comma_list
with_element ::= query_name [ (column_name_list) ]
     AS ( query_expression ) [ search_or_cycle_clause ]
search_or_cycle_clause ::= search_clause
                         | cycle_clause
                         | search_clause cycle_clause
search_clause ::= SEARCH recursive_search_order SET
                    sequence_column_name
recursive_search_order ::= DEPTH FIRST BY
                  order_item_commalist
                  | BREADTH FIRST BY order_item_commalist
cycle_clause ::= CYCLE cycle_column_name_comma_list
          SET cycle_mark_column_name TO value_expression
          DEFAULT value_expression
          USING path_column_name
```

Для иллюстрации возможностей рекурсивных запросов с разделом WITH и пояснения смысла конструкций SEARCH и CYCLE воспользуемся классическим примером «разборки деталей» (в данном случае мы будем разбирать автомобиль). Предположим, что данные о конструктивных элементах автомобиля хранятся в таблице CAR, определенной следующим образом:

```
CREATE TABLE CAR (CONTAINING_PART VARCHAR (10),
                  CONTAINED_PART VARCHAR (10),
                  NUMBER_OF_PARTS INTEGER,
                  PART_COST DECIMAL (6,2));
```

У автомобиля имеется один конструктивный элемент верхнего уровня – полностью собранный автомобиль. Этот элемент не является составной частью какого-либо другого элемента, и для его строки значением столбца CONTAINING_PART является текстовая строка длины 0. В любой другой строке таблицы CAR, соответствующей некоторому неатомарному конструктивному элементу *e*, столбец CONTAINING_PART содержит идентификационный номер элемента *e1*, в который входит элемент *e*, столбец NUMBER_OF_PARTS – число экземпляров элемента *e*, входящих в *e1*, а столбец CONTAINED_PART – идентификационный номер самого элемента *e*. В любой строке таблицы CAR, соответствующей некоторому атомарному конструктивному элементу, значением столбца CONTAINED_PART является строка длины 0, а в столбце PART_COST сохраняется цена атомарного конструктивного элемента (для неатомарных элементов значение этого столбца равно нулю).

Предположим, что нам требуется разобрать автомобиль, начиная с элемента самого верхнего уровня, и для каждого конструктивного элемента

получить его номер, общее число используемых экземпляров этого элемента, а также, если элемент является атомарным, общую стоимость используемых экземпляров. Вот возможная формулировка запроса (**пример 16.3**):

```
WITH RECURSIVE PARTS (PART_NUMBER, NUMBER_OF_PARTS, COST) AS
   (SELECT CONTAINED_PART, 1, 0.00                          (a)
   FROM CAR
   WHERE CONTAINING_PART = ''
 UNION ALL
   SELECT CAR.CONTAINED_PART, CAR.NUMBER_OF_PARTS,
      CAR.NUMBER_OF_PARTS * CAR.PART_COST
   FROM CAR, PARTS
   WHERE PARTS.PART_NUMBER = CAR.CONTAINING_PART)
SELECT PART_NUMBER, SUM(NUMBER_OF PARTS), SUM(COST)         (b)
FROM PARTS
GROUP BY PART_NUMBER;
```

Этот запрос будет выполняться следующим образом. При вычислении раздела FROM основного запроса (b) начнется выполнение рекурсивного выражения запросов (a), определенного в разделе WITH. На первом шаге рекурсии будет выполнена часть данного выражения, предшествующая операции UNION ALL и образующая начальный источник рекурсии. В результате будет произведено исходное состояние виртуальной таблицы PARTS, в котором, в нашем случае, появится единственная строка, соответствующая автомобилю целиком. На следующем шаге к таблице PARTS будут добавлены строки, соответствующие конструктивным элементам второго уровня (для автомобиля это, по-видимому, двигатель, колеса, шасси и т. д.). Этот процесс будет продолжаться до тех пор, пока мы не дойдем до атомарных конструктивных элементов и не достигнем, тем самым, фиксированной точки. Поскольку в рекурсивном запросе содержится операция UNION ALL, в результирующей таблице могут появляться строки-дубликаты. Наличие строки-дубликата вида <*part_no*, *number*, *cost*> означает, что элемент с номером *part_no* входит в одном и том же числе экземпляров в несколько конструктивных элементов более высокого уровня.

Раздел SEARCH

В приведенном выше примере не определялся порядок, в котором строки добавляются к частичному результату рекурсивного запроса. Однако иногда требуется, чтобы иерархия обходилась в глубину или в ширину. Соответствующая возможность обеспечивается конструкцией SEARCH. При указании требования обхода в глубину гарантируется, что каждый элемент-

предок появится в результате раньше своих потомков и своих братьев справа. Если указывается требование обхода иерархии в ширину, в результате все братья одного уровня появляются раньше, чем какой-либо их потомок. Ниже показан вариант запроса, в котором содержится раздел SEARCH с требованием обхода иерархии элементов автомобиля в ширину (**пример 16.4**).

```
WITH RECURSIVE PARTS (ASSEMBLY, PART_NUMBER,
       NUMBER_OF_PARTS, COST) AS
     (SELECT CONTAINING_PART, CONTAINED_PART, 1, 0.00
     FROM CAR
     WHERE CONTAINING_PART = ''
   UNION ALL
     SELECT  CAR.CONTAINING_PART, CAR.CONTAINED_PART,
             CAR.NUMBER_OF_PARTS, CAR.NUMBER_OF_PARTS *
             CAR.PART_COST
     FROM CAR, PARTS
     WHERE PARTS.PART_NUMBER = CAR.CONTAINING_PART)
     SEARCH BREADTH FIRST BY CONTAINING_PART, CONTAINED_PART
     SET ORDER_COLUMN
SELECT PART_NUMBER, NUMBER_OF PARTS, COST
FROM PARTS
ORDER BY ORDER_COLUMN;
```

В списке столбцов сортировки раздела SEARCH должны указываться имена столбцов виртуальной таблицы, определенной в разделе WITH. Поскольку в данном случае мы хотим, чтобы в результате сначала появлялись все конструктивные элементы одного уровня (CONTAINING_PART), а затем все их подэлементы (CONTAINED_PART), в список выборки рекурсивного запроса PARTS добавлен столбец CONTAINING_PART, который не используется нигде, кроме раздела SEARCH. В разделе SET к результирующей таблице рекурсивного запроса добавлен столбец, который мы назвали ORDER_COLUMN. Название соответствует природе столбца, потому что при выполнении рекурсивного запроса в этот столбец автоматически заносятся значения, характеризующие порядок генерируемых строк в соответствии с выбранным способом обхода иерархии. Чтобы строки результата основного запроса появлялись в должном порядке, в этом запросе требуется наличие раздела ORDER BY с указанием столбца, определенного в разделе SET.

Раздел CYRCLE

Наконец, обсудим, для чего нужен раздел CYRCLE. Дело в том, что иногда сами данные, хранимые в таблицах базы данных, могут иметь цик-

лическую природу. Представим себе, например, компанию, в которой существует совет директоров, являющийся высшим органом управления компанией. Обычным случаем является тот, когда, по крайней мере, один из членов совета директоров является простым служащим этой же компании (например, он может входить в совет директоров как представитель профсоюза). Назовем данного члена совета директоров EMP_DIR. Как член совета директоров, EMP_DIR «управляет» деятельностью президента компании. С другой стороны, как служащий компании, EMP_DIR находится в прямом или косвенном подчинении у президента компании. Такое положение может привести к зацикливанию выполнения рекурсивных запросов. Раздел CYRCLE обеспечивает некоторую возможность распознавать подобные ситуации. Если у пользователя имеется полная уверенность в отсутствии циклов в данных, к которым адресуется рекурсивный запрос, то использование раздела CYRCLE не требуется.

Подход к распознаванию зацикленных запросов, принятый в SQL, состоит в том, что распознаются данные, которые уже участвовали ранее в формировании результата рекурсивного запроса. При наличии раздела CYRCLE при добавлении к результату строк, удовлетворяющих условию запроса, такие строки помечаются указанным значением, которое означает, что эти строки уже вошли в результат. При попытке добавления к результату каждой новой строки проверяется, не находится ли она уже в результате, т. е. не помечена ли она этим указанным в разделе CYRCLE значением. Если это действительно так, то считается, что имеет место цикл, и дальнейшее выполнение рекурсивного запроса прекращается.

Обсудим все это более формально. Для удобства воспроизведем еще раз синтаксис раздела CYRCLE.

```
cycle_clause ::= CYCLE cycle_column_name_comma_list
        SET cycle_mark_column_name TO value_expression_1
        DEFAULT value_expression_2
        USING path_column_name
```

В списке cycle_column_name_comma_list указываются имена одного или нескольких столбцов, которые используются для идентификации новых строк результата на основе строк, уже входящих в результат. Например, в примерах 16.3 и 16.4 столбец CONTAINED_PART связывает конструктивный элемент автомобиля с входящими в его состав подэлементами (через значения их столбцов CONTAINING_PART). Раздел SET приводит к образованию нового столбца результирующей таблицы. Для строк, которые попадают в результат первый раз, в столбец cycle_mark_column_name заносится значение выражения value_expression_2. В повторно заносимых строках значение столбца — value_expression_1. Типом данных

этого столбца является тип символьных строк длины один, так что в качестве value_expression_1 и value_expression_2 разумно использовать константы '0' и '1' или 'Y' и 'N'.

Раздел USING приводит к образованию еще одного дополнительного столбца результата с именем path_column_name. Типом данных столбца является ARRAY, причем кардинальность этого типа предполагается достаточно большой, чтобы сохранить информацию обо всех строках, попавших в результат. Элементы массива имеют «строчный тип» (row type), содержащий столько столбцов, сколько их указано в списке раздела CYRCLE. Каждый элемент массива соответствует строке результата, и в его столбцах содержится копия значений соответствующих столбцов этой строки. Вот пример запроса, содержащего раздел CYRCLE (**пример 16.5**):

```
WITH RECURSIVE PARTS (PART_NUMBER, NUMBER_OF_PARTS, COST) AS
   (SELECT CONTAINED_PART, 1, 0.00
      FROM CAR
      WHERE CONTAINING_PART = ''
   UNION ALL
      SELECT CAR.CONTAINED_PART, CAR.NUMBER_OF_PARTS,
         CAR.NUMBER_OF_PARTS * CAR.PART_COST
      FROM CAR, PARTS
      WHERE PARTS.PART_NUMBER = CAR.CONTAINING_PART)
      CYRCLE CONTAINED_PART
      SET CYCLEMARK TO 'Y' DEFAULT 'N'
      USING CYRCLEPATH
SELECT PART_NUMBER, SUM(NUMBER_OF PARTS), SUM(COST)
FROM PARTS
ORDER BY PART_NUMBER;
```

Имена столбцов CYCLEMARK и CYRCLEPATH выбраны произвольным образом — требуется только, чтобы имена этих столбцов отличались от имен столбцов рекурсивного запроса. При выполнении запроса строки, удовлетворяющие его условию, накапливаются в результирующей таблице. Но, кроме того, эти строки «кэшируются» в столбце CYRCLEPATH. При попытке добавления к результату новой строки на основе текущего содержимого столбца CYRCLEPATH проверяется, не содержится ли она уже в результате. Если не содержится, то данные об этой строке добавляются к столбцу CYRCLEPATH (к массиву добавляется новый элемент), в столбец CYCLEMARK этой строки заносится значение 'N', и строка добавляется к результату. Иначе в столбец CYCLEMARK соответствующей строки результата заносится значение 'Y', означающее, что от этой строки начинается цикл.

Рекурсивные представления

Рекурсивным называется представление, в определяющем выражении запроса которого используется имя этого же представления. В представлениях может использоваться и прямая, и взаимная рекурсия. Синтаксис оператора определения рекурсивного запроса выглядит следующим образом:

```
CREATE RECURSIVE VIEW table_name
   [ column_name_comma_list ]
   AS query_expression
```

Хотя для того, чтобы представление было рекурсивным, требуется рекурсивность определяющего выражения запроса (т. е. в нем должна присутствовать спецификация RECURSIVE); наличие избыточного ключевого RECURSIVE в определении рекурсивного представления является обязательным. Как говорят авторы стандарта, это сделано для того, чтобы избежать случайного появления непредусмотренных рекурсивных представлений. Наконец, обратите внимание на то, что еще не обсуждавшийся нами необязательный раздел WITH CHECK OPTION не может присутствовать в определении рекурсивного представления (по той причине, что разработчики стандарта не смогли найти разумной интерпретации для комбинации RECURSIVE и WITH CHECK OPTION).

В заключение этого раздела могу сказать, что лично мне механизм рекурсии, предлагаемый в стандарте SQL, представляется громоздким и ограниченным. Кроме того, насколько мне известно, компании, поставляющие SQL-ориентированные СУБД, не спешат внедрять в свои продукты средства рекурсии в соответствии со стандартом SQL:1999 (или, по крайней мере, не слишком их афишируют).

Заключение

Если вернуться к синтаксическим определениям разд. «Общие синтакические правила построения скалярных выражений» лекции 13, то можно убедиться, что в последних четырех лекциях мы рассмотрели все варианты организации оператора SELECT языка SQL (за исключением конструкций collection_derived_table и ONLY (table_or_query_name), относящихся к объектным расширениям языка SQL).

Для общего понимания языка на модельном уровне более важными являются предыдущие три лекции. Данная лекция включена в курс, скорее, с целью общего ознакомления читателей с новыми возможностями оператора выборки, чем с целью их подробного описания. С большой вероятностью средства формулировки аналитических и рекурсивных запросов языка SQL будут пересматриваться при подготовке следующих вариантов стандарта языка.

Лекция 17. Язык баз данных SQL: средства манипулирования данными

Название этой лекции не совсем правильно отражает ее реальное содержание. Собственно средствам языка SQL, предназначенным для обновления базы данных, посвящается только один из трех основных разделов. Однако и следующие два раздела непосредственно примыкают к этой теме. В разделе «Представления, над которыми возможны операции обновления» рассматриваются возможности выполнения операций обновления базы данных через представляемые таблицы. Обсуждается проблема в целом, подход к ее решению, применявшийся в ранних вариантах стандарта SQL, и решение, принятое в стандарте SQL:1999. Последний основной раздел лекции посвящается механизму триггеров, специфицированному в стандарте SQL:1999. Здесь тоже имеется явная связь с основной темой лекции, потому что главным условием срабатывания триггера является выполнение некоторой операции обновления базы данных, и действия, выполняемые в триггерной процедуре, как правило, тоже связаны с обновлением базы данных.

Ключевые слова: оператор INSERT, вставка всех строк указанной таблицы, вставка явно заданного набора строк, вставка строк результата запроса, оператор UPDATE, оператор DELETE, представления, над которыми возможны операции обновления (updatable view), материализация представления, представления, допускающие применение операций обновления, в стандарте SQL/92, представления, допускающие применение операций обновления, в стандарте SQL:1999, потенциальная применимость операций обновления, применимость операций обновления, простая применимость операций обновления, применимость операции вставки, аксиоматическая функциональная зависимость, известная функциональная зависимость, BUC-множество столбцов, BPK-множество столбцов, раздел WITH CHECK OPTION определения представления, режимы проверки CASCADED и LOCAL, триггер, триггерная процедура, активная база данных, механизм триггеров в SQL:1999, предметная таблица (subject table), инициирующий оператор SQL (triggering SQL statement), триггеры BEFORE и AFTER, триггеры INSERT, UPDATE и DELETE, триггеры ROW и STATEMENT, раздел WHEN определения триггера, инициируемый SQL-оператором (triggered_SQL_statement), составной инициируемый SQL-оператор, SQL-процедура (SQL procedure), SQL/PSM, выполнение триггеров, контекст выполнения триггера, переходная таблица, порядок выполнения триггеров, триггеры и ссылочные действия.

Введение

Базы данных, по крайней мере, в приложениях категории OLTP, являются высоко динамичными объектами. В таких приложениях на две операции выборки данных в среднем приходится одна операция обновления содержимого базы данных (добавления новых данных, удаления или модификации существующих данных). Поэтому для пользователей и разработчиков OLTP-приложений средства *манипулирования данными* по важности находятся на втором месте после средств выборки данных.

В этой лекции мы обсудим средства манипулирования данными, входящие в *прямой* SQL. Заметим, что с практической точки зрения более важными являются средства манипулирования данными, выходящие за пределы прямого SQL и присутствующие во *встраиваемом* и *динамическом* SQL. Но, как мы неоднократно отмечали, в этом курсе мы не обсуждаем возможности использования SQL для создания приложений. По мнению автора, материал данной лекции полезен для общего понимания специфики операторов манипулирования данными, а расширения этих операторов, присутствующие во встраиваемом и динамическом SQL, в любом случае нужно изучать совместно с другими аспектами подобных уровней языка.

Лекция состоит из трех основных разделов. В первом разделе мы обсудим синтаксис и семантику операторов манипулирования данными, полагая, что они действуют над базовыми таблицами. Во втором разделе будет продемонстрировано, что в ряде случаев, специфицированных в стандарте языка SQL, операторы манипулирования данными можно применять к порождаемым таблицам и представлениям с однозначным отображением результатов действия этих операторов на соответствующие базовые таблицы. Третий раздел посвящен механизму триггеров, которые, по существу, представляют собой «хранимые процедуры», автоматически вызываемые при возникновении соответствующих условий. Триггеры не обязательно связываются с действиями, производимыми при манипулировании данных, но, поскольку одно из основных функций этого механизма состоит в поддержании целостности баз данных, как правило, такая связь имеется. Поэтому мы включили обсуждение механизма триггеров в соответствии со стандартом SQL именно в данную лекцию.

Базовые средства манипулирования данными

К базовым средствам манипулирования данными языка SQL относятся «поисковые» варианты операторов `UPDATE` и `DELETE`. Эти варианты называются поисковыми, потому что при задании соответствующей операции задается логическое условие, налагаемое на строки адресуемой

оператором таблицы, которые должны быть подвергнуты модификации или удалению. Кроме того, в такую категорию языковых средств входит оператор INSERT, позволяющий добавлять строки в существующие таблицы. Логично начать изложение именно с оператора INSERT, поскольку, для того чтобы можно было что-либо модифицировать в таблицах или удалять из таблиц, нужно, чтобы в таблицах содержались какие-то строки.

Оператор INSERT для вставки строк в существующие таблицы

Общий синтаксис оператора INSERT выглядит следующим образом:

```
INSERT INTO table_name
       { [ (column_commalist) ] query_expression
     | DEFAULT VALUES
```

На вид синтаксические правила кажутся очень простыми, пока не вспомнишь, что обозначает синтаксическая категория query_expression (см. раздел «Общие синтаксические правила построения скалярных выражений» лекции 13). Даже если ограничиться простейшей составляющей этой конструкции (simple_table), то мы имеем следующие возможности:

```
simple_table ::= query_specification
               | table_value_constructor
               | TABLE table_name
```

Вставка всех строк указанной таблицы

Тем самым, стандарт допускает вставку в указанную таблицу всех строк некоторой другой таблицы (вариант table_name). Эта другая таблица может быть как базовой, так и представляемой. Естественно, что в последнем случае в определении представления не должны присутствовать ссылки на таблицу, в которую производится вставка. При использовании данного варианта оператора вставки число столбцов вставляемой таблицы должно совпадать с числом столбцов таблицы, в которую производится вставка, или с числом столбцов, указанных в списке column_commalist, если этот список задан. Типы данных соответствующих столбцов вставляемой таблицы и таблицы, в которую производится вставка, должны быть совместимыми. Если в операции задан список column_commalist и в нем содержатся не все имена столбцов таблицы, в которую производится вставка, то в оставшиеся столбцы во всех строках заносятся значения столбцов по умолчанию. Если для какого-либо из оставшихся столбцов

значение по умолчанию не определено, при выполнении операции вставки фиксируется ошибка.

Чтобы привести пример этого варианта операции INSERT (**пример 17.1**), предположим, что в базе данных EMP-DEPT-PRO имеется еще одна промежуточная таблица EMP_TEMP, в которой временно хранятся данные о служащих, проходящих испытательный срок. Пусть эта таблица имеет следующий заголовок:

EMP_TEMP:

EMP_NO : EMP_NO
EMP_NAME : VARCHAR
EMP_BDATE : DATE

В таблице EMP_TEMP хранятся не полные сведения о служащих, а именно те, которые требуются на время испытательного срока. Если выполнить операцию

```
INSERT INTO EMP (EMP_NO, EMP_NAME, EMP_BDATE) TABLE EMP_TEMP;
```

то в основной таблице EMP появятся строки, соответствующие сотрудникам, проходившим испытательный срок. При этом в столбцах EMP_NO, EMP_NAME, EMP_BDATE этих строк будут содержаться данные, взятые из таблицы EMP_TEMP, а в столбцах EMP_SAL, DEPT_NO, PRO_NO будут находиться значения, определенные для данных столбцов по умолчанию. Конечно, поскольку столбец EMP_NO является первичным ключом таблицы EMP (по всей видимости, и таблицы EMP_TEMP), операция вставки будет успешно выполнена только в том случае, когда ограничение первичного ключа таблицы EMP не будет нарушено (конечно же, требуется выполнение и всех других ограничений целостности, определенных для таблицы EMP).

Вставка явно заданного набора строк

Теперь обратимся к варианту оператора INSERT, в котором набор вставляемых строк задается явно с использованием синтаксической конструкции table_value_constructor. Напомним синтаксические правила, определяющие эту конструкцию:

```
table_value_constructor ::= VALUES row_value_constructor_comma_list
row_value_constructor ::= row_value_constructor_element
        | [ ROW ] (row_value_constructor_element_comma_list)
        | row_subquery
row_value_constructor_element ::= value_expression | NULL | DEFAULT
```

Самый простой пример использования этого варианта оператора вставки состоит в занесении в таблицу EMP явно задаваемых данных о новом служащем **(пример 17.2)**:

```
INSERT INTO EMP
   ROW (2445, 'Brown', '1985-04-08', 16500.00, 630, 772);
```

В этом примере явно заданы значения всех столбцов заносимой строки (как показывают синтаксические правила, ключевое слово ROW можно опустить). Возможен и такой вариант **(пример 17.2a)**:

```
INSERT INTO EMP
   ROW ( 2445, DEFAULT, NULL, DEFAULT, NULL, NULL);
```

В этом случае мы знаем о новом служащем очень мало, но уверены в том, что его имя и размер заработной платы должны быть назначены по умолчанию, а про дату рождения, номер отдела и номер проекта ничего не известно. Обратите внимание, что выполнение подобной операции не нарушает ограничения целостности таблицы EMP.

Если обладать полной информацией об определении таблицы EMP, то формулировку операции примера 17.2a можно переписать короче следующим эквивалентным образом **(пример 17.2b)**:

```
INSERT INTO EMP (EMP_NO) 2445;
```

Вспомним теперь, что одной из разновидностей value_expression_primary является scalar_subquery (см. раздел «Скалярные выражения» лекции 13). Это означает, что в список элементов конструктора строки могут входить скалярные запросы, т. е. запросы, результат выполнения которых состоит из единственной строки, включающей единственный столбец. Поэтому допустима, например, такая операция вставки **(пример 17.3)**:

```
INSERT INTO EMP VALUES
          ROW (2445, (SELECT EMP_NAME
                      FROM EMP
                      WHERE EMP_NO = 2555),
                   '1985-04-08',
                   SELECT EMP_SAL
                      FROM EMP
                      WHERE EMP_NO = 2555),
                   NULL, NULL ),
```

```
ROW (2446, (SELECT EMP_NAME
                FROM EMP
                WHERE EMP_NO = 2556),
           '1978-05-09',
           (SELECT EMP_SAL
                FROM EMP
                WHERE EMP_NO = 2556),
           NULL, NULL );
```

После выполнения этой операции в таблице EMP появятся две новые строки для служащих с уникальными идентификаторами 2445 и 2446, причем первому из них будет присвоено имя и размер заработной платы служащего с уникальным идентификатором 2555, а второму – аналогичные данные о служащем с уникальным идентификатором 2556.

Вставка строк результата запроса

Наконец, обсудим вариант оператора вставки, когда набор вставляемых строк определяется через спецификацию запроса. Предположим, например, что требуется сохранить в отдельной таблице DEPT_SUMMARY сведения о числе служащих каждого отдела, их максимальной, минимальной и суммарной заработной плате. Пусть таблица DEPT_SUMMARY уже создана и имеет следующий заголовок*:

DEPT_SUMMARY:

DEPT_NO : DEPT_NO
DEPT_EMP_NO : INTEGER
DEPT_MAX_SAL : SALARY
DEPT_MIN_SAL : SALARY
DEPT_TOTAL_SAL : SALARY

Тогда заполнить таблицу можно с помощью следующей операции вставки (**пример 17.4**):

```
INSERT INTO DEPT_SUMMARY
   (SELECT DEPT_NO, COUNT(*), MAX (EMP_SAL),
        MIN (EMP_SAL), SUM (EMP_SAL)
      FROM EMP
      GROUP BY DEPT_NO);
```

* Мы не будем приводить полное определение таблицы, включающее требуемые ограничения целостности.

Оператор UPDATE для модификации существующих строк в существующих таблицах

Общий синтаксис оператора UPDATE выглядит следующим образом:

```
UPDATE table_name SET update_assignment_commalist
   WHERE conditional_expression
update_assignment ::= column_name =
                    { value_expression | DEFAULT | NULL }
```

Семантика оператора модификации существующих строк определяется следующим образом:

(1) для всех строк таблицы с именем table_name вычисляется булевское выражение conditional_expression. Строки, для которых значением этого булевского выражения является *true*, считаются подлежащими модификации (обозначим множество таких строк через T_m);

(2) каждая строка s ($s \in T_m$) подвергается модификации таким образом, что значение каждого столбца этой строки, указанного в списке update_assignment_commalist, заменяется значением, указанным в правой части соответствующего элемента списка модификации*. Значения столбцов строки s, не указанные в списке модификации, остаются неизменными.

Приведем примеры операций модификации таблиц.

Пример 17.5. Перевести всех служащих, выполняющих проект с номером 772, в отдел 632 и повысить им заработную плату на 1000 руб.

```
UPDATE EMP SET DEPT_NO = 632, EMP_SAL = EMP_SAL + 1000.00
   WHERE PRO_NO = 772;
```

При выполнении данной операции на первом шаге в таблице EMP будут найдены все строки, относящиеся к служащим, которые участвуют в проекте с номером 772. На втором шаге во всех этих строках значение столбца DEPT_NO будет изменено на 632, а к значению столбца EMP_SAL будет прибавлено 1000.00.

Пример 17.6. Для всех служащих, работающих в отделах, заработная плата менеджеров которых превышает 30000 руб., установить размер заработной платы, на 1000 руб. превышающий средний размер

* Если в правой части элемента модификации присутствует value_expression, в котором содержится запрос, то в случае использования в этом запросе имен столбцов модифицируемой таблицы под значениями этих столбцов понимается значение до модификации.

заработной платы соответствующего отдела, а номера проектов, в которых участвуют эти служащие, сделать неопределенными.

```
UPDATE EMP SET EMP_SAL = (SELECT AVG (EMP1_SAL)
                          FROM EMP EMP1
                          WHERE EMP.DEPT_NO = EMP1.DEPT_NO)
                        + 1000.00, PRO_NO = NULL
     WHERE (SELECT EMP1.EMP_SAL
            FROM EMP EMP1, DEPT
            WHERE EMP.DEPT_NO = DEPT.DEPT_NO
              AND DEPT_MNG = EMP1.EMP_NO AND) > 30000.00;
```

Конечно, если вам больше нравится другой стиль, то запрос, фигурирующий в разделе WHERE, можно переформулировать с использованием вложенного подзапроса (**пример 17.6а**).

```
UPDATE EMP SET EMP_SAL = (SELECT AVG (EMP1_SAL)
                          FROM EMP EMP1
                          WHERE EMP.DEPT_NO = EMP1.DEPT_NO)
                        + 1000.00, PRO_NO = NULL
     WHERE DEPT.NO IN (SELECT DEPT.DEPT_NO
                          FROM EMP, DEPT
                          WHERE DEPT_MNG = EMP_NO
                          AND EMP_SAL > 30000.00);
```

Эти примеры позволяют понять, насколько богаты возможности оператора UPDATE. В разделе WHERE может содержаться любое условие, допускаемое в операторе выборки, а в элементах списка раздела SET может присутствовать любой вид value_expression, в том числе любой запрос, вырабатывающий одиночное значение (скалярный подзапрос).

Оператор DELETE для удаления строк в существующих таблицах

Общий синтаксис оператора DELETE выглядит следующим образом:

```
DELETE FROM table_name
  WHERE conditional_expression
```

В некотором смысле оператор DELETE является частным случаем оператора UPDATE (или, наоборот, действие оператора UPDATE представляет собой комбинацию действий операторов DELETE и INSERT).

Семантика оператора модификации существующих строк определяется следующим образом:

(1) для всех строк таблицы с именем table_name вычисляется булевское выражение conditional_expression. Строки, для которых значением этого булевского выражения является *true*, считаются подлежащими удалению (обозначим множество таких строк через T_d);

(2) каждая строка s ($s \in T_d$) удаляется из указанной таблицы.

С целью иллюстрации приведем два примера операции удаления строк.

Пример 17.7. Удалить из таблицы EMP все строки, относящиеся к служащим, которые участвуют в проекте с номером 772.

```
DELETE FROM EMP WHERE PRO_NO = 772;
```

Пример 17.8. Удалить из таблицы EMP все строки, относящиеся к служащим, размер заработной платы которых превышает размер заработной платы менеджеров их отделов.

```
DELETE FROM EMP WHERE EMP_SAL >
             (SELECT EMP1.EMP_SAL
                FROM EMP EMP1, DEPT
                WHERE EMP.DEPT_NO = DEPT.DEPT_NO
                   AND DEPT.DEPT.MNG = EMP1.EMP_NO);
```

Как и в операторе UPDATE, в разделе WHERE оператора DELETE можно использовать любой вид булевского выражения, допустимого в операторе выборки. Поэтому возможности оператора удаления строк ограничены лишь фантазией пользователя.

Представления, над которыми возможны операции обновления

В разделе «Общие синтакические правила построения скалярных выражений» лекции 13 было введено понятие представления (VIEW). Кратко повторим, что представление – это сохраняемое в каталоге базы данных выражение запросов, обладающее собственным именем и, возможно, собственными именами столбцов. Для удобства повторим синтаксические правила определения представления:

```
create_view ::= CREATE [ RECURSIVE ] VIEW table_name
     [ column_name_comma_list ]
   AS query_expression
   [ WITH [ CASCADED | LOCAL ] CHECK OPTION ]
```

В операциях выборки к любому представлению можно адресоваться таким же образом, как и к любой базовой таблице. Естественно, возникает вопрос: а можно ли использовать имена представлений и в операциях обновления базы данных и если такая возможность допускается, то как это следует понимать?

Напомним, что в соответствии с семантикой языка SQL при выполнении запроса, в разделе FROM которого прямо или косвенно присутствует имя представления, прежде всего, производится *материализация* представления, т. е. вычисляется результат соответствующего выражения запросов, сохраняется во временной базовой таблице, и далее запрос выполняется по отношению к этой базовой таблице. Хотя в реализациях SQL обычно стремятся избегать материализации представлений, любая реализация обязана обеспечить такое выполнение запроса над представлением, которое было бы эквивалентно выполнению запроса с явной материализацией представления.

Если допустить выполнение над представлениями операций обновления (сразу заметим, что, вообще говоря, в языке SQL это всегда разрешалось), то в этом случае семантика материализации явно не подходит. На первое место выходит требование, чтобы операция обновления над представлением однозначно отображалась в одну или несколько операций обновления над теми постоянно хранимыми базовыми таблицами, над которыми прямо или косвенно определено данное представление.

Представления, допускающие применение операций обновления, в стандарте SQL/92

Поскольку базовым элементом выражения запросов является спецификация запроса, прежде всего нужно понять, какой класс спецификаций запросов является *допускающим операции обновления* (термин updatable — *обновляемый*, используемый в стандарте SQL, кажется не слишком удачным в русском варианте). В стандарте SQL/92 спецификация запроса считалась допускающей операции обновления в том и только в том случае, когда выполнялись следующие условия:

- в разделе SELECT спецификации запроса отсутствует ключевое слово DISTINCT (т. е. не требуется удаление строк-дубликатов из результата запроса);
- все элементы списка выборки раздела SELECT являются именами столбцов, и ни одно имя столбца не встречается в этом списке более одного раза;
- в разделе FROM присутствует только одна ссылка на таблицу, и она указывает либо на базовую таблицу, либо на порождаемую таблицу, допускающую операции обновления;

- прямые или косвенные ссылки на базовую таблицу, прямо или косвенно идентифицируемую ссылкой на таблицу в разделе FROM, не встречаются в разделе FROM ни одного подзапроса, участвующего в разделе WHERE спецификации запроса;
- в спецификации запроса отсутствуют разделы GROUP BY и HAVING.

Нетрудно убедиться в том, что эти требования являются достаточными для однозначной интерпретации операций обновления над представлениями. Например, пусть имеется следующая спецификация запроса (**пример 17.9**):

```
SELECT EMP_SAL
   FROM (SELECT EMP_SAL, DEPT_NO
         FROM EMP
         WHERE EMP_NAME = (SELECT EMP_NAME
                           FROM EMP
                           WHERE EMP_NO = 4425))
   WHERE DEPT_NO <> 630;
```

Эту спецификацию можно упростить до эквивалентной формулировки*:

```
SELECT EMP_SAL
   FROM EMP
   WHERE EMP_NAME = (SELECT EMP_NAME
                     FROM EMP
                     WHERE EMP_NO = 4425 )
                        AND DEPT_NO <> 630;
```

Предположим, что с данной спецификацией запроса связано представление с именем EMPSAL. Тогда операция

```
UPDATE EMPSAL SET EMP_SAL = EMP_SAL - 1000.00;
```

эквивалентна операции

```
UPDATE EMP SET EMP_SAL = EMP_SAL - 1000.00
   WHERE EMP_NAME = (SELECT EMP_NAME
                     FROM EMP
                     WHERE EMP_NO = 4425 )
                        AND DEPT_NO <> 630;
```

* Обратите внимание, что формально эта формулировка не отвечает требованиям SQL/92 для спецификаций запросов, допускающих применение операций обновления. Но в действительности здесь вложенный подзапрос вычисляется в единственное значение при отсутствии какой-либо корреляции с внешним вхождением таблицы EMP.

Операция

```
DELETE FROM EMPSAL WHERE EMP_SAL > 20000.00;
```

эквивалентна операции

```
DELETE EMPSAL
   WHERE EMP_SAL > 20000.00 AND
         EMP_NAME = (SELECT EMP_NAME
                     FROM EMP
                     WHERE EMP_NO = 4425 )
                        AND DEPT_NO <> 630;
```

Операция вставки над представлением EMPSAL

```
INSERT INTO EMPSAL 25000.00;
```

трактуется как

```
INSERT INTO EMP
  ROW (DEFAULT, DEFAULT, DEFAULT, 25000.00, DEFAULT, DEFAULT);
```

Понятно, что такая операция будет отвергнута системой, потому что для столбца EMP_NO таблицы EMP значения по умолчанию не определены (это первичный ключ таблицы, значения которого должны явно задаваться в любой операции вставки).

С другой стороны, условия допустимости операций обновления, специфицированные в SQL/92, не являются необходимыми. Например, над представлением EMPMNG, определенным над спецификацией запроса («выбрать данные о служащих, являющихся руководителями отделов»).

```
SELECT *
   FROM EMP
   WHERE EXISTS (SELECT *
                 FROM DEPT
                 WHERE DEPT_MNG = EMP_NO);
```

можно было бы совершенно корректно выполнять операции обновления (с некоторыми оговорками насчет операции вставки; см. ниже в этом разделе).

Представления, допускающие применение операций обновления, в стандарте SQL:1999

В стандарте SQL:1999 правила применимости операций обновления к спецификации запроса существенно уточнены.

Критерии применимости операций обновления

Введены понятия *потенциальной применимости операций обновления, применимости операций обновления, простой применимости операций обновления и применимости операции вставки*. К спецификации запроса потенциально применимы операции обновления в том и только в том случае, когда выполняются следующие условия:

- в разделе SELECT спецификации запроса отсутствует ключевое слово DISTINCT;
- элемент списка выборки раздела SELECT, состоящий из ссылки на некоторый столбец, не может присутствовать в этом списке более одного раза;
- в спецификации запроса отсутствуют разделы GROUP BY и HAVING.

Если выражение запросов отвечает условиям потенциальной применимости операций обновления и в его разделе FROM присутствует только одна ссылка на таблицу, то к каждому столбцу выражения запроса, соответствующему одному столбцу таблицы из раздела FROM, применимы операции обновления. Если выражение запроса отвечает условиям потенциальной применимости операций обновления, но в его разделе FROM присутствуют две или более ссылки на таблицы, то операции обновления применимы к столбцу выражения запросов только при выполнении следующих условий:

- столбец порождается из столбца только одной таблицы из раздела FROM;
- эта таблица используется в выражении запросов таким образом, что сохраняются свойства ее первичного и всех возможных ключей.

Другими словами, к столбцу таблицы, которая отвечает условиям потенциальной применимости операций обновления, применимы операции обновления только в том случае, когда этот столбец может быть однозначно сопоставлен с единственным столбцом единственной таблицы, участвующей в выражении запроса, и каждая строка выражения запроса может быть однозначно сопоставлена с единственной строкой данной таблицы.

Выражение запросов удовлетворяет условию применимости операций обновления, если по крайней мере к одному столбцу выражения запросов применимы операции обновления. Выражение запросов удовлетворяет условию простой применимости операций обновления, если в разделе FROM выражения запросов содержится ссылка только на одну таб-

лицу, и все столбцы выражения запросов удовлетворяют условию применимости операций обновления.

Выражение запросов удовлетворит условию применимости операций вставки, если оно удовлетворяет условию применимости операций обновления; каждая из таблиц, от которых зависит это выражение (т. е. таблиц, на которые имеются ссылки в разделе FROM), удовлетворяет условию применимости операций вставки и выражение запросов не содержит операций UNION, INTERSECT и EXCEPT. Конечно, это определение базируется на том факте, что для любой базовой таблицы условие применимости операции вставки удовлетворяется (при наличии привилегии INSERT, см. следующую лекцию).

Правила функциональных зависимостей

Приведенный набор правил является достаточно грубым. В стандарте SQL:1999 он уточняется набором дополнительных правил, устанавливающих восприимчивость различных языковых конструкций к операциям обновления и вставки. В основе этих правил лежит понятие *функциональной зависимости (Functional Dependency – FD*, см. лекцию 6). Полагая, что в целом понятие функциональной зависимости уже не должно вызывать у читателей каких-либо затруднений, приведем несколько дополнительных определений, требуемых для понимания подхода, используемого в SQL:1999.

- Пусть S обозначает некоторое множество столбцов таблицы T, а SS обозначает некоторое подмножество S ($SS \subset S$). Тогда по первой аксиоме Армстронга (см. раздел «Функциональные зависимости» лекции 6) $SS \rightarrow S$. В терминологии SQL:1999 эта FD называется аксиоматической. Все ФЗ, не являющиеся *аксиоматическими*, называются *неаксиоматическими*.
- Все аксиоматические FD являются *известными FD*. В стандарте определяются правила определения других известных FD. Кроме того, стандарт оставляет свободу для реализаций SQL в пополнении этой системы правил с целью нахождения известных FD, не специфицированных в стандарте.
- Если некоторый столбец $C1$ виртуальной таблицы $T1$ (порождаемой таблицы или представления) определяется путем ссылки на столбец $C2$ некой другой (базовой или виртуальной) таблицы $T2$, на основе которой порождается $T1$, то $C1$ является *двойником* $C2$. Более точно, $C1$ является *двойником* $C2$ *в соответствии с таблицей* $T2$.
- Понятие двойников расширяется на множества столбцов. Если некоторое множество столбцов $S1$ виртуальной таблицы $T1$ определяется (путем отображения «один – в – один») множеством столбцов $S2$ определяющей таблицы $T2$, и каждый столбец из множества $S1$ является

двойником соответствующего столбца из множества *S2*, то *S1* называется *двойником S2 в соответствии с таблицей T2*.

- Если ни в одном из столбцов возможного ключа (набора столбцов, специфицированного в неоткладываемом ограничении уникальности) не допускается наличие неопределенных значений, то это множество столбцов называется BUC-множеством (акроним BUC происходит от *Base table Unique Constraint)*. Любое множество столбцов, являющееся двойником BUC-множества, также есть BUC-множество, так что это свойство распространяется через различные выражения, производящие виртуальные таблицы. Если имеются два множества столбцов *S1* и *S2*, такие, что $S1 \subseteq S2$, $S1 \rightarrow S2$, и *S2* является *BUC*-множеством, то и *S1* является *BUC*-множеством. Могут существовать таблицы, у которых BUC-множество является пустым. Такая таблица может содержать не более одной строки*. С другой стороны, могут существовать таблицы, у которых вообще отсутствуют BUC-множества**.
- Множество столбцов, составляющих первичный ключ таблицы, называется ее *BPK*-множеством (акроним BPK происходит от *Base table Primary Key)*. Понятно, что каждое BPK-множество является BUC-множеством. Если имеются два множества столбцов *S1* и *S2*, такие, что $S1 \subseteq S2$, $S1 \rightarrow S2$, и *S2* является BPK-множеством, то и *S1* является BPK-множеством. Подобно BUC-множествам, BPK-множества могут быть пустыми.

На основе этих определений в стандарте SQL:1999 устанавливаются правила функциональных зависимостей для 11 компонентов языка.

- *Базовые таблицы*. Если у таблицы имеется первичный ключ, то соответствующее множество столбцов образует BPK-множество этой таблицы. Если у таблицы имеется не откладываемое ограничение уникальности, и ни у одного столбца, указанного в этом ограничении, не допускается наличие неопределенных значений, то соответствующее множество столбцов является BUC-множеством. Если множество столбцов *UCL* базовой таблицы —BUC-множество, а *CT* обозначает все множество столбцов этой таблицы, то FD $UCL \rightarrow CT$ представляет собой *известную функциональную зависимость* базовой таблицы.
- *Конструкторы табличных значений*. Поскольку для конструкторов табличных значений невозможно определять ограничения, в стандарте SQL:1999 для них не специфицированы BUC- и BPK-множества. В стандарте не определяются *известные функциональные зависимости* для такого рода конструкций, отличные от аксиоматических. Однако стандарт допускает, чтобы реализации SQL включали дополнительные механизмы определения известных функциональных зависимостей.

* Множество, элементы которого невозможно различить, может быть либо пустым, либо содержать только один элемент.

** В этом случае таблица соответствует понятию мультимножества.

- *Соединенные таблицы.* Если говорить о соединенных таблицах, получаемых в результате применения операций естественного соединения (NATURAL JOIN) или соединения с заданием списка имен столбцов, значения которых должны совпадать (USING), то понятно, что соединенная таблица будет содержать двойников из одной или двух исходных таблиц. Если обозначить через *S* некоторое множество столбцов результирующей таблицы, а через *CT* – все множество столбцов этой таблицы, то *S* является BPK-множеством в том и только в том случае, когда имеет двойника в одной или обеих исходных таблицах. В таком случае во всех столбцах *S* не допускаются неопределенные значения, и FD *S*→*CT* является *известной функциональной зависимостью.*

  В стандарте определяется несколько правил, на основе которых устанавливаются *известные функциональные зависимости* соединенных таблиц, но здесь мы приведем только простейшее из этих правил. Если соединенная таблица производится на основе одной из двух указанных выше операций, то в первой таблице-источнике присутствует один или более столбцов, соответствующих одноименным столбцам второй таблицы-источника. Обозначим через *SLCC* список следующих выражений (элемент списка соответствует общему столбцу):

```
COALESCE (t1.colname, t2.colname) AS colname*
```

  Пусть *JT* обозначает ключевые слова, определяющие тип соединения (INNER, LEFT, RIGHT, FULL и т. д.), и пусть *TN1* и *TN2* обозначают имена таблиц или (если они заданы) имена псевдонимов двух таблиц-источников соответственно. Обозначим через *IR* результат вычисления следующего выражения запросов:

```
SELECT SLCC, T1*, T2* FROM T1 JT JOIN T2;
```

  Тогда, в соответствии с правилами SQL, дополнительными известными функциональными зависимостями являются следующие:

  - если *JT* задает INNER или LEFT, то действует FD COALESCE (T1.Ci, T2.Ci)→T1.Ci для всех i от единицы до числа столбцов в *IR*;
  - если *JT* задает INNER или RIGHT, то действует FD COALESCE (T1.Ci, T2.Ci)→T2.Ci для всех i от единицы до числа столбцов в *IR*.

  Обозначим через *SL* некоторый список выборки. Пусть:

  - если все столбцы первой и второй таблиц-источников являются общими, то *SL* совпадает с *SLCC*;

* Определение выражения COALESCE (V1, V2) см. в разделе «Средства определения, изменения и ликвидации базовых таблиц» лекции 12.

- если среди столбцов таблиц-источников нет общих столбцов, то *SL* состоит из списка столбцов первой таблицы-источника, за которым следует список столбцов второй таблицы-источника;
- если все столбцы первой таблицы-источника являются общими, но у второй таблицы-источника имеются необщие столбцы, то *SL* состоит из *SLCC*, за которым следует список необщих столбцов второй таблицы-источника;
- аналогично, если все столбцы второй таблицы-источника являются общими, но у первой таблицы-источника имеются не общие столбцы, то *SL* состоит из *SLCC*, за которым следует список не общих столбцов первой таблицы-источника;
- наконец, если среди столбцов первой таблицы-источника и среди столбцов второй таблицы-источника имеются необщие столбцы, то *SL* состоит из *SLCC*, за которым следует список необщих столбцов первой таблицы-источника, а далее располагается список не общих столбцов второй таблицы-источника.

Тогда, в соответствии со стандартом, известными функциональными зависимостями виртуальной таблицы, получаемой путем соединения, являются известные функциональные зависимости выражения

```
SELECT SL FROM IR;
```

- *Ссылки на таблицы.* Столбцы виртуальной таблицы, производимой по ссылке на таблицу, являются естественными двойниками столбцов таблицы, которая идентифицируется ссылкой. Поэтому BUC- и BPK-множества результирующей таблицы являются двойниками BUC- и BPK-множеств исходной таблицы, и известные функциональные зависимости результирующей таблицы получаются путем замены имен столбцов исходной таблицы на имена столбцов результирующей таблицы в известных функциональных зависимостях исходной таблицы.
- *Раздел FROM.* Описывая в лекции 13 общую семантику оператора выборки, мы отмечали, что на первом шаге выполнения этого оператора производится (виртуальная) таблица, являющаяся расширенным декартовым произведением всех таблиц, специфицированных в разделе FROM. Поэтому в стандарте SQL естественным образом формулируются следующие правила. Если в списке ссылок на таблицы раздела FROM содержится всего одна ссылка, то BUC- и BPK-множества результирующей таблицы являются двойниками BUC- и BPK-множеств исходной таблицы. Если в списке раздела FROM содержатся две или более ссылки на таблицы, то, в соответствии со стандартом, BUC- и BPK-множества результирующей таблицы не определены. Известные функциональные зависимо-

сти результирующей таблицы состоят из известных функциональных зависимостей каждой таблицы, специфицированной в разделе FROM.

- *Раздел WHERE.* В стандарте содержится набор правил, позволяющих определить BUC- и BPK-множества результирующей таблицы этого раздела*, а также известные функциональные зависимости результирующей таблицы. Правила основываются на особенностях поведения предиката сравнения по равенству и логической операции AND.
- *Раздел GROUP BY.* Для определения BUC- и BPK-множеств и известных функциональных зависимостей результирующей таблицы раздела GROUP BY требуется фактическое образование в результирующей таблице нового столбца, значения которого могли бы каким-то образом идентифицировать строки исходной таблицы, образующие группы сгруппированной таблицы.
- *Раздел HAVING.* BUC- и BPK-множества и известные функциональные зависимости результирующей таблицы раздела HAVING получаются из соответствующих множеств и FD таблицы, к которой применяется этот раздел**, на основе правил, связанных с условным выражением раздела HAVING (как и в случае условия раздела WHERE, в данных правилах учитываются операции сравнения по равенству и логические операции AND).
- *Раздел SELECT.* На определение BUC- и BPK-множеств и известных функциональных зависимостей результата спецификации запроса влияет наличие в списке выборки выражений (value_expression), отличных от ссылок на столбцы.
- *Выражение запроса.* На определение BUC- и BPK-множеств и известных функциональных зависимостей результата выражения запроса влияет наличие в этом выражении операций UNION, INTERSECT и EXCEPT. В стандарте отсутствуют какие-либо правила для определения функциональных зависимостей в результатах рекурсивных запросов. Отмечается лишь возможность введения таких правил в реализациях.

Раздел WITH CHECK OPTION определения представления

Пусть в базе данных имеется упрощенная таблица EMP, содержащая следующее множество строк (как в примере с GROUP BY ROLLUP разделе «Возможности формулирования аналитических запросов» лекции 16).

Предположим, что в базе данных имеется представление RICH_EMP, определенное следующим образом:

* Напомним из лекции 13, что в соответствии с семантикой оператора выборки в результат раздела WHERE входят все строки результата раздела FROM, для которых результатом вычисления логического условия раздела WHERE является *true*.

** Напомним из лекции 13, что на вход раздела HAVING подается результат раздела GROUP BY, если этот раздел присутствует в спецификации запроса, иначе — результат раздела WHERE, если этот раздел присутствует в спецификации запроса, иначе — результат раздела FROM.

EMP

EMP_NO	DEPT_NO	EMP_BDATE	EMP_SAL
2440	1	1950	15000.00
2441	1	1950	16000.00
2442	1	1960	14000.00
2443	1	1960	19000.00
2444	2	1950	17000.00
2445	2	1950	16000.00
2446	2	1960	14000.00
2447	2	1960	20000.00
2448	3	1950	18000.00
2449	3	1950	13000.00
2450	3	1960	21000.00
2451	3	1960	22000.00

```
CREATE VIEW RICH_EMP AS
   SELECT *
      FROM EMP
      WHERE EMP_SAL > 18000.00;
```

Понятно, что в соответствии с правилами SQL (и здравым смыслом) над этим представлением можно выполнять операции обновления. Как видно, в таблице EMP содержится строка, которая соответствует служащему с номером 2447, получающему зарплату в размере 20000 руб. Естественно, эта строка будет присутствовать в виртуальной таблице RICH_EMP. Поэтому можно было бы выполнить, например, операцию

```
UPDATE RICH_EMP
   SET EMP_SAL = EMP_SAL - 3000
   WHERE EMP_NO = 4452;
```

Но если выполнение такой операции действительно допускается, то в результате строка, соответствующая служащему с номером 2447, исчезнет из виртуальной таблицы RICH_EMP! Аналогичный эффект возникнет при выполнении операции вставки

```
INSERT INTO RICH_EMP (EMP_NO) 2452;
```

В базовой таблице EMP появится строка, в которой значением столбца EMP_NO будет 2452, а значения остальных столбцов будут установлены по умолчанию. В частности, значением столбца EMP_SAL будет 10000.00. Тем самым, если подобная операция вставки действительно допустима, то мы вставили в виртуальную таблицу RICH_EMP строку, которую в этой виртуальной таблице увидеть невозможно.

Чтобы избежать такого противоречивого поведения представляемых таблиц, нужно включать в определение представления раздел WITH CHECK OPTION. При наличии этого раздела до реального выполнения операций модификации или вставки строк через представление для каждой строки будет проверяться, что она соответствует условиям представления. Если данное условие не выполняется хотя бы для одной модифицируемой или вставляемой строки, то операция полностью отвергается. В некотором смысле (при наличии раздела WITH CHECK OPTION) условие выборки, содержащееся в выражении запросов представления, можно считать ограничением целостности этого представления.

Режимы проверки CASCADED и LOCAL

Вспомним теперь, что в полном виде синтаксис раздела WITH CHECK OPTION может включать ключевые слова CASCADED или LOCAL. Обсудим их смысл. Предположим, что представление *V2* определяется над представлением *V1* следующим образом:

```
CREATE VIEW V2 AS
   SELECT …
   FROM V1
   WHERE …
   [ WITH [ CASCADED | LOCAL ] CHECK OPTION ]
```

Пусть над *V2* выполняется некоторая операция *O* обновления базы данных. Тогда:

- если представление *V2* определялось без раздела WITH CHECK OPTION, то при выполнении операции *O* будут проверяться все условия, определяющие ограничения целостности *V1* (если в определении *V1* присутствовал раздел WITH CHECK OPTION), но никаким образом не будут учитываться условия выборки, содержащееся в выражении запросов представления *V2*;
- если в определении представления *V2* содержался раздел WITH LOCAL CHECK OPTION, то при выполнении операции *O* будут проверяться все условия, определяющие ограничения целостности *V1*, и все условия, содержащееся в выражении запросов представления *V2*;

- наконец, если в определении представления *V2* содержался раздел WITH CASCADED CHECK OPTION, то при выполнении операции *O* будут проверяться все условия, определяющие ограничения целостности *V1* (так, как если бы в определении *V1* присутствовал раздел WITH CASCADED CHECK OPTION). Тем самым, будут проверяться все ограничения целостности, установленные для всех базовых таблиц, на которых основывается определение *V1*; все условия всех представлений, определенных над этими базовыми таблицами; и, конечно, все условия, содержащиеся в выражении запросов представления *V2*.

Примеры результатов действия раздела WITH CHECK OPTION

Чтобы пояснить результаты действия раздела WITH CHECK OPTION, допустим, что в базе данных присутствуют определения двух представлений MIDDLE_RICH_EMP и MORE_RICH_EMP:

```
CREATE VIEW MIDDLE_RICH_EMP AS
   SELECT *
      FROM EMP
      WHERE EMP_SAL < 20000.00
      [ WITH [ CASCADED | LOCAL ] CHECK OPTION ];
CREATE VIEW MORE_RICH_EMP AS
   SELECT *
      FROM MIDDLE_RICH_EMP
      WHERE EMP_SAL > 18000.00
      [ WITH [ CASCADED | LOCAL ] CHECK OPTION ];
```

Очевидно, что в тело (материализованного) представления MIDDLE_RICH_EMP будут входить следующие строки базовой таблицы EMP:

MIDDLE_RICH_EMP

EMP_NO	DEPT_NO	EMP_BDATE	EMP_SAL
2440	1	1950	15000.00
2441	1	1950	16000.00
2442	1	1960	14000.00
2443	1	1960	19000.00
2444	2	1950	17000.00
2445	2	1950	16000.00
2446	2	1960	14000.00
2448	3	1950	18000.00
2449	3	1950	13000.00

В тело (материализованного) представления MORE_RICH_EMP будут входить следующие строки представляемой таблицы MIDDLE_RICH_EMP:

MORE_RICH_EMP

EMP_NO	DEPT_NO	EMP_BDATE	EMP_SAL
2443	1	1960	19000.00

В каждом из представлений MIDDLE_RICH_EMP и MORE_RICH_EMP может отсутствовать или присутствовать (в одном из двух видов) раздел WITH CHECK OPTION. В совокупности возможен один из девяти случаев:

MORE_RICH_EMP / MIDDLE_RICH_EMP	none	LOCAL	CASCADED
none	Случай 1	Случай 2	Случай 3
LOCAL	Случай 4	Случай 5	Случай 6
CASCADED	Случай 7	Случай 8	Случай 9

Чтобы рассмотреть каждый из возможных случаев по отдельности, обсудим, что будет происходить в каждом случае при выполнении следующих двух операций модификации строк (будем называть эти операции *U1* и *U2* соответственно)*:

```
UPDATE MORE_RICH_EMP
   SET EMP_SAL = EMP_SAL + 7000.00;
UPDATE MORE_RICH_EMP
   SET EMP_SAL = EMP_SAL - 7000.00;
```

Случай 1. Ни в одном из представлений не содержится раздел WITH CHECK OPTION.

Первый неожиданный результат состоит в том, что после выполнения операции *U1* тело представления MORE_RICH_EMP оказывается пустым. Действительно, у единственной строки таблицы EMP (со значением EMP_NO, равным 2443), одновременно удовлетворяющей условиям обоих представлений, столбец EMP_SAL принимает значение 26000.00. После этого строка перестает удовлетворять условию представления MIDDLE_RICH_EMP и исчезает из результирующей таблицы MORE_RICH_EMP. Этот результат может быть особенно неожиданным для пользователей базы данных, которым известно, что условие представления MORE_RICH_EMP

* Будем считать, что тем, кто пользуется представлением MORE_RICH_EMP, неизвестно ограничение EMP_SAL < 20000.00, на котором основывается представление MIDDLE_RICH_EMP.

имеет вид EMP_SAL > 18000.00, и соблюдение этого условия должно сохраняться при увеличении размера зарплаты.

Выполнение операции *U2* также приведет к опустошению тела MORE_RICH_EMP (в базовой таблице EMP не останется ни одной строки, удовлетворяющей условию этого представления). Возможно, это будет достаточно естественно для пользователей представления MORE_RICH_EMP, которым известно условие представления, но те, кто работает с представлением MIDDLE_RICH_EMP, с удивлением обнаружат в теле результирующей таблицы новые строки.

Случай 2. В определении представления MIDDLE_RICH_EMP содержится раздел WITH LOCAL CHECK OPTION, а в определении MORE_RICH_EMP раздел WITH CHECK OPTION отсутствует.

В этом случае, в соответствии с первыми двумя правилами проверки корректности выполнения операций обновления над представлениями, операция *U1* должна быть отвергнута системой (поскольку ее выполнение нарушает условие представления MIDDLE_RICH_EMP). Но заметим, что такое поведение системы будет совершенно неожиданным и непонятным для тех пользователей базы данных, которым известно только определение «верхнего» представления MORE_RICH_EMP, поскольку операция *U1* явно не может нарушить видимое ими ограничение.

С другой стороны, операция *U2* будет успешно выполнена и попрежнему приведет к опустошению тела результирующей таблицы представления MORE_RICH_EMP.

Случай 3. В определении представления MIDDLE_RICH_EMP содержится раздел WITH CASCADED CHECK OPTION, а в определении MORE_RICH_EMP раздел WITH CHECK OPTION отсутствует.

В этой ситуации будут проверяться условия, содержащиеся в определении представления MIDDLE_RICH_EMP, а также все ограничения целостности таблицы EMP и всех других представлений, определенных над этой базовой таблицей. В результате операция *U1* будет отвергнута системой, а операция *U2* будет «успешно» выполнена. Другими словами, повторится Случай 2.

Случай 4. В определении представления MIDDLE_RICH_EMP раздел WITH CHECK OPTION отсутствует, а в определении MORE_RICH_EMP содержится раздел WITH LOCAL CHECK OPTION.

Понятно, что в этом варианте операция *U2* не сработает (ее выполнение не будет допущено условием «ограничения целостности» представления MORE_RICH_EMP). Но операция *U1* (увеличение размера зарплаты слу-

жащих) будет успешно выполнена, поскольку она не противоречит локальным ограничениям представления MORE_RICH_EMP.

Случай 5. В определениях представлений MIDDLE_RICH_EMP и MORE_RICH_EMP содержится раздел WITH LOCAL CHECK OPTION.

Выполнение обеих операций *U1* и *U2* будет справедливо отвергнуто. На первый взгляд все в порядке. Но если над представлением MORE_RICH_EMP будет определено еще одно представление *V*, то мы можем получить ситуацию Случая 2, где *V* будет играть роль MORE_RICH_EMP, а MIDDLE_RICH_EMP – роль MORE_RICH_EMP.

Случай 6. В определении представления MIDDLE_RICH_EMP содержится раздел WITH CASCADED CHECK OPTION, а в определении MORE_RICH_EMP содержится раздел WITH LOCAL CHECK OPTION.

Снова, если над представлением MORE_RICH_EMP будет определено еще одно представление *V*, то мы можем попасть в ситуацию Случая 2, где *V* будет играть роль MORE_RICH_EMP, а MIDDLE_RICH_EMP – роль MORE_RICH_EMP.

Случай 7. В определении представления MIDDLE_RICH_EMP раздел WITH CHECK OPTION отсутствует, а в определении MORE_RICH_EMP содержится раздел WITH CASCADED CHECK OPTION.

Если над представлением MORE_RICH_EMP будет определено еще одно представление *V*, то мы можем попасть в ситуацию Случая 3, где *V* будет играть роль MORE_RICH_EMP, а MIDDLE_RICH_EMP – роль MORE_RICH_EMP.

Случай 8. В определении представления MIDDLE_RICH_EMP содержится раздел WITH LOCAL CHECK OPTION, а в определении MORE_RICH_EMP – раздел WITH CASCADED CHECK OPTION.

Если над представлением MORE_RICH_EMP будет определено еще одно представление *V*, то мы можем получить ситуацию Случая 3, где *V* будет играть роль MORE_RICH_EMP, а MIDDLE_RICH_EMP – роль MORE_RICH_EMP.

Случай 9. В определениях представлений MIDDLE_RICH_EMP и MORE_RICH_EMP содержится раздел WITH CASCADED CHECK OPTION.

Только в этом случае операции обновления будут выполняться корректно, независимо от того, имеются ли в базе данных представления, определенные над MORE_RICH_EMP или между MORE_RICH_EMP, MIDDLE_RICH_EMP и EMP.

Очевидный вывод из приведенного анализа заключается в том, что единственным способом обеспечить корректность выполнения операций обновления через представления (допускающие операции обновления)

является включение в определение каждого представления раздела WITH CASCADED CHECK OPTION. В этом случае поведение системы будет оставаться корректным при введении дополнительных представлений над представлением MORE_RICH_EMP, между представлениями MORE_RICH_EMP и MIDDLE_RICH_EMP или между представлением MIDDLE_RICH_EMP и базовой таблицей EMP, если в определениях всех этих представлений присутствует раздел WITH CASCADED CHECK OPTION.

Исторический очерк

Завершим обсуждение возможностей применения операций обновления к виртуальным таблицам небольшим экскурсом в историю. На протяжении более чем тридцатилетней истории реляционных баз данных вопрос о возможности однозначной интерпретации операций обновления баз данных через виртуальные таблицы интересовал многих исследователей. Причины этого интереса состоят в следующем.

Во-первых, как отмечалось в лекции 3, одной из наиболее привлекательных черт реляционной алгебры является замкнутость относительно понятия отношения. В любой алгебраической операции, операндом которой является отношение, в качестве операнда можно использовать алгебраическое выражение. С другой стороны, имеется явное неравноправие по отношению к операциям обновления. Мы можем вставлять, модифицировать и удалять кортежи в базовых отношениях, но не можем (в общем случае) применять эти операции к алгебраическим выражениям. Хотелось максимальным образом устранить подобное неравноправие.

Во-вторых, на первый взгляд задача не является слишком трудной (по крайней мере, если оставаться в пределах реляционной алгебры). Действительно, базовых операций совсем немного, и каждая базовая операция очень проста.

К сожалению, это ощущение простоты проблемы оказалось обманчивым. Было выполнено множество исследований, опубликовано множество статей (нам кажется нецелесообразным приводить список этих статей в данном курсе), но так и не удалось обнаружить полное множество алгебраических выражений, для которых возможна однозначная интерпретация операций обновления. На мой взгляд, данная ситуация оказала заметное влияние на подход к решению проблемы применимости операций обновления к виртуальным таблицам, которым руководствуются разработчики языка SQL.

В двух первых международных стандартах (SQL/89 и SQL/92) к виду таких виртуальных таблиц предъявлялись чрезмерно строгие требования. Это показывают даже те простые примеры, которые приводились в начале данного раздела. И конечно, наличие таких ограничений в стандарте

языка приводило к тому, что в реализациях SQL появлялось много расширений, которые поддерживались только отдельными компаниями-производителями СУБД. Создается впечатление, что когда более десяти лет назад был инициирован проект нового стандарта SQL-3 (который в конце концов привел к появлению SQL:1999), разработчики находились в состоянии растерянности.

Кстати, одна из идей, включавшихся в ранние варианты проекта SQL-3, состояла в том, чтобы расширить определение представляемой таблицы средствами, позволяющими явно специфицировать действия, которые нужно предпринимать при выполнении над представлением операций `INSERT`, `UPDATE` и `DELETE`. Другими словами, предлагалось переложить решение проблемы на плечи пользователей СУБД. Конечно, это радикальный подход, но, с другой стороны, он мог бы привести к полной анархии.

Как можно заметить, в официально принятом стандарте SQL:1999 используется некоторый компромиссный подход. В стандарте не фиксируются жесткие правила, ограничивающие вид виртуальных таблиц, к которым применимы операции обновления. Вместо этого сформулирован ряд рекомендаций, которыми следует руководствоваться производителям СУБД. Нельзя утверждать, что такое решение является идеальным, но более удачного решения найти не удалось.

Операции обновления баз данных и механизм триггеров

Термин *триггер* в контексте реляционных баз данных был введен в обиход участниками проекта System R (разд. «Введение» лекции 11). В терминологии этого проекта триггером называлась хранимая в базе данных процедура, автоматически вызываемая СУБД при возникновении соответствующих условий. При определении триггера указывались два условия его применимости – общее условие (имя отношения и тип операции манипулирования данными) и конкретное условие (логическое выражение, построенное по правилам, близким к правилам ограничений целостности), а также действие, которое должно быть выполнено над БД при наличии условий применимости.

Конечно, термин *триггер* в данном контексте является жаргонным. Но, с другой стороны, он достаточно точно соответствует ситуации: для применения процедуры должны быть произведены «возбуждающие» ее действия. Как отмечалось в лекции 11, после завершения проекта System R на протяжении более десяти лет триггеры не поддерживались ни в одной коммерческой SQL-ориентированной СУБД. Но затем практически во всех ведущих СУБД механизм триггеров в том или ином виде был реализован.

В стандарте же языка SQL спецификации триггеров отсутствовали до принятия стандарта SQL:1999. По словам главного редактора стандартов SQL/92 и SQL:1999 Джима Мелтона, эта спецификация была уже полностью готова к моменту принятия SQL/92 и не вошла в текст стандарта только по причине ограниченности его объема. Однако, как мне кажется, этому препятствовали и расхождения в подходах, существовавшие между основными компаниями-производителями СУБД.

Заметим, что альтернативным термином по отношению к базам данных, содержащим триггерные процедуры, является термин *активная база данных*. Наверное, этот термин более точен, поскольку действительно речь идет о базах данных, содержащих процедуры, которые автоматически вызываются при срабатывании связанных с ними правил. Однако в обиходе пользователей SQL-ориентированных СУБД по-прежнему более распространен термин *триггер*.

Понятие триггера в SQL:1999

В языке обеспечиваются возможности определения триггеров, которые вызываются («срабатывают») при вставке одной или нескольких строк в указанную таблицу, при модификации одной или нескольких строк в указанной таблице или при удалении одной или нескольких строк из указанной таблицы. Вообще говоря, триггер может производить любое действие, необходимое для соответствующего приложения. Можно определить триггеры, срабатывающие по одному разу для операций `INSERT`, `UPDATE` или `DELETE`, но существует и возможность определения триггеров, вызываемых при вставке, модификации или удалении каждой отдельной строки. Таблица, с которой связывается определение триггера, называется *предметной таблицей (subject table)*, а оператор SQL, выполнение которого приводит к срабатыванию триггера, мы будем называть *инициирующим (triggering SQL statement)*.

Триггеры могут срабатывать *после* и *до* реального выполнения инициирующего оператора SQL. В теле триггера допускается доступ к значениям вставляемых, модифицируемых и удаляемых строк. В случае операции модификации возможен доступ к значениям строк до модификации и к значениям после модификации. В соответствии со стандартом SQL:1999 любой триггер ассоциируется только с одной базовой таблицей. Не допускается определение триггеров над представлениями*.

Можно придумать различные способы полезного применения механизма триггеров, но принято считать, что основными областями использования этого механизма являются следующие.

* Непонятно, откуда происходит это ограничение. Скорее всего, в будущих версиях стандарта оно будет снято.

- *Журнализация и аудит.* С помощью триггеров можно отслеживать изменения таблиц, для которых требуется поддержка повышенного уровня безопасности. Данные об изменении таблиц могут сохраняться в других таблицах и включать, например, идентификатор пользователя, от имени которого выполнялась операция обновления; временную метку операции обновления; сами обновляемые данные и т. д.
- *Согласование и очистка данных.* С любым простым оператором SQL, обновляющим некоторую таблицу, можно связать триггеры, производящие соответствующие обновления других таблиц. Например, с операцией вставки новой строки в таблицу EMP (прием на работу нового служащего) можно было связать триггер, модифицирующий значения столбцов DEPT_EMP_NO и DEPT_TOTAL_SAL* строки таблицы DEPT со значением столбца DEPT_NO, которое соответствует номеру отдела нового служащего.
- *Операции, не связанные с изменением базы данных.* В триггерах могут выполняться не только операции обновления базы данных. Стандарт SQL позволяет определять хранимые процедуры (которые могут вызываться из триггеров), посылающие электронную почту, печатающие документы и т. д.

Синтаксис определения триггеров и типы триггеров

Для более подробного обсуждения механизма триггеров в SQL:1999 необходимо ввести набор синтаксических правил:

```
trigger_definition ::=
     CREATE TRIGGER trigger_name
     { BEFORE | AFTER }
     { INSERT | DELETE | UPDATE [ OF column_commalist ] }
     ON table_name [ REFERENCING
old_or_new_values_alias_list ]
     triggered_action
triggered_action ::=
     [ FOR EACH { ROW | STATEMENT } ]
     [ WHEN left_paren conditional_expression right_paren ]
     triggered_SQL_statement
triggered_SQL_statement ::= SQL_procedure_statement
     | BEGIN ATOMIC
       SQL_procedure_statement_semicolonlist
       END
```

* В примерах этой лекции мы будем считать, что в столбце DEPT_TOTAL_SAL таблицы DEPT хранится суммарное значение заработной платы служащих соответствующего отдела.

```
old_or_new_values_alias ::= OLD [ ROW ] [ AS ] correlation_name
                          | NEW [ ROW ] [ AS ] correlation_name
                          | OLD TABLE [ AS ] identifier
                          | NEW TABLE [ AS ] identifier
```

Естественно, в языке имеется и конструкция, отменяющая определение триггера:

```
DROP TRIGGER trigger_name.
```

(Конструкция ALTER TRIGGER в языке SQL не поддерживается.)

Как мы видим, синтаксические правила допускают несколько разновидностей определения триггера. Кратко обсудим эти разновидности.

Триггеры BEFORE и AFTER

Если в определении триггера указано ключевое слово BEFORE, то триггер будет срабатывать непосредственно до выполнения операции обновления базовой таблицы соответствующим инициирующим оператором SQL. При задании ключевого слова AFTER триггер будет вызываться немедленно после выполнения инициирующего оператора.

Триггеры INSERT, UPDATE и DELETE

Выбор одного из этих ключевых слов при определении триггера указывает на природу события, которое должно приводить к срабатыванию триггера. При задании ключевого слова INSERT к срабатыванию триггера может привести только выполнение операции вставки строк в предметную таблицу. Если указываются ключевые слова UPDATE или DELETE, то число возможных событий, приводящих к срабатыванию триггера, возрастает. Кроме явных операций модификации строк предметной таблицы или удаления из нее строк к срабатыванию триггера могут привести ссылочные действия (см. раздел «Средства определения, изменения и ликвидации базовых таблиц» лекции 12).

Заметим, что в стандарте SQL:1999 отсутствует возможность определения триггеров, для которых событием было бы выполнение операции выборки из предметной таблицы. Разработчики стандарта сочли, что область применения триггеров такого рода чересчур узка (трудно придумать какое-либо применение, кроме как для журнализации и аудита).

Триггеры ROW и STATEMENT

Если в определении триггера присутствует конструкция FOR EACH ROW, то триггер будет вызываться для каждой строки предметной таблицы, обновляемой инициирующим SQL-оператором. Если же задано FOR EACH STATEMENT (или явная спецификация FOR EACH отсутствует), то триггер сработает один раз на всем протяжении процесса выполнения инициирующего SQL-оператора.

Раздел WHEN

Включение в определение триггера раздела WHEN с соответствующим условным выражением позволяет более точно специфицировать условие применимости триггера. Вычисление условного выражения производится над строками предметной таблицы, и триггер срабатывает только в том случае, когда значением условного выражения является *true*. Понятно, что виды и интерпретация логических выражений, допускаемых в разделе WHEN, различаются у триггеров с FOR EACH ROW и у триггеров с FOR EACH STATEMENT. В первом случае условное выражение вычисляется для одной строки, которая должна быть обновлена инициирующим SQL-оператором. Во втором – условное выражение вычисляется для всей предметной таблицы целиком и, по всей видимости, должно базироваться на «кванторных» предикатах. Следует также понимать, что вычисление условия раздела WHEN данного триггера производится только в том случае, если произошло событие срабатывания триггера.

Тело триггера

Операции, которые должны быть выполнены при срабатывании триггера, специфицируются в синтаксической конструкции triggered_SQL_statement (будем называть ее *инициируемым SQL-оператором*). Как видно из синтаксических правил, возможны два вида построения этой конструкции: в виде одиночного оператора SQL и в виде списка операторов со скобками BEGIN ATOMIC и END.

Недоумение читателей может вызвать неуточненная конструкция SQL_procedure_statement. Постараемся объяснить ее происхождение и смысл. Дело в том, что в стандарте SQL:1999 определено процедурное расширение SQL, называемое SQL/PSM (от *Persistent Stored Modules*). Это достаточно большой язык, который мы не будем подробно рассматривать в этом курсе лекций*. Тем не менее для понимания синтаксиса оп-

* Для читателей, которые имеют хотя бы минимальный опыт работы с продуктами компании Oracle, заметим, что во многих своих чертах SQL/PSM напоминает PL/SQL. Одной из причин, на основании которых мы отказались от описания SQL/PSM в этой книге, является то, что до сих пор (первый вариант стандарта SQL/PSM был опубликован в 1996 г.) нет ни одной реализации SQL, в которой этот стандарт был бы реализован полностью (точнее, ни одна такая реализация не известна автору).

ределения триггеров необходимо отметить, что: (a) SQL/PSM включает основные операторы SQL, связанные с обновлением данных; (b) язык является вычислительно полным, т. е. включает развитые средства вычислений; (c) в языке содержатся средства определения и вызова функций и процедур,* и (d) SQL/PSM содержит стандартный комплект управляющих конструкций – циклы, ветвления разных типов и т. д. Тем самым, SQL_procedure_statement – это любая процедура, определенная на языке SQL/PSM.** В частности, эта процедура может представлять собой оператор SQL обновления базы данных.

Обсудим теперь, откуда возникает потребность в составном инициируемом SQL-операторе. Дело в том, что на практике при определении триггеров в качестве SQL_procedure_statement чаще всего используются операторы SQL обновления базы данных. Иногда (и мы покажем это на примере) для корректного определения функциональности триггера одного оператора не хватает, а в SQL отсутствует возможность определения составных операторов. Поэтому допускается использование средств определения составных операторов, присутствующих в SQL/PSM (BEGIN ATOMIC и END).

Для иллюстрации случая, когда при определении триггера достаточно специфицировать один оператор SQL, приведем пример определения триггера, условием срабатывания которого является выполнение операции вставки новой строки в таблицу EMP (прием на работу нового служащего). Если значение столбца DEPT_NO в очередной вставляемой строке отлично от NULL, то триггер должным образом модифицирует значения столбцов DEPT_EMP_NO и DEPT_TOTAL_SAL строки таблицы DEPT со значением столбца DEPT_NO, которое соответствует номеру отдела нового сотрудника (**пример 17.10**):

```
CREATE TRIGGER DEPT_CORRECTION AFTER INSERT ON EMP
   FOR EACH ROW
   WHEN (EMP.DEPT_NO IS NOT NULL)
        UPDATE DEPT SET
        DEPT_EMP_NO = DEPT_EMP_NO + 1,
        DEPT_TOTAL_SAL = DEPT_TOTAL_SAL + EMP_SAL
        WHERE DEPT.DEPT_NO = EMP.DEPT_NO;
```

* Во многом на этих возможностях основываются механизмы SQL:1999, предназначенные для определения на уровне пользователя новых типов данных и их операций. Эта тематика также выходит за пределы данного курса (хотя мы немного затронем соответствующие вопросы в последней лекции этого курса).

** На самом деле, для написания процедур, функций и методов допускается использование не только языка SQL/PSM, но и традиционных языков программирования, для которых в стандарте определены правила связывания с SQL. В последней лекции курса мы немного затронем и эту тему.

Теперь предположим, что при увольнении служащего (удалении строки из таблицы EMP) мы хотим не только должным образом модифицировать таблицу DEPT, но и сохранять (с целью аудита) данные об уволенном служащем в таблице EMP_DISMISSED*:

EMP_DISMISSED:

EMP_NO : EMP_NO
EMP_NAME : VARCHAR
DEPT_NO : DEPT_NO

Определение соответствующего триггера могло бы выглядеть следующим образом (**пример 17.11**):

```
CREATE TRIGGER EMP_DISMISSION AFTER DELETE ON EMP
   FOR EACH ROW
      BEGIN ATOMIC
         INSERT INTO EMP_DISMISSED
            ROW (EMP.EMP_NO, EMP.EMP_NAME, EMP.DEPT_NO);
         UPDATE DEPT SET
            DEPT_EMP_NO = DEPT_EMP_NO - 1,
            DEPT_TOTAL_SAL = DEPT_TOTAL_SAL - EMP_SAL
         WHERE DEPT.DEPT_NO = EMP.DEPT_NO
   END;
```

Выполнение триггеров

При выполнении каждого триггера система устанавливает *контекст выполнения триггера*. Выполнение любого оператора SQL, обновляющего базовую таблицу базы данных, может привести к срабатыванию одного или нескольких триггеров, а выполнение операторов SQL, содержащихся в триггерах, может привести к обновлению других базовых таблиц. Эти «внутритриггерные» (инициируемые) операторы выполняются в контексте текущего триггера, но их выполнение может привести к срабатыванию других триггеров. Для каждого из «вторичных» триггеров образуется собственный контекст выполнения, позволяющий определить их действия точно и независимо от действий первого набора триггеров. Выполнение вторичных триггеров может привести к срабатыванию «третичных» триггеров и т. д. – допускается произвольная глубина вложенности. Для каждого триггера на каждом уровне образуется собственный контекст.

* Для упрощения будем считать, что идентификаторы уволенных служащих не используются повторно.

Контекст выполнения триггера всегда является атомарным, т. е. инициируемый SQL-оператор либо успешно завершается, либо результаты его действия гарантированно отсутствуют в базе данных.

Обсудим понятие контекста триггера немного более подробно. Предположим, что в нашей базе данных EMP-DEPT-PRO должно поддерживаться правило, в соответствии с которым каждый служащий, становящийся руководителем проекта, автоматически получает прибавку к заработной плате в 10 000 руб. (Для простоты будем считать, что снятие служащего с должности руководителя проекта не приводит к автоматическому изменению его зарплаты и что для каждого служащего, являющегося руководителем проекта, определен номер отдела, в котором он работает.) Тогда мы могли бы определить триггер CHANGE_MNG_NO следующим образом:

```
CREATE TRIGGER CHANGE_MNG_NO AFTER UPDATE OF PRO_MNG ON PRO
   FOR EACH ROW
      UPDATE EMP SET EMP_SAL = EMP_SAL + 10000.00
         WHERE EMP_NO = PRO_MNG;
```

Но очевидно, что для поддержания корректности данных в таблице DEPT нам требуется триггер, условием срабатывания которого было бы изменение значений столбца EMP_SAL в таблице EMP. Определим соответствующий триггер DEPT_CORRECTION_1:

```
CREATE TRIGGER DEPT_CORRECTION_1 AFTER UPDATE OF EMP_SAL ON EMP
   REFERENCING OLD ROW AS OLD_EMP NEW ROW AS NEW_EMP
      FOR EACH ROW
         UPDATE DEPT SET
            DEPT_TOTAL_SAL =
            DEPT_TOTAL_SAL + NEW_EMP.EMP_SAL - OLD_EMP.EMP_SAL
               WHERE EMP.DEPT_NO = DEPT.DEPT_NO;
```

Пусть теперь выполняется операция

```
UPDATE PRO SET PRO_MNG = 4455
   WHERE PRO_NO = 554;
```

Сразу после выполнения этой операции сработает триггер CHANGE_MNG_NO. Этот триггер будет выполняться в контексте, который мы для удобства назовем *контекстом CMN*. Заметим, что исходный оператор модификации в действительности изменяет только одну строку таблицы PRO, но триггеру CHANGE_MNG_NO это неизвестно, и он будет

работать так, как если бы изменялось произвольное число строк таблицы PRO.

Выполнение операции модификации таблицы EMP приведет к срабатыванию триггера DEPT_CORRECTION_1. В этот момент контекст CMN будет «упрятан в стек», образуется и станет активным контекст следующего триггера – *контекст DR1*. После завершения выполнения этого триггера контекст DR1 больше не требуется, и он ликвидируется, а из стека восстанавливается контекст CMN, в котором и будет завершено выполнение триггера CHANGE_MNG_NO.

Контекст выполнения триггера служит для того, чтобы обеспечить СУБД данными, необходимыми для корректного выполнения инициируемого оператора SQL. Эти данные представляют собой набор *изменений состояния*, где каждое изменение состояния описывает изменение данных в целевой таблице триггера. Изменение состояния включает следующие данные:

- триггерное событие – INSERT, UPDATE или DELETE;
- имя предметной таблицы триггера;
- имена столбцов предметной таблицы, специфицированных в определении триггера (только для триггеров по UPDATE);
- набор *переходов* (представление всех строк, вставляемых в предметную таблицу, модифицируемых в ней или удаляемых из нее), список всех триггеров уровня STATEMENT, уже выполненных в некотором (не обязательно активном) контексте выполнения, и список всех триггеров уровня ROW, уже выполненных в некотором (не обязательно активном) контексте выполнения, и строк, над которыми эти триггеры выполнялись.

Отслеживание уже выполненных триггеров ведется для предотвращения многократного выполнения одного и того же триггера в результате возникновения одного события, что могло бы потенциально привести к зацикливанию выполнения системы триггеров.

При создании контекста выполнения триггера его набор изменений состояния изначально пуст. В набор изменений состояния добавляется каждое встречающееся «новое» изменение состояния, в котором не дублируются триггерное событие существующего изменения состояния, имя предметной таблицы и имена столбцов предметной таблицы. Набор переходов каждого изменения состояния изначально пуст, и переходы добавляются при каждом обновлении предметной таблицы, ассоциированной с изменением состояния (включая обновления, производимые ссылочными действиями).

Возможности использования старых и новых значений

Мы уже продемонстрировали использование старых и новых значений в определении триггера DEPT_CORRECTION_1. Поскольку эта возможность является важной особенностью языка SQL, обсудим ее более подробно.

Сначала немного поговорим о синтаксисе. Итак, в определении триггера может присутствовать раздел REFERENCING old_or_new_values_alias_list, причем список определений псевдонимов может включать следующие элементы:

```
OLD [ ROW ] [ AS ] correlation_name
NEW [ ROW ] [ AS ] correlation_name
OLD TABLE [ AS ] identifier
NEW TABLE [ AS ] identifier
```

Каждая из этих конструкций может входить в список определений псевдонимов не более одного раза, и спецификации OLD ROW и NEW ROW могут присутствовать только в определении триггеров уровня ROW. Определяемые корреляционные имена и псевдонимы можно использовать внутри триггера для ссылок на значения предметной таблицы. Если определяется корреляционное имя для новых значений (NEW ROW) или псевдоним для нового содержимого таблицы (NEW TABLE), то эти имена можно использовать для ссылок на значения, которые будут существовать в предметной таблице *после* выполнения операций INSERT или UPDATE. Если же определяется корреляционное имя для старых значений (OLD ROW) или псевдоним для старого содержимого таблицы (OLD TABLE), то данные имена можно использовать для ссылок на значения, которые существовали в предметной таблице *до* выполнения операций UPDATE или DELETE. Конечно, нельзя использовать NEW ROW или NEW TABLE в триггерах DELETE, поскольку никакие новые значения не создаются. Аналогично, нельзя использовать OLD ROW или OLD TABLE в триггерах INSERT, поскольку никакие старые значения не существовали.

Таблицы, на которые указывают корреляционные имена или псевдонимы, называются *переходными*. Эти таблицы не сохраняются в базе данных долговременно; они создаются и уничтожаются динамически, по мере надобности в контексте выполнения триггера. В триггерах уровня ROW можно использовать корреляционное имя, определенное в конструкции OLD ROW, для ссылки на значения строки, удаляемой или модифицируемой инициирующим оператором, в том виде, в котором данная строка существовала в предметной таблице до того, как была удалена или модифицирована при выполнении инициирующего опера-

тора. В триггерах этого уровня можно также использовать псевдоним, определенный в конструкции OLD TABLE, для ссылки на любое значение переходной таблицы в том виде, в котором она находилась до удаления или модификации очередной строки при выполнении инициирующего оператора. Аналогично обстоят дела с использованием корреляционных имен и псевдонимов, определенных в конструкциях NEW ROW и NEW TABLE.

Для триггеров категории BEFORE имеется существенное ограничение: в них не разрешается использовать конструкции OLD TABLE и NEW TABLE, а внутритриггерный SQL-оператор не может производить какие-либо изменения в базе данных. Основанием для такого ограничения является то, что на переходные таблицы, порождаемые OLD TABLE и NEW TABLE, могут существенно влиять ссылочные действия, которые активизируются в результате изменений базы данных при выполнении внутритриггерного SQL-оператора. Поэтому значения строк в таких таблицах могут оказаться нестабильными и недостаточно предсказуемыми, если триггер срабатывает раньше действия триггерного оператора SQL.

Обработка нескольких триггеров, связанных с одной предметной таблицей

В SQL:1999 не запрещается определение нескольких триггеров, ассоциированных с одной предметной таблицей, относящихся к одной и той же категории (BEFORE или AFTER) и срабатывающих по одному и тому же событию. Понятно, что при возникновении условия срабатывания всех таких триггеров система должна выбрать порядок, в котором они будут выполняться.

Решение, принятое в SQL, является предельно простым, хотя и несколько странным. При определении каждого триггера фиксируется временная метка выполнения оператора CREATE TRIGGER, и все триггеры, ассоциированные с одной предметной таблицей, относящиеся к одной и той же категории (BEFORE или AFTER) и срабатывающие по одному и тому же событию, упорядочиваются в соответствии со своими временными метками. Тогда при возникновении условия срабатывания всех триггеров одной группы сначала выполняется первый триггер, затем второй и т. д. В стандарте не специфицируется точность временной метки, связываемой с триггером, и если в одной группе обнаруживаются два или более триггеров с неразличимыми временными метками, то порядок их выполнения должен определяться в реализации.

Подход к установлению порядка выполнения триггеров в соответствии с их временными метками может вызвать чисто практические трудности у пользователей SQL-ориентированных СУБД. Например, если в

ходе разработки приложения выяснится потребность в определении нового триггера, который должен выполняться раньше некоторого существующего триггера той же группы, то стандарт не может предложить ничего лучшего, кроме как уничтожить определения всех триггеров этой группы, а затем заново определить их в нужном порядке.

И еще одно интересное свойство триггеров в SQL:1999. Как уже говорилось ранее в этом разделе, каждый инициируемый SQL-оператор должен являться атомарным, т. е. если его выполнение завершается неуспешно, то в базе данных не должно остаться никаких следов подобного выполнения. Но в стандарте говорится больше: неуспешное выполнение хотя бы одного триггера из группы с одинаковым условием срабатывания должно приводить к отмене результатов выполнения инициируемых SQL-операторов всех триггеров этой группы, а также к отмене результатов выполнения самого инициирующего SQL-оператора*.

Триггеры и ссылочные действия

В разделе «Введение» лекции 12 мы достаточно подробно обсуждали механизм определения ссылочных действий, служащий для автоматической поддержки ссылочной целостности. Напомним, что ссылочные действия автоматически модифицируют значения внешнего ключа соответствующей таблицы при удалении или модификации строк таблицы, на которую указывают ссылки.

Конечно, ссылочные действия весьма напоминают триггеры, и в некоторых SQL-ориентированных СУБД они реализуются на основе общего механизма триггеров. Разработчики стандарта SQL:1999 считают этот подход неудачным, поскольку процедурная природа триггеров входит в противоречие с тщательно разработанной декларативной основой ссылочных ограничений целостности. Другими словами, спецификация ссылочной целостности, содержащаяся в стандарте, препятствует возможности встраивания в триггер упрощенного процедурного кода.

Однако даже в тех СУБД, где не смешиваются механизмы ссылочных действий и триггеров, неминуемо возникает взаимосвязь между ссылочными действиями, изменяющими некоторую таблицу, и триггерами, которые определены в этой таблице или также изменяют ее. В SQL:1999 эта взаимосвязь немного упрощается за счет того, что контроль всех ограничений целостности (включая ссылочные ограничения) и выполнение всех ссылочных действий должны производиться до срабатывания триггеров категории `AFTER`. Если выполняется некоторая операция обновле-

* Помимо прочего, этот факт означает, что определение в базе данных нового триггера может привести к неработоспособности существующих приложений, разработчики которых, вообще говоря, могут даже и не знать о появлении нового триггера.

ния таблицы *T*, то после ее выполнения и срабатывания всех ссылочных действий инициируются все триггеры, ассоциированные с таблицей *T* и видом произведенной операции, а также соответствующие триггеры, ассоциированные с любой таблицей, которая затрагивалась ссылочным действием, если в этой таблице была изменена хотя бы одна строка. Конечно, срабатывание триггера может привести к новым ссылочным действиям, которые повлекут за собой срабатывание других триггеров и т. д.*

В заключение этого раздела, посвященного механизму триггеров, заметим, что многие спецификации стандарта SQL:1999 выглядят недостаточно убедительными. По всей видимости, полезные на практике триггеры слишком сложны с точки зрения теории. Создается впечатление, что за годы, прошедшие после завершения проекта System R, с подобными трудностями так и не удалось справиться. Отсюда практический совет: если вам действительно требуется использование триггеров, обращайтесь к документации используемой вами СУБД, а если и документация не содержит ясных рекомендаций, прибегайте к осмысленным экспериментам.

Заключение

В этой лекции мы обсудили важные аспекты языка SQL, относящиеся к механизмам обновления данных. В первом разделе были рассмотрены операторы прямого SQL, предназначенные для вставки, модификации и удаления данных из существующих таблиц. Операторы `UPDATE` и `DELETE` этой категории иногда называют поисковыми, поскольку в них включаются условия на строки таблицы, которые должны быть модифицированы или удалены. В языке SQL определены также *позиционные* операторы модификации и удаления строк, а также *динамические позиционные* варианты данных операторов, но для их обсуждения требуется общее рассмотрение встраиваемого и динамического SQL, что выходит за рамки данного курса. На мой взгляд, поисковые версии операторов модификации и удаления хорошо характеризуют соответствующие возможности языка SQL. Кроме того, оператор `INSERT`, представленный в этой лекции, специфицирован в языке SQL только в таком варианте.

Второй раздел лекции посвящен обсуждению возможностей языка SQL, связанных с применимостью операций обновления базы данных через виртуальные таблицы, в том числе через представления. Мы рассмотрели ограничения языка SQL/92, накладываемые на виртуальные таблицы, к которым применимы операции обновления. Отмечалось, что эти

* Здесь я опять честно пересказал стандарт SQL:1999. И снова предложенное решение выглядит простым, но не убедительным.

ограничения являются достаточными, но не необходимыми для применения операций обновления. Был описан подход стандарта SQL:1999, где предлагаются рекомендации, но не требования, которых следует придерживаться реализациям SQL, чтобы соответствовать стандарту.

Наконец, в третьем разделе лекции рассматривался механизм триггеров. В первом подразделе упоминались основные понятия триггеров, которые были введены при выполнении проекта System R. Далее приводились основные синтаксические конструкции, предназначенные для определения триггеров, а также была описана их базовая семантика. В следующем подразделе обсуждались принципы выполнения триггеров, заложенные в стандарт SQL:1999. Наконец, в заключение раздела были рассмотрены имеющиеся взаимосвязи между ссылочными действиями и триггерами.

Один из основных выводов лекции состоит в том, что в стандарте SQL:1999 спецификации многих аспектов, относящихся к обновлению баз данных, обоснованы недостаточно убедительно. В ряде случаев разработчики стандарта ожидают улучшения спецификаций в следующих версиях стандарта.

Часть следующей лекции, относящаяся к средствам языка SQL, которые предназначены для управления транзакциями, также имеет непосредственное отношение к операторам обновления баз данных.

Лекция 18. Язык баз данных SQL: средства языка SQL для обеспечения авторизации доступа к данным, управления транзакциями, сессиями и подключениями

В этой лекции обсуждаются основные средства SQL:1999, предназначенные для регулирования работы с базами данных. Сначала рассматривается механизм авторизации доступа к объектам SQL-ориентированной базы данных, основанный на понятиях идентификатора пользователя, имени роли и привилегии доступа. Затем описываются особенности SQL-транзакций и основные языковые средства, воздействующие на поведение транзакций. Наконец, обсуждаются зафиксированные в стандарте SQL средства управления подключением к серверу баз данных.

Ключевые слова: мандатный (mandatory) способ защиты данных, дискреционный (discretionary) способ защиты данных, авторизация доступа к данным, идентификатор авторизации (authorization identifier, authID), функция CURRENT_USER, привилегии доступа, роль, идентификатор псевдопользователя PUBLIC, владелец объекта базы данных, привилегии уровня DBA (DataBase Administrator), идентификатор пользователя, имя роли, SQL-сессия, идентификатор пользователя SQL-сессии, имя роли SQL-сессии, текущий идентификатор пользователя SQL-сессии, функция SESSION_USER, текущее имя роли SQL-сессии, функция CURRENT_ROLE, оператор CREATE ROLE, раздел WITH ADMIN, оператор DROP ROLE, оператор GRANT, передача привилегий, раздел WITH GRANT OPTION, раздел GRANTED BY, правила определения привилегий над представлениями, передача ролей, оператор SET SESSION AUTHORIZATION, оператор SET ROLE, оператор REVOKE, аннулирование привилегий, режимы RESTRICT и CASCADE, раздел GRANT OPTION FOR, аннулирование ролей, ACID-транзакция, атомарность (Atomicy), согласованность (Consistency), изоляция (Isolation), долговечность (Durability), уровень изоляции (isolation level), режим доступа (access mode), размер области диагностики, оператор SET TRANSACTION, оператор START TRANSACTION, феномен «грязного» чтения (dirty read), уровень изоляции READ UNCOMMITTED, феномен неповторяемого чтения (unrepeatable read), уровень изоляции READ COMMITTED, феномен фантомов, уровень изоляции REPEATABLE READ, уровень изоляции SERIALIZABLE, оператор COMMIT, оператор ROLLBACK (ROLLBACK WORK), раздел AND [NO] CHAIN, оператор SET CONSTRAINTS, оператор SAVEPOINT, операция ROLLBACK TO SAVEPOINT, оператор RELEASE, установление соединений, оператор CONNECT, оператор SET CONNECTION, оператор DISCONNECT.

Введение

В этой лекции мы обсудим средства языка, которые касаются скорее администраторов баз данных, нежели конечных пользователей или программистов приложений. Но надо сказать, что любой квалифицированный пользователь SQL-ориентированной базы данных должен иметь представление об административных средствах SQL (тем более что средства управления транзакциями во многом затрагивают и его интересы).

Данная лекция включает материал, в меньшей степени концептуально связанный, чем это было в предыдущих лекциях курса, посвященных языку SQL. В первом из основных разделов лекции мы обсудим базовые идеи авторизации доступа к данным, заложенные в основу языка SQL. Метод авторизации доступа, используемый в SQL, относится к *мандатным (mandatory)* видам защиты данных. При этом подходе с каждым зарегистрированным в системе пользователем (*субъектом*) и каждым защищаемым объектом системы связывается *мандат*, определяющий действия, которые может выполнять данный субъект над данным объектом. В отличие от такого подхода, при применении *дискреционного (discretionary)* метода ограничения доступа с каждым из объектов системы связывается одна или несколько *категорий пользователей*, каждой из которых позволяются или запрещаются некоторые действия над объектом.

Следующий раздел посвящен фундаментальному в области баз данных (и не только) понятию *транзакции* – последовательности операций над базой данных (в общем случае включающей операции обновления базы данных), которая воспринимается системой как одна неделимая операция. При классическом подходе к управлению транзакциями следуют принципу `ACID (Atomicy, Consistency, Isolation, Durability)`. Этому принципу следовали и разработчики языка SQL. Однако понятие транзакции выходит далеко за пределы SQL; механизмы управления транзакциями составляют отдельную и большую исследовательскую область. В данной лекции мы не будем углубляться в технические детали управления транзакциями и ограничимся возможностями, заложенными в язык SQL.

Наконец, в последнем основном разделе лекции мы обсудим средства языка SQL, предназначенные для управления *сессиями* и *подключениями* пользовательских приложений. Подавляющее большинство реализаций языка SQL основывается на архитектурной модели *клиент-сервер*. Приложения обычно выполняются на *клиентской* аппаратуре, отделенной (по крайней мере, логически) от *серверной* аппаратуры, на которой работает собственно СУБД. Чтобы получить доступ к базе данных, приложение должно *подключиться* к серверу и образовать *сессию* в этом подключении. У приложения может одновременно существовать несколько подключений к разным серверам баз данных, но не более одной сессии в каждом подключении.

Поддержка авторизации доступа к данным в языке SQL

В общем случае база данных является слишком дорогостоящим предметом, чтобы можно было использовать ее в автономном режиме. Обычно с достаточно большой базой данных (параллельно или последовательно) работает много приложений и пользователей, и не для всех них было бы разумно обеспечивать равноправный доступ к хранящимся данным.

В языке SQL (SQL:1999) предусмотрены возможности контроля доступа к разным объектам базы данных, в том числе к следующим объектам:

- таблицам;
- столбцам таблиц;
- представлениям;
- доменам;
- наборам символов*;
- порядкам сортировки символов (collation);
- преобразованиям (translation);
- триггерам;
- подпрограммам, вызываемым из SQL;
- определенным пользователями типам.

В совокупности в SQL:1999 может поддерживаться девять видов защиты разных объектов в соответствии со следующими возможными действиями (см. табл. 18.1).

При разработке средств контроля доступа к объектам баз данных создатели SQL придерживались принципа сокрытия информации об объектах, содержащихся в схеме базы данных, от пользователей, которые лишены доступа к этим объектам. Другими словами, если некоторый пользователь не обладает, например, привилегией на просмотр таблицы `PRO`, то при выполнении операции `SELECT * FROM PRO` он получит такое же диагностическое сообщение, как если бы таблица `PRO` не существовала. Если бы в случае отсутствия этой таблицы и в случае отсутствия привилегии доступа выдавались разные диагностические сообщения, то непривилегированный пользователь получил бы данные о том, что интересующая его таблица существует, но он лишен доступа к ней.

В лекции 12 мы бегло упоминали, что в SQL-ориентированной системе каждому зарегистрированному в системе пользователю соответствует его уникальный идентификатор (в стандарте используется термин *идентификатор авторизации*, *authorization identifier – authID*). Как мы отмечали, в стандарте SQL:1999 не зафиксированы точные правила представления идентификатора пользователя, хотя обычно в реализациях

* Напомним, что в этом курсе мы не касаемся вопросов интернационализации и локализации языка SQL.

Таблица 18.1.

Вид защиты и соответствующее действие	*Название привилегии*	*Применимо к следующим объектам*
Просмотр	SELECT	Таблицы, столбцы, подпрограммы, вызываемые из SQL
Вставка	INSERT	Таблицы, столбцы
Модификация	UPDATE	Таблицы, столбцы
Удаление	DELETE	Таблицы
Ссылка	REFERENCES	Таблицы, столбцы
Использование	USAGE	Домены, определенные пользователями типы, наборы символов, порядки сортировки символов, преобразования
Инициирование	TRIGGER	Таблицы
Выполнение	EXECUTE	Подпрограммы, вызываемые из SQL
Подтипизация	UNDER	Структурные типы

SQL *ниладическая* функция CURRENT USER выдает текстовую строку, содержащую регистрационное имя пользователя, как оно сохраняется в файлах соответствующей операционной системы (ОС). Привилегии доступа к объектам базы данных могут предоставляться пользователям, представляемым своими идентификаторами, а также *ролям** (см. следующий подраздел), выполнение которых, в свою очередь, может предоставляться пользователям. Кроме того, в SQL поддерживается концепция псевдоидентификатора (или идентификатора псевдо) пользователя PUBLIC, который соответствует любому приложению или пользователю, зарегистрированному в системе баз данных. «Пользователю» PUBLIC могут предоставляться привилегии доступа к объектам базы данных, как и любому другому пользователю.

В модели контроля доступа SQL создатель любого объекта базы данных автоматически становится *владельцем* этого объекта. При этом владелец объекта может идентифицироваться либо своим идентификатором пользователя, либо именем своей роли. Вообще говоря, владелец объекта

* Как будет показано в следующем подразделе, термин *роль* в языке SQL полностью соответствует своему житейскому смыслу. И в мире баз данных люди большей частью играют чью-то роль, а не представляют себя лично.

обладает полным набором привилегий для выполнения действий над объектом (с одним исключением, которое мы обсудим в данном разделе позже). Владелец объекта, помимо прочего, обладает привилегией на передачу всех (или части) своих привилегий другим пользователям или ролям. В частности, владелец объекта может передать другим пользователям или ролям привилегию на передачу привилегий последующим пользователям или ролям (эти действия с передачей привилегии на передачу привилегий могут продолжаться рекурсивно).

Во многих реализациях поддерживаются привилегии уровня DBA (DataBase Administrator) для возможности выполнения операций DDL – Data Definition Language (CREATE, ALTER и DROP над объектами, входящими в схему базы данных). В стандарте SQL требуется лишь соблюдение следующих правил.

- Любые пользователь или его роль могут выполнять любые операции DDL внутри схемы, которой владеют*.
- Не допускается выполнение каких-либо операций DDL внутри схемы, которой не владеет пользователь или роль, пытающиеся выполнить соответствующую операцию.
- Эти правила не допускают исключений.

Пользователи и роли

Как говорилось в начале этого раздела, любой пользователь характеризуется своим *идентификатором авторизации* (authID). В стандарте ничего не говорится о том, что authID должен быть идентичен *регистрационному имени пользователя* в смысле операционной системы. Согласно стандарту SQL:1999, authID строится по тем же правилам, что и любой другой идентификатор, и может включать до 128 символов. Тем не менее во многих реализациях SQL, выполненных в среде ОС семейства UNIX, длина authID составляет не более восьми символов, как это свойственно ограничениям на длину регистрационного имени в этих ОС.

В стандарте языка SQL не специфицированы средства создания идентификаторов авторизации. Если говорить более точно, в стандарте не определяется какой-либо явный способ создания допустимых идентификаторов пользователей. Идентификатор авторизации может являться либо идентификатором пользователя, либо именем роли, а для создания ролей в SQL поддерживаются соответствующие средства (см. ниже). Но в соответствии с правилами стандарта SQL, все authID должны отслеживаться СУБД (имеются в виду все authID, для которых существует хотя

* В соответствии со стандартом, любые зарегистрированные в системе пользователь или роль автоматически являются владельцами части схемы базы данных, имена объектов которой начинаются с соответствующего идентификатора, за которым следует символ «.».

бы одна привилегия). И в стандарте поддерживаются точные правила порождения и распространения привилегий. Привилегии по отношению к объекту базы данных предоставляются системой владельцу схемы при создании объекта в этой схеме, и привилегии могут явно передаваться от имени одного `authID` другому `authID` при наличии у первого `authID` привилегии на передачу привилегий.

Итак, `authID` может являться либо идентификатором пользователя, либо идентификатором роли. Попробуем разобраться в сути термина *роль*. При работе с большими базами данных в крупных организациях часто сотни служащих производят над базой данных одни и те же операции. Конечно, для этого каждый из служащих должен быть зарегистрированным пользователем соответствующей системы баз данных, и тем самым, обладать собственным `authID`. Используя базовые средства авторизации доступа (зафиксированные в стандарте SQL/92), можно предоставить каждому пользователю группы одни и те же привилегии доступа к требуемым объектам базы данных. Но схема авторизации доступа при этом становится очень сложной*. В некотором смысле имя *роли* идентифицирует динамически образуемую группу пользователей системы баз данных, каждый из которых обладает, во-первых, привилегией на исполнение данной роли и, во-вторых, всеми привилегиями данной роли для доступа к объектам базы данных. Другими словами, наличие ролей упрощает построение и администрирование системы авторизации доступа. Проиллюстрируем это на рис. 18.1.

Каждая стрелка на рис. 18.1 соответствует мандату доступа (паре `<authID, набор_привилегий_доступа_к_ объекту_БД>`), который требуется сохранять в каталоге базы данных и проверять при попытке доступа от имени `authID`. Как видно, в случае (a) требуется сохранение и проверка $n*m$ мандатов, где n – число пользователей в группе, а m – число объектов базы данных, для которых пользователи группы должны иметь одни и те же привилегии. В случае (b) число требуемых для корректной работы мандатов равно лишь $n+m$, и схема авторизации резко упрощается.

Группы пользователей, объединенных одной ролью, являются динамическими, поскольку в SQL поддерживаются возможности предоставления пользователю привилегии на исполнение данной роли и лишения пользователя этой привилегии (см. ниже в этом разделе). Более того, имеются возможности предоставления заданной роли 1 всех или части привилегий другой роли 1. Естественно, что при этом привилегии изменяются у всех пользователей, которые могут исполнять роль 1.

* Для каждого объекта базы данных и для каждого пользователя, обладающего какими-либо привилегиями доступа к этому объекту, требуется хранить список его привилегий. Если учесть еще и возможность передачи привилегий от одного пользователя к другому, то образуется произвольно сложный граф, за которым трудно следить администраторам базы данных.

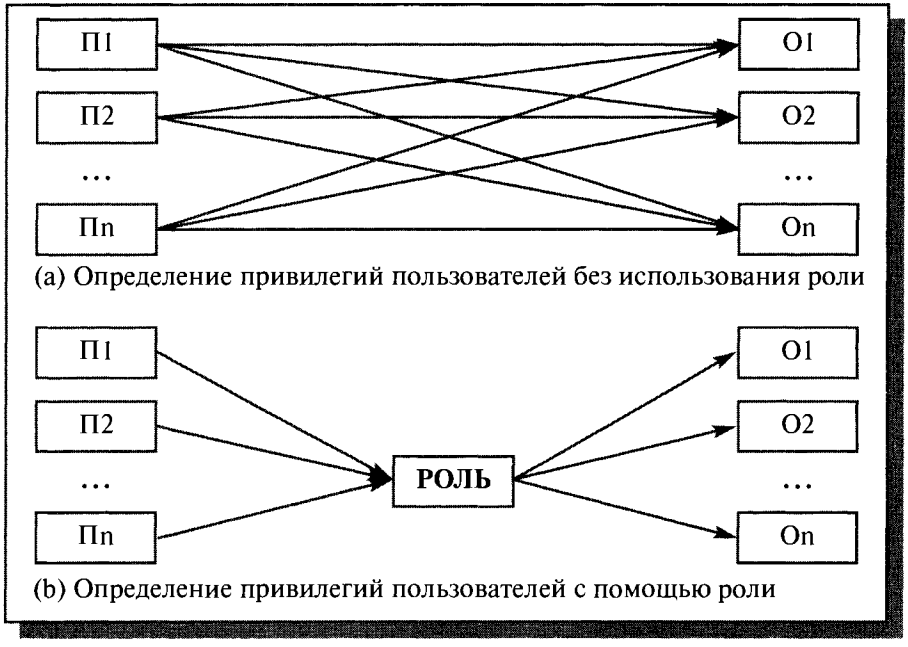


Рис. 18.1. Привилегии, пользователи и роли

Более того, имеются возможности предоставления заданной роли *A* всех или части привилегий другой роли *B*. Естественно, что при этом привилегии изменяются у всех пользователей, которые могут исполнять роль *A*.

Применение идентификаторов пользователей и имен ролей

В этом подразделе мы вынуждены использовать понятие SQL-сессии, которое более последовательно обсуждается в третьем основном разделе лекции. Как и в предыдущих лекциях, посвященных языку SQL, мы можем оправдать подобную нелогичность только рекурсивной природой стандарта SQL*.

Итак, в любой момент заданная SQL-сессия ассоциируется с идентификатором пользователя, называемым *идентификатором пользователя SQL-сессии* и с именем роли, называемым *именем роли SQL-сессии*. Почти всегда привилегии, связанные с этими идентификатором и именем, используются для определения допустимости выполнения различных операций во время данной сессии.

В стандарте не специфицированы все способы ассоциирования `authID` с SQL-сессией. Определено лишь то, что если сессия образуется

* Чтобы хотя бы немного облегчить чтение данного подраздела, забегая вперед, заметим, что понятия *сессии* и *подключения* относятся к сеансу работы клиентского приложения с некоторым сервером SQL-ориентированной базы данных.

с помощью оператора CONNECT (см. раздел «Подключения и сессии»), то authID указывается в качестве параметра соответствующей операции. Для реализаций SQL допускается, чтобы пользовательский идентификатор SQL-сессии совпадал с регистрационным именем пользователя с точки зрения операционной системы или являлся идентификатором, специально устанавливаемым специалистами организации, ответственными за обеспечение безопасности. Кроме того, допускается наличие в реализации оператора SET SESSION AUTHORIZATION, применение которого приводит к смене пользовательского идентификатора SQL-сессии. В начале SQL-сессии значение *текущего идентификатора пользователя* SQL-сессии совпадает со значением пользовательского идентификатора SQL-сессии, и такая ситуация сохраняется до тех пор, пока пользовательский идентификатор SQL-сессии не будет каким-либо образом изменен. Значение текущего пользовательского идентификатора SQL-сессии возвращается вызовом *ниладической* функции SESSION_USER (лекция 13).

Для каждой SQL-сессии существует также *текущее имя роли* (это имя можно получить путем вызова функции CURRENT_ROLE). Сразу после образования сессии текущему имени роли соответствует неопределенное значение, что трактуется как «роль для сессии не назначена». Имеется несколько способов подмены пользовательского идентификатора и/или имени роли SQL-сессии. При этом если задается идентификатор пользователя, то одновременно полагается, что неявно указывается имя роли, имеющее неопределенное значение. Если же задается имя роли, то, за несколькими исключениями, считается, что неявно указывается идентификатор пользователя, имеющий неопределенное значение. Более подробно мы обсудим подобные возможности ниже в этом разделе.

Если либо текущий идентификатор пользователя, либо текущее имя роли содержат неопределенное значение, то тот идентификатор или то имя, у которого значение не является неопределенным, используется в качестве *текущего* authID SQL-сессии. Если ни текущий идентификатор, ни текущее имя не содержат NULL, то текущим authID сессии служит текущий идентификатор пользователя.

Как и везде в этом курсе, мы опустим детали, относящиеся к особенностям авторизации при использовании встраиваемого и динамического SQL.

Создание и ликвидация ролей

Для создания новой роли используется оператор CREATE ROLE, определяемый следующим синтаксическим правилом:

```
CREATE ROLE role_name
   [ WITH ADMIN { CURRENT_USER | CURRENT_ROLE } ]
```

Имя создаваемой роли должно отличаться от любого идентификатора авторизации, уже определенного и сохраненного в базе данных. В случае успешного создания роли некоторый authID получает привилегию на исполнение данной роли. Если в операторе CREATE ROLE не содержится раздел WITH ADMIN, то привилегию на исполнение роли получает текущий идентификатор пользователя SQL-сессии, если значение этого идентификатора отлично от NULL; иначе привилегия на исполнение роли дается текущему имени роли сессии.

Если в состав оператора включается раздел WITH ADMIN, то можно выбрать, будет ли являться владельцем роли authID, соответствующий текущему идентификатору пользователя SQL-сессии, или authID, соответствующий текущему имени роли (при условии, что соответствующие текущий идентификатор или текущее имя не содержат NULL). Кроме того, включение этого раздела означает, что authID-владелец роли получает право на передачу привилегии исполнения данной роли другим authID.

В соответствии со стандартом SQL:1999, привилегии, требуемые для выполнения оператора CREATE ROLE, определяются в реализациях SQL. Например, в некоторых реализациях выполнение этой операции разрешается только администратору базы данных.

Существующую роль можно ликвидировать с помощью оператора

```
DROP ROLE role_name
```

Для выполнения этой операции требуется, чтобы текущий authID SQL-сессии прямо или косвенно (через цепочку ролей) являлся владельцем ликвидируемой роли. При ликвидации роли, прежде всего, изымается привилегия на ее исполнение у всех authID, которым данная привилегия была ранее передана.

Передача привилегий и ролей

Для передачи привилегий и ролей от одних authID другим поддерживается оператор GRANT, который мы обсудим отдельно для случаев передачи привилегий и передачи ролей.

Передача привилегий

В случае передачи привилегий используется следующий синтаксис оператора GRANT:

```
GRANT { ALL PRIVILEGES | privilege_commalist }
  ON privilege_object
```

```
TO { PUBLIC | authID_commalist } [ WITH GRANT OPTION ]
   [ GRANTED BY { CURRENT_USER | CURRENT_ROLE } ]
privilege ::= SELECT [ column_name_commalist ]
            | DELETE
            | INSERT [ column_name_commalist ]
            | UPDATE [ column_name_commalist ]
            | REFERENCES [ column_name_commalist ]
            | USAGE
            | TRIGGER
            | EXECUTE
privilege_object ::= [ TABLE ] table_name
                     | DOMAIN domain_name
                     | CHARACTER SET character_set_name
                     | COLLATION collation_name
                     | TRANSLATION translation_name
```

Поскольку authID может являться идентификатором пользователя или именем роли, привилегии могут передаваться от пользователей пользователям, от пользователей ролям, от ролей ролям и от ролей пользователям.

В списке привилегий можно использовать SELECT, DELETE, INSERT, UPDATE, REFERENCES и TRIGGER только в том случае, когда в качестве объекта привилегий указывается таблица. Соответственно, список привилегий может состоять из единственной привилегии USAGE только в том случае, когда объектом является домен, набор символов, порядок сортировки или трансляция. Если в списке привилегий указывается более одной привилегии, то они все передаются указанным authID, но для этого текущий authID SQL-сессии должен обладать привилегией на передачу привилегий.

Использование ключевого слова ALL PRIVILEGES вместо явного задания списка привилегий означает, что передаются все привилегии доступа к соответствующему объекту базы данных, которыми обладает текущий authID SQL-сессии.

Как показывает синтаксис, один оператор GRANT позволяет передавать привилегии доступа только к одному объекту, но в том случае, когда объектом является таблица, разные привилегии могут передаваться по отношению к одному и тому же набору столбцов или к разным наборам. Если при указании привилегий SELECT, DELETE, UPDATE и REFERENCES список имен столбцов не задается, передаются привилегии по отношению ко всем столбцам таблицы. Заметим, что эти привилегии касаются всех существующих столбов данной таблицы, а также всех столбцов, которые когда-либо будут к ней добавлены.

Включение в оператор необязательного раздела WITH GRANT OPTION означает, что получателям передаваемых привилегий дается также приви-

легия на дальнейшую передачу полученных привилегий, включая привилегию на передачу привилегий. Включение в оператор раздела GRANTED BY позволяет явно указать, передаются ли привилегии от имени текущего идентификатора пользователя или же текущего имени роли.

При проверке возможности выполнения операции в SQL-сессии учитываются привилегии текущего authID SQL-сессии, а также привилегии всех ролей, которые переданы данному authID. Поскольку этим ролям могли быть переданы другие роли, обладающие собственными привилегиями, анализ возможности выполнения операции является рекурсивной процедурой.

Если одна и та же привилегия передается более одного раза одному и тому же authID2 от имени одного и того же authID1, то возникает ситуация, называемая избыточной *дублирующей привилегией*. Эта ситуация не вызывает дополнительных проблем, поскольку избыточная передача привилегии игнорируется. Для аннулирования данной привилегии у authID2 от имени authID2 требуется выполнение всего лишь одной операции REVOKE (см. ниже в этом разделе). Если привилегия была один раз передана authID2 от имени authID1 вместе с привилегией на передачу этой привилегии (WITH GRANT OPTION), а в другой раз – без этой опции (порядок действий не является существенным), то authID2 обладает данной привилегией и привилегией на ее передачу.

Если предпринимается попытка передачи нескольких привилегий, но соответствующий authID не обладает ни одной из них, то фиксируется ошибка. Аналогично, если производится попытка передачи нескольких привилегий с передачей привилегии на передачу привилегий, но соответствующий authID не обладает привилегией WITH GRANT OPTION ни для одной из передаваемых привилегий, то фиксируется ошибка. Наконец, если производится попытка передачи нескольких привилегий с передачей привилегии на передачу привилегий и соответствующий authID обладает привилегией на передачу только части этих привилегий, то в результате выполнения операции вырабатывается предупреждение, но соответствующая часть привилегий передается с привилегией WITH GRANT OPTION.

Привилегии и представления

При определении представлений действуют специальные правила определения привилегий над этими представлениями. Если при создании обычных объектов базы данных, таких, как таблица или домен, текущий authID автоматически получает все возможные привилегии доступа к соответствующему объекту, включая привилегию на передачу привилегий, то для представлений ситуация иная. Поскольку создаваемое представление всегда основывается на одной или нескольких базовых таблицах (или

представлениях), привилегии, которые получает создатель представления, должны основываться на привилегиях, которыми располагает текущий authID по отношению к этим базовым таблицам или представлениям.

Например, чтобы операция создания представления была выполнена успешно, текущий authID должен обладать привилегией SELECT по отношению ко всем базовым таблицам и представлениям, на которых основывается новое представление. Тогда текущий authID автоматически получит привилегию SELECT для нового представления. Но текущий authID сможет передавать эту привилегию другим authID только тогда, когда обладает соответствующей привилегией для всех базовых таблиц и представлений, на которых основывается новое представление. Аналогичным образом на представление распространяются привилегии DELETE, INSERT, UPDATE и REFERENCES. Поскольку триггеры над представлениями создавать не разрешается, привилегия TRIGGER представлениям не передается.

Наконец, посмотрим, что происходит при смене привилегий владельца представления по отношению к таблицам, на которых основано это представление. Для простоты предположим, что представление V основано на базовой таблице *T*. Если во время создания *V* текущий authID (будущий владелец представления) обладал по отношению к *T* привилегиями SELECT и INSERT, то он будет обладать этими привилегиями и по отношению к *V**. Если впоследствии владелец представления получит по отношению к *T* дополнительные привилегии, то он (и все authID, которым передавались все привилегии – ALL PRIVILEGES для *V*) получит те же привилегии для *V*. Должно быть понятно, каким образом обобщается этот подход на случай, когда представление определяется над несколькими таблицами или представлениями.

Передача ролей

Для передачи ролей используется следующий вариант оператора GRANT:

```
GRANT role_name_commalist
TO { PUBLIC | authID_commalist } [ WITH ADMIN OPTION ]
   [ GRANTED BY { CURRENT_USER | CURRENT_ROLE } ]
```

Как показывает синтаксис, оператор позволяет передавать произвольное число ролей произвольному числу authID (которые могут представлять собой идентификаторы пользователей или имена ролей). Как и в случае передачи привилегий, от данного authID можно передавать

* Кстати, это один из тех случаев, когда *иметь право* не означает автоматически *иметь возможность реализации своего права*. SQL допускает, например, наличие привилегии INSERT для представления, к которому операция INSERT не применима.

только те роли, которые были получены этим authID с привилегией на дальнейшую передачу (WITH ADMIN OPTION). При включении в состав оператора GRANT раздела GRANTED BY можно явно указать, что роли передаются от имени текущего идентификатора пользователя или же текущего имени роли.

Изменение текущих идентификаторов пользователей и имен ролей

Как мы отмечали ранее в этом разделе, в SQL:1999 специфицированы некоторые операторы, позволяющие изменять текущий идентификатор пользователя и текущее имя роли SQL-сессии.

Оператор SET SESSION AUTHORIZATION

Для изменения текущего идентификатора пользователя SQL-сессии может использоваться оператор

```
SET SESSION AUTHORIZATION value_specification
```

Как указывалось в лекции 13, value_specification может быть либо литералом (в данном случае литералом типа символьных строк), либо вызовом *ниладической* функции, такой, как CURRENT_USER, SESSION_USER и т. д. Если указанная спецификация значения не соответствует требованиям, предъявляемым в реализации к представлению идентификатора пользователя, операция изменения текущего идентификатора пользователя аварийно завершается.

В стандарте также говорится, что если спецификация значения, заданная в операции, формально соответствует требованиям, предъявляемым к формату идентификатора пользователя конкретной системы, но в действительности не представляет *известный системе* идентификатор пользователя, то опять же фиксируется ошибка, и операция не выполняется. Допускается, чтобы в реализации принималось решение о смене идентификатора пользователя сессии одновременно с регистрацией нового идентификатора пользователя. Ограничения на регистрацию таким способом нового пользователя тоже определяются на уровне реализации. После успешного выполнения оператора SET SESSION AUTHORIZATION текущее имя роли соответствующей сессии принимает значение NULL, так что текущим authID этой сессии становится заданное значение идентификатора пользователя.

Опять по необходимости забегая вперед, заметим, что операцию смены текущего идентификатора пользователя SQL-сессии не разреша-

ется выполнять внутри какой-либо транзакции этой сессии. Иначе терялся бы смысл привилегий доступа, которыми руководствуется система при выполнении операций внутри транзакции.

Оператор SET ROLE

Для смены текущего имени роли SQL-сессии можно использовать оператор

```
SET ROLE { value_specification | NONE }
```

Ограничения на выполнение операции SET ROLE почти совпадают с определенными в стандарте ограничениями на выполнение операции SET SESSION AUTHORIZATION. Наиболее важные отличия состоят в том, что эту операцию от имени текущего authID сессии всегда разрешается выполнять для ролей, которые переданы «пользователю» PUBLIC или данному текущему authID, а также в том, что всегда разрешается применение конструкции SET ROLE NONE. Выполнение последней конструкции приводит к тому, что значение текущего имени роли сессии становится неопределенным.

Заметим, что при смене текущего имени роли SQL-сессии значение текущего пользовательского идентификатора сессии не меняется, так что вполне вероятно, что после выполнения операции и текущий идентификатор, и текущее имя роли будут иметь значения, отличные от неопределенного значения. И конечно, операция SET ROLE NONE будет выполнена успешно только в том случае, когда значение текущего пользовательского идентификатора не является неопределенным*.

Аннулирование привилегий и ролей

Если от имени некоторого authID некоторые привилегия или роли были переданы одному или нескольким другим authID, то впоследствии первый authID (в сессии, где этот authID является текущим) можно изъять, или *аннулировать*, переданные привилегии или роли путем применения оператора REVOKE. Как и в случае передачи привилегий и ролей, способы аннулирования привилегий и ролей похожи, но между ними имеются некоторые отличия. Поэтому мы снова обсудим эти способы в отдельности.

* Кстати, стандарт полностью отдает на волю реализации способ того, каким образом сделать неопределенным значение текущего пользовательского идентификатора SQL-сессии.

Аннулирование привилегий

Для аннулирования привилегий используется оператор REVOKE, определяемый следующим синтаксическим правилом:

```
REVOKE [ GRANT OPTION FOR] privilege_commalist
   ON privilege_object
FROM { PUBLIC | authID_commalist }
   [ GRANTED BY { CURRENT_USER | CURRENT_ROLE } ]
   { RESTRICT | CASCADE }
```

Синтаксис конструкций privilege и privilege_object такой же, как для оператора GRANT. Общий смысл операции должен быть понятен из синтаксиса: у указанных authID аннулируются указанные привилегии доступа к указанному объекту базы данных.

Первой важной особенностью оператора аннулирования привилегий является обязательность указания одного из ключевых слов RESTRICT или CASCADE. Если в операторе содержится RESTRICT, то при выполнении операции система проверит, не передавалась ли какая-либо из указанных привилегий каким-либо authID от того authID, у которого привилегия должна быть аннулирована (это вполне возможно, если ранее привилегия была передана с правом передачи). Если это действительно так, операция не выполняется; в противном случае указанные привилегии у указанных authID аннулируются. Иначе говоря, при наличии ключевого слова RESTRICT не допускается, например, ситуация, показанная на рис. 18.2.

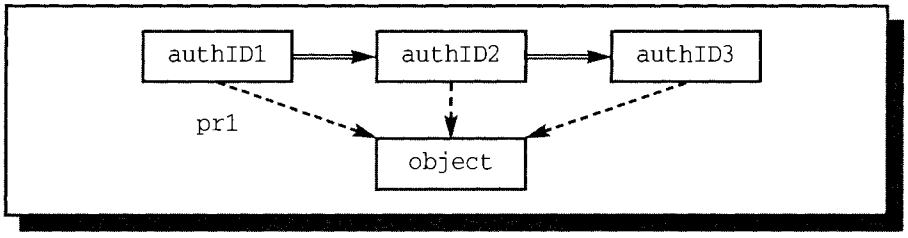


Рис. 18.2. Передача полученной привилегии

На этом рисунке authID1 является владельцем объекта базы данных с именем object и, следовательно, обладает всеми привилегиями над этим объектом. Пунктирной стрелкой обозначена одна из подобных привилегий pr1. От имени authID1 привилегия pr1 была передана authID2 вместе с привилегией на ее дальнейшую передачу. Наконец, от имени authID2 привилегия pr1 была передана authID3. Тогда операция аннули-

рования этой привилегии от имени authID1 у authID2 при наличии ключевого слова RESTRICT не будет выполнена успешно.

В той же ситуации привилегия была бы аннулирована для authID2 (и для authID3), если бы в операторе GRANT присутствовало ключевое слово CASCADE. В общем случае если выполняется операция REVOKE ... CASCADE, то указанные привилегии аннулируются у всех authID, прямо или косвенно (через промежуточные authID) получивших привилегии от текущего authID SQL-сессии, в которой выполняется данная операция.

Если в операторе содержится раздел GRANT OPTION FOR, но имеется ключевое слово RESTRICT, то указанные привилегии для указанных authID не аннулируются, но у указанных authID аннулируется привилегия передачи данных привилегий (операция должна успешно выполняться только при соблюдении обсуждавшихся ранее условий). Однако если в операторе одновременно содержатся и GRANT OPTION FOR, и CASCADE, то указанные привилегии аннулируются у всех authID, которые прямо или косвенно (через промежуточные authID) получили привилегии от текущего authID SQL-сессии, в которой выполняется данная операция.

Задание в операторе необязательного раздела GRANTED BY позволяет явно указать, что должно использоваться в качестве текущего authID — текущий пользовательский идентификатор или текущее имя роли SQL-сессии. Если раздел GRANTED BY в операторе REVOKE не содержится, то действия производятся от имени текущего authID SQL-сессии (о том, как он определяется, см. выше).

Если текущий (или указанный) authID не обладает ни одной из указанных в операторе REVOKE привилегий, то выполнение операции не производится (фиксируется ошибка). Если authID обладает некоторыми, но не всеми, привилегиями из числа указанных, то операция выполняется по отношению к этим некоторым привилегиям, но выдается предупреждение.

Возможны ситуации, когда у некоторого authID остается некоторая привилегия после выполнения операции аннулирования у этого authID этой привилегии. Одна из таких ситуаций проиллюстрирована на рис. 18.3.

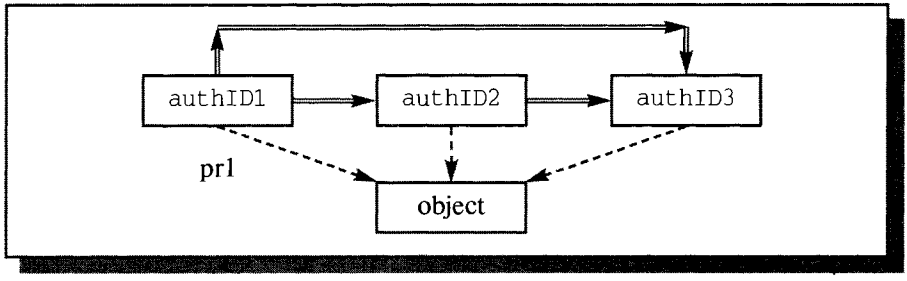


Рис.18.3. Косвенная и прямая передача привилегий

Здесь привилегия pr1 передана от authID1 к authID2 вместе с правом на дальнейшую передачу этой привилегии. Далее, привилегия pr1 передается от authID2 к authID3. И затем выполняется прямая передача привилегии от authID1 к authID3 (на самом деле, порядок таких действий не является существенным). Теперь предположим, что от имени authID1 выполняется операция

```
REVOKE pr1 ON object FROM authID2 CASCADED
```

В соответствии с правилами SQL:1999 после выполнения этой операции authID3 будет продолжать владеть привилегией pr1 по отношению к объекту object, поскольку получил данную привилегию *двумя разными способами*. Грубо говоря, операция REVOKE, выполняемая от имени authID1, выполняется только по тем путям графа идентификаторов авторизации и объектов базы данных, которые начинаются с узлов, соответствующих authID, указанных в разделе FROM этой операции.

Далее, напомним, что если при передаче от authID1 к authID2 привилегии на выполнение некоторых действий над некоторой таблицей *T* (например, UPDATE) явно не указывается список имен столбцов этой таблицы, то привилегия распространяется на все столбцы этой таблицы (включая столбцы, которые, возможно, еще будут созданы). Если действительно использовался такой способ передачи привилегий, то в дальнейшем можно аннулировать привилегию authID2 на модификацию отдельных (уже определенных) столбцов таблицы *T*, оставив привилегию на модификацию всех остальных столбцов (включая те, которые еще не созданы).

И последнее замечание. Если некоторая привилегия была передана *псевдо*authID PUBLIC, то, конечно, этой привилегией обладают все authID. Но нет возможности аннулировать такую привилегию у отдельно указываемого authID. Привилегия была передана всем, и аннулировать ее можно только сразу у всех.

Аннулирование ролей

Вариант оператора REVOKE, используемый для аннулирования ролей, выглядит следующим образом:

```
REVOKE [ ADMIN OPTION FOR ] role_name_commalist
FROM { PUBLIC | authID_commalist }
   [ GRANTED BY { CURRENT_USER | CURRENT_ROLE } ]
   { RESTRICT | CASCADE }
```

Действие операции аннулирования ролей очень похоже на действие операции аннулирования привилегий. Отличие состоит в том, что аннулируются не привилегии, а роли, а также в том, что для аннулирования привилегии на передачу роли используется раздел ADMIN OPTION FOR*.

Управление транзакциями в SQL

Область организации транзакций и управления ими настолько широка, что заслуживает отдельных книг и учебных курсов. В этом курсе мы ограничимся рассмотрением вопросов управления транзакциями в пределах стандарта языка SQL.

ACID-транзакция

В SQL:1999 поддерживается классическое понимание транзакции, характеризуемое аббревиатурой ACID (от Atomicy, Consistency, Isolation и Durability). В соответствии с этим понятием под транзакцией разумеется последовательность операций (например, над базой данных), обладающая следующими свойствами.

- *Атомарность* (*Atomicy*). Это свойство означает, что результаты всех операций, успешно выполненных в пределах транзакции, должны быть отражены в состоянии базы данных, либо в состоянии базы данных не должно быть отражено действие ни одной операции (конечно, здесь речь идет об операциях, изменяющих состояние базы данных). Свойство атомарности, которое часто называют свойством «все или ничего», позволяет относиться к транзакции, как к динамически образуемой составной операции над базой данных**.
- *Согласованность* (*Consistency*). В классическом смысле это свойство означает, что транзакция может быть успешно завершена с *фиксацией* результатов своих операций только в том случае, когда действия операций не нарушают целостность базы данных, т. е. удовлетворяют набору ограничений целостности, определенных для этой базы данных. В стандарте SQL:1999 это свойство расширяется тем, что во время выполнения транзакции разрешается устанавливать точки согласованности (см. ниже про точки сохранения) и явным образом проверять ограничения целостности***.

* В действительности, как видно из приведенных описаний, варианты операторов GRANT и REVOKE для привилегий и ролей настолько близки, что непонятно их синтаксическое разделение, которое, очевидно, усложняет реализацию. Как кажется, это разделение не обосновано в стандарте SQL:1999.

** В общем случае состав и порядок выполнения операций, выполняемых внутри транзакции, становится известным только на стадии выполнения.

*** Читателей может смутить параллельное использование терминов *согласованность* и

- *Изоляция* (`Isolation`). Требуется, чтобы две одновременно выполняемые транзакции* никоим образом не действовали одна на другую. Другими словами, результаты выполнения операций транзакции `T1` не должны быть видны никакой другой транзакции `T2` до тех пор, пока транзакция `T1` не завершится успешным образом.
- *Долговечность* (`Durability`). После успешного завершения транзакции все изменения, которые были внесены в состояние базы данных операциями этой транзакции, должны гарантированно сохраняться, даже в случае сбоев аппаратуры или программного обеспечения.

Порождение транзакций в SQL:1999

В соответствии со стандартом языка SQL:1999 транзакции** могут образовываться явным образом с использованием оператора `START TRANSACTION`, либо неявно, когда выполняется оператор, для которого требуется контекст транзакции, а этого контекста не существует. Например, операторы `SELECT`, `UPDATE` или `CREATE TABLE` могут выполняться только в контексте транзакции, а для выполнения оператора `CONNECT` (см. раздел «Подключения и сессии») такой контекст не требуется, и выполнение оператора `CONNECT` не приводит к неявному образованию транзакции. Для завершения транзакции должен быть явно использован один из двух операторов – `COMMIT` (требование завершить транзакцию с фиксацией ее результатов) или `ROLLBACK` (требование завершить транзакцию с удалением результатов всех выполненных операций из состояния базы данных***).

Установка характеристик транзакции

У каждой выполняемой транзакции имеются три характеристики, значения которых существенно влияют на действия системы при управлении транзакцией, – *уровень изоляции (isolation level)*, *режим доступа*

целостность. С точки зрения автора этого курса, в контексте баз данных эти два термина эквивалентны. Единственным критерием согласованности данных является их удовлетворение ограничениям целостности, т. е. база данных находится в согласованном состоянии тогда и только тогда, когда она находится в целостном состоянии.

* Здесь мы опять сталкиваемся с терминологической трудностью, существующей уже много лет. В англоязычной терминологии имеется замечательный термин *concurrent*, который соответствует как реально параллельному, так и квазипараллельному выполнению транзакций (или процессов). Русский эквивалент *одновременный* не совсем точно соответствует смыслу оригинала, но лучшего варианта пока нет.

** Правильнее было бы говорить *SQL-транзакции*, но в этом курсе мы не обсуждаем другие модели транзакций и поэтому будем использовать термин «транзакция» в смысле *SQL-транзакция*.

*** В русской терминологии для краткой характеристики этого действия часто используется не очень элегантный, но точно отражающий суть происходящего термин *откат* транзакции.

(access mode) и *размер области диагностики*. При неявном образовании транзакции эти характеристики устанавливаются по умолчанию: транзакция получает максимальный уровень изоляции от одновременно выполняемых транзакций; режим доступа, позволяющий выполнять и операции выборки, и операции обновления базы данных; и назначаемый по умолчанию размер области диагностики.

Если значения характеристик транзакции, устанавливаемых по умолчанию, в некотором случае не являются пригодными, то до выполнения оператора, неявно инициирующего транзакцию, можно явно установить характеристики данной транзакции с использованием оператора SET TRANSACTION. Этот оператор определяется следующими синтаксическими правилами:

```
SET TRANSACTION mode_commalist
mode ::= isolation_level
       | access_mode
       | diagnostics_size
isolation_level ::= READ UNCOMMITED
                  | READ COMITTED
                  | REPEATABLE READ
                  | SERIALIZABLE
access_mode ::= READ ONLY
              | READ WRITE
diagnostics_size ::= DIAGNOSTIC SIZE value_specification
```

Операцию установки характеристик транзакции нельзя выполнять в контексте какой-либо активной транзакции. Выполнение операции допустимо только до образования первой транзакции SQL-сессии или между последовательно выполняемыми транзакциями этой сессии. В одном операторе SET TRANSACTION можно задать только по одному значению каждой из трех характеристик, но допускается последовательное выполнение нескольких таких операций с разными операндами.

Как видно из синтаксических правил, у характеристики *режим доступа* может быть указано одно из двух значений – READ ONLY или READ WRITE. Если устанавливается режим READ ONLY, то в транзакции нельзя будет выполнять никакие операции, изменяющие базу данных, в том числе операции обновления таблиц и определения новых объектов базы данных. Если режим доступа явно не указывается, по умолчанию принимается характеристика READ WRITE, если только в качестве значения характеристики *уровень изоляции* не указывается READ UNCOMITTED (в этом случае устанавливается режим доступа READ ONLY).

Если указывается размер области диагностики, то после ключевых слов DIAGNOSTIC SIZE должен следовать целочисленный литерал, определяющий число диагностических элементов, которые должны разместиться в области диагностики (число исключительных ситуаций, предупреждений, сообщений об отсутствии данных и об успешном выполнении, которые будут вырабатываться при выполнении операторов внутри будущей транзакции). Если размер области диагностики явно не указывается, то решение о размере этой области принимается в реализации*.

Уровни изоляции будут подробно обсуждаться ниже, но здесь мы заметим, что если значение уровня изоляции явно не задано, то по умолчанию принимается уровень изоляции SERIALIZABLE. Кроме того, еще раз обратим внимание читателей, что одновременное задание уровня изоляции READ UNCOMITTED и режима доступа READ WRITE не допускается.

Еще одна интересная деталь оператора установки характеристик транзакции состоит в том, что выполнение каждого следующего оператора SET TRANSACTION полностью перекрывает эффект выполнения предыдущего такого оператора. В частности, если в предыдущем операторе явно задавалось значение некоторой характеристики, а в следующем это значение принимается по умолчанию, то именно значение по умолчанию будет являться значением характеристики транзакции.

Явная инициация транзакции

Для явного образования транзакции поддерживается оператор START TRANSACTION, определяемый следующими синтаксическими правилами:

```
START TRANSACTION mode_commalist
```

Этот оператор очень похож на SET TRANSACTION. Единственное (хотя и очень существенное) отличие состоит в том, что выполнение оператора START TRANSACTION приводит не только к установке характеристик транзакции, но и к реальной инициации транзакции.

Уровни изоляции SQL-транзакции

В стандарте SQL:1999 уровни изоляции определяются на основе нескольких феноменов, которые могут возникать при выполнении транзакций**.

* В этом курсе мы не будем более подробно обсуждать способы получения и обработки диагностических сообщений, поскольку это потребовало бы привлечения слишком большого числа технических деталей, не слишком существенных для общего понимания языка.

** В действительности, этот подход был введен еще в проекте System R.

Феномен «грязного» чтения (dirty read)

Этому феномену подвержены транзакции, в которых допускается возможность видеть изменения объектов базы данных, производимые другими одновременно выполняемыми и еще не зафиксированными транзакциями. Простой пример феномена *«грязного»* чтения показан на рис. 18.4.

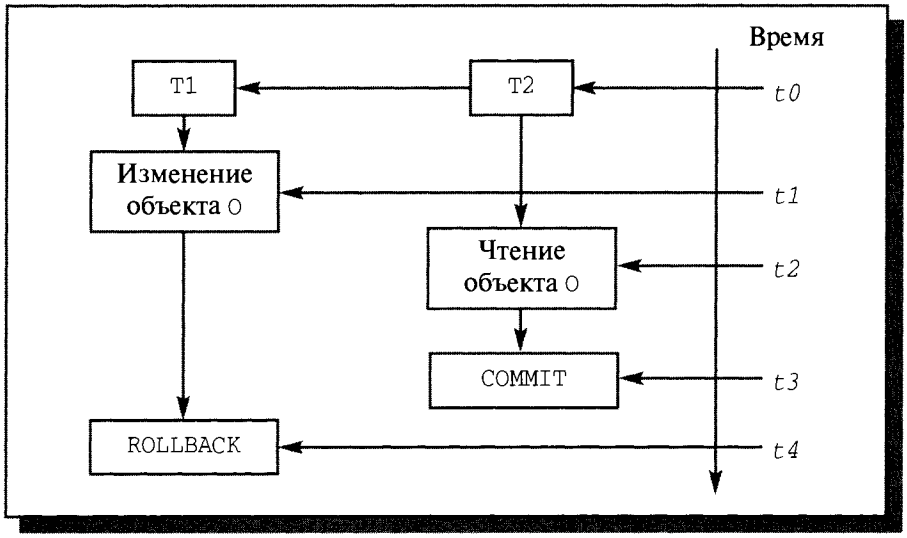


Рис. 18.4. Феномен «грязного» чтения

На этом рисунке показано, что в момент времени *t0* были образованы две транзакции T1 и T2. В момент времени *t1* транзакция T1 успешно выполняет операцию модификации некоторого объекта базы данных O. В момент времени *t2* (*t2>t1*) транзакция T2 читает объект O, после чего успешно завершается в момент времени *t3*. Транзакция же T1 завершается в момент времени *t4* (*t4>t3*), причем в ней выполняется оператор ROLLBACK, что приводит к ликвидации в базе данных последствий изменения объекта O. В результате оказывается, что в транзакции T2 обрабатывались данные, которые реально не существуют в базе данных (отсюда и термин *«грязные» данные*).

В SQL феномен «грязного» чтения может наблюдаться у транзакций, выполняемых на уровне изоляции READ UNCOMMITTED. Рекомендуется использовать этот уровень изоляции только в тех транзакциях, для выполнения функций которых точные данные не обязательны (например, в транзакциях, производящих статистическую обработку).

Феномен неповторяемого чтения (unrepeatable read)

Этому феномену подвержены транзакции, читающие некоторые объекты базы данных и допускающие изменения уже прочитанных объектов другими транзакциями. Пример феномена неповторяемого чтения показан на рис. 18.5.

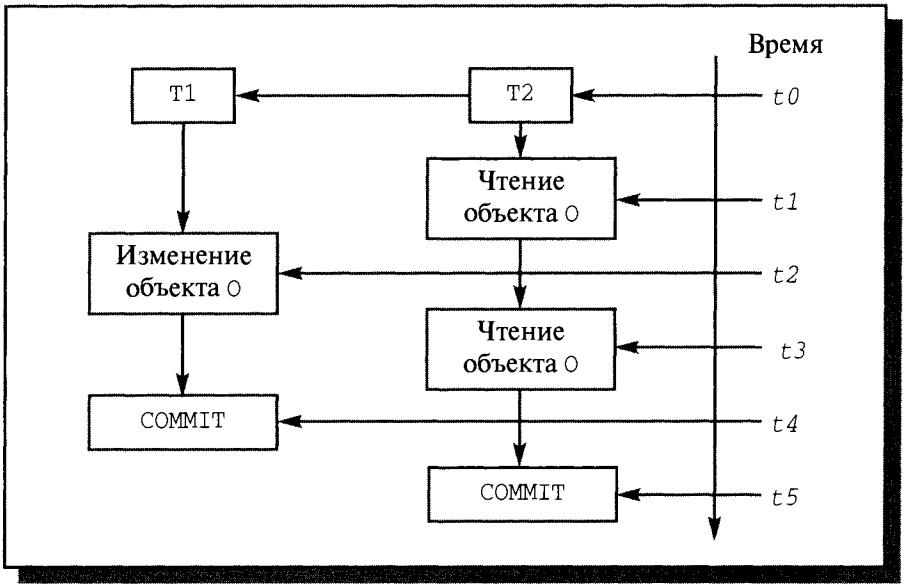


Рис. 18.5. Феномен неповторяемого чтения

На этом рисунке показано, что в момент времени *t0* были образованы две транзакции T1 и T2. В момент времени *t1* транзакция T2 выполняет операцию чтения некоторого объекта базы данных O (например, производит выборку строки из таблицы с указанием значения первичного ключа). В момент времени *t2* (*t2>t1*) транзакция T1 изменяет объект O (модифицирует или даже удаляет). В момент времени *t3* (*t3>t2*) транзакция T2 повторно считывает объект O и обнаруживает, что он изменился или вовсе отсутствует. Другими словами, в транзакции T2 повторное выполнение выборки объекта базы данных O дало результат, отличный от результата первого выполнения (отсюда и происходит термин «*неповторяемое чтение*»).

В SQL феномен неповторяемого чтения может наблюдаться у транзакций, выполняемых на уровне изоляции READ COMMITTED (этот уровень изоляции, как показывает его название, гарантирует отсутствие феномена «грязного» чтения).

Феномен фантомов

Этому феномену подвержены транзакции, производящие выборку строк и таблиц базы данных и допускающие добавление к данным таблицам другими транзакциями строк, которые удовлетворяют условию выборки. Пример феномена фантомов показан на рис. 18.6.

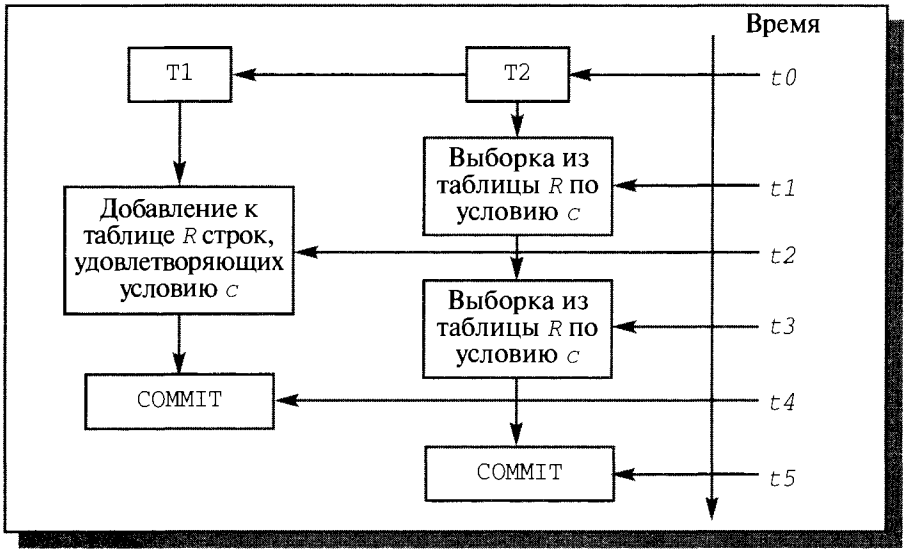


Рис. 18.6. Феномен фантомов

На этом рисунке показано, что в момент времени *t0* были образованы две транзакции, T1 и T2. В момент времени *t1* транзакция T2 выполняет операцию выборки строк из таблицы *R* по условию *c*. В момент времени *t2* (*t2>t1*) транзакция T1 выполняет над таблицей *R* операцию обновления (вставки или модификации строк), в результате которой в таблице *R* появляются дополнительные строки, удовлетворяющие условию *c*. В момент времени *t3* (*t3>t2*) транзакция T2 повторно выполняет операцию выборки строк из таблицы *R* по условию *c* и обнаруживает наличие в результате дополнительных *фантомных* строк.

В SQL феномен фантомов может наблюдаться у транзакций, выполняемых на уровне изоляции REPEATABLE READ (этот уровень изоляции, как показывает его название, гарантирует отсутствие феномена неповторяемого чтения).

Наконец, для транзакций, выполняемых на уровне изоляции SERIALIZABLE, невозможно и проявление феномена фантомов. Термин *serializable (сериализуемый)* используется по той причине, что при работе

на данном уровне изоляции суммарный эффект выполнения набора транзакций {T1, T2, ..., Tn} идентичен эффекту некоторого последовательного выполнения этих транзакций. Это означает предельную изолированность транзакций. Общая картина взаимосвязи уровней изоляции и феноменов транзакций показана в таблице 18.2.

Таб. 18.2. Уровни изоляции и феномены

Уровень	«Грязное» чтение	Неповторяемое чтение	Фантомы
READ UNCOMMITTED	Возможно	Возможно	Возможны
READ COMMITTED	Невозможно	Возможно	Возможны
REPEATABLE READ	Невозможно	Невозможно	Возможны
SERIALIZABLE	Невозможно	Невозможно	Невозможны

Завершение транзакций

Как мы отмечали в начале этого раздела, транзакции могут инициироваться как явным способом (с помощью оператора START TRANSACTION), так и неявно, при выполнении первого оператора, требующего наличия контекста транзакции. Для завершения транзакции всегда* требуется выполнение одного из двух операторов COMMIT (фиксация транзакции) или ROLLBACK (откат транзакции), которые имеют следующий синтаксис:

```
COMMIT [ WORK ] [ AND [ NO ] CHAIN ]
ROLLBACK [ WORK ] [ AND [ NO ] CHAIN ]
[ TO SAVEPOINT savepoint_name ]
```

При желании завершить транзакцию таким образом, чтобы все произведенные ею изменения были навсегда сохранены в базе данных, следует завершать транзакцию оператором COMMIT (как видно из синтаксиса, допускается эквивалентный вид COMMIT WORK). Если требуется завершить транзакцию с аннулированием всех произведенных изменений, то нужно использовать оператор ROLLBACK (ROLLBACK WORK).

Заметим, что и операция фиксации транзакции, и операция отката являются достаточно сложными и выполняются не мгновенно. Поэтому в ходе выполнения этих операций, вообще говоря, может произойти ава-

* Правильнее было бы сказать *почти всегда*, поскольку в SQL предусматривается особый способ терминации транзакций, инициированных программными агентами. Но в данном курсе мы этого не касаемся.

рийный отказ системы. Естественно (хотя в этом курсе мы не обсуждаем технические детали возможных реализаций), база данных будет восстановлена в свое последнее согласованное состояние, но ситуации прерванного выполнения операции фиксации и операции отката коренным образом различаются. Оператор COMMIT считается безусловно выполненным только в том случае, когда сервер баз данных подтвердил это после выполнения всех действий, требуемых для фиксации транзакции. Аварийная ситуация во время выполнения операции ROLLBACK ничем не отличается от аварийной ситуации, возникшей в процессе выполнения транзакции. В этом случае (при восстановлении базы данных) прерванная транзакция считается незафиксированной (что так и есть), и все ее изменения автоматически удаляются из состояния базы данных. Поэтому окончательный результат выполнения операции фиксации транзакции, прерванной аварийным отказом системы, эквивалентен успешному выполнению операции отката транзакции*.

Синтаксис обоих операторов показывает, что в каждом из них может содержаться раздел AND [NO] CHAIN. Постараемся кратко пояснить смысл этого раздела (не вдаваясь в детали, поскольку стандарт SQL:1999 оставляет окончательное решение за реализацией).

Операции образования и завершения транзакции являются достаточно дорогостоящими. В особенности это касается операции завершения транзакции, при выполнении которой необходимо выполнить обмены с внешней памятью. С другой стороны, использование долговременных транзакций чревато снижением уровня параллелизма в системе, а также повышает риск утраты результатов транзакции в результате системного отказа.

Поэтому часто используется компромиссный вариант, при котором действия операторов COMMIT или ROLLBACK приводят не только к завершению текущей транзакции, но и к образованию новой транзакции**. Именно эту возможность поддерживает раздел AND [NO] CHAIN операторов COMMIT и ROLLBACK. Если такой раздел отсутствует в операторе завершения транзакции, то подразумевается наличие раздела AND NO CHAIN, и новая транзакция не образуется. Если же раздел AND CHAIN присутствует, то немедленно после завершения выполнения COMMIT или ROLLBACK текущей транзакции образуется новая транзакция, наследующая все характеристики завершенной транзакции.

Семантику раздела TO SAVEPOINT мы поясним немного позже.

* Возможно, некоторым читателям эти рассуждения покажутся несколько расплывчатыми, но в действительности за ними стоит развитая техника журнализации и восстановления, применяемая во всех развитых SQL-ориентированных СУБД.

** При этом экономятся хотя бы ресурсы, требуемые для создания транзакций. Иногда такие цепочки транзакций поэтически называют *сагами*: если вы когда-нибудь пробовали писать саги, то должны были почувствовать, что это проще, чем писать отдельные сказания.

Транзакции и ограничения целостности

Материал этого подраздела уже излагался в лекции 13, но там это делалось в контексте определений ограничений целостности. Для полноты картины мы воспроизведем часть этого материала в контексте управления транзакциями.

Итак, любое ограничение целостности обладает атрибутом, определяющим время проверки данного ограничения. Этот атрибут может иметь значения DEFERRABLE (отложенная проверка) или NOT DEFERRABLE (немедленная проверка). Чтобы данное ограничение целостности могло когда-либо обладать свойством отложенной проверки, нужно, чтобы в определении такого ограничения присутствовали ключевые слова INITIALLY DEFERRED или INITIALLY IMMEDIATE. В любом случае, в каждый момент времени выполнения транзакции любое ограничение целостности находится в одном из двух состояний – *отложенная проверка* или *немедленная проверка*. Если начальным состоянием ограничения является INITIALLY DEFERRED, то в начале любой транзакции его текущим состоянием будет *отложенная проверка*. Аналогично для ограничений с начальным состоянием INITIALLY IMMEDIATE.

Любое ограничение, находящееся в состоянии *немедленной проверки*, всегда проверяется в конце выполнения любого оператора SQL*. Немедленно проверяются и те ограничения, которые были определены как NOT DEFERRABLE, но для которых впоследствии был установлен режим немедленной проверки. Однако если текущим состоянием ограничения является *отложенная проверка*, оно будет проверяться только тогда, когда перейдет в состояние *немедленной проверки*. Это делается неявно при выполнении оператора COMMIT или явно при выполнении оператора SET CONSTRAINTS. Этот оператор имеет следующий синтаксис:

```
SET CONSTRAINTS { ALL | constraint_name_commalist}
   { DEFERRED | IMMEDIATE }
```

Ключевое слово ALL является сокращенной формой задания списка имен всех ограничений целостности, определенных в базе данных, которые специфицированы с указанием ключевого слова DEFERRABLE. Если список имен ограничений задается явно, то все входящие в него имена должны соответствовать ограничениям, определенным с указанием ключевого слова DEFERRABLE.

При попытке фиксации транзакции, для которой имеются одно или несколько ограничений целостности, текущим режимом которых является

* Естественно, на практике проверяются только те ограничения, которые могут быть потенциально нарушены в результате выполнения соответствующего оператора.

отложенная проверка, система (ненадолго, поскольку транзакция скоро тем или иным способом завершится) устанавливает для всех этих ограничений режим немедленной проверки и проверяет ограничения. Если какое-либо из ограничений нарушается, то операция `COMMIT` трактуется как операция `ROLLBACK`, и пользователю (или приложению) сообщается, что возникла ошибка. Избежать этой неприятной ситуации можно явным выполнением оператора `SET CONSTRAINTS ALL IMMEDIATE` до фиксации транзакции, для которой имеются `DEFERRABLE` ограничения, текущим режимом которых является *отложенная проверка*.

Точки сохранения

Как мы уже отмечали, использование долговременных транзакций повышает риск полного аннулирования результатов транзакции по причине нарушения ограничений с отложенной проверкой при выполнении каких-либо экспериментальных (недостаточно проверенных) операций. Конечно, теоретически можно было бы оформлять выполнение каждой такой подозрительной операции в виде отдельной транзакции, но это часто противоречит общей логике приложения, когда последовательность действий должна быть атомарной.

Частичное решение этой проблемы предоставляет механизм *точек сохранения* (*savepoint*) SQL:1999. Точка сохранения представляет собой своего рода пометку в последовательности операций транзакции, которую в дальнейшем можно использовать для частичного отката транзакции с сохранением жизнеспособности транзакции и результатов операций, выполненных в транзакции до точки сохранения. Пример использования точки сохранения показан на рис. 18.7.

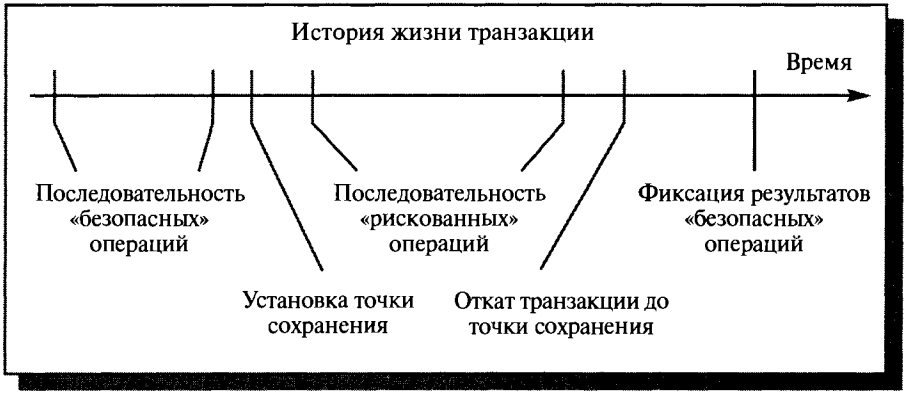


Рис. 18.7. Пример транзакции с точкой сохранения

На этом рисунке после выполнения последовательности проверенных «безопасных» операций, которые, по мнению пользователя, не могут нарушить ограничения целостности с отложенной проверкой, устанавливается точка сохранения. За этой точкой следует серия «рискованных» операций. Если по каким-то причинам (например, путем немедленной проверки отложенных ограничений) затем принимается решение о нецелесообразности фиксации результатов данных операций, то выполняется частичный откат транзакции к точке сохранения, а затем фиксируются результаты безопасных операций.

Допускается установка в одной транзакции нескольких последовательных точек сохранения. При установке каждой точки сохранения ей назначается некоторое (локальное в пределах транзакции) имя, которое в дальнейшем может использоваться в операции ROLLBACK для задания точки частичного отката транзакции (см. выше синтаксис оператора ROLLBACK). Если последовательно устанавливаются две точки сохранения SP1 и SP2 и затем выполняется операция ROLLBACK TO SAVEPOINT SP1, то восстановление производится до SP1 (через SP2), и точка сохранения SP2 «забывается»*.

Для установления точки сохранения используется оператор SAVEPOINT с очевидным синтаксисом

```
SAVEPOINT savepoint_name
```

Можно также отказаться от ранее установленной точки сохранения, удалив ее из контекста транзакции. Для этого предназначен оператор RELEASE, синтаксис которого также очевиден:

```
RELEASE SAVEPOINT savepoint_name
```

После выполнения этой операции в данной транзакции невозможно выполнять какие-либо другие операции над точкой сохранения с данным именем, пока не будет образована другая одноименная точка сохранения.

* Обратите внимание, что оператор ROLLBACK TO SAVEPOINT savepoint_name, хотя и синтаксически схож с «обычным» оператором ROLLBACK, принципиально отличается по своей семантике. Откат транзакции до указанной точки сохранения не означает завершения транзакции. Видимо, из соображений здравого смысла, следовало бы ввести в язык SQL операцию ABORT TRANSACTION для аварийного завершения транзакции с ликвидацией всех последствий ее операций в базы данных и, отдельно, операцию ROLLBACK TO, в которой всегда явно указывалось бы, до какого уровня требуется откат. Кстати, заметим, что комбинация

```
ROLLBACK AND CHAIN TO SAVEPOINT savepoint_name
```

является недопустимой (поскольку текущая транзакция не завершается).

Подключения и сессии

Может показаться странным, что мы оставили на конец этой лекции материал, который, казалось бы, необходимо знать, чтобы иметь возможность приступить к работе с какой-либо из современных систем баз данных. Объяснение очень простое. Чем ниже уровень средств языка SQL, чем ближе эти средства соприкасаются с индивидуальными особенностями реализаций, тем менее точен и конкретен стандарт SQL. А в данном разделе речь идет о средствах, реализация которых в СУБД разных поставщиков обладает очень большой спецификой.

Сильно упрощая текущую ситуацию, можно сказать, что практически все современные продукты управления SQL-ориентированными базами данных основаны на архитектуре «клиент-сервер». Принципиальная схема клиент-серверной организации показана на рис. 18.8.

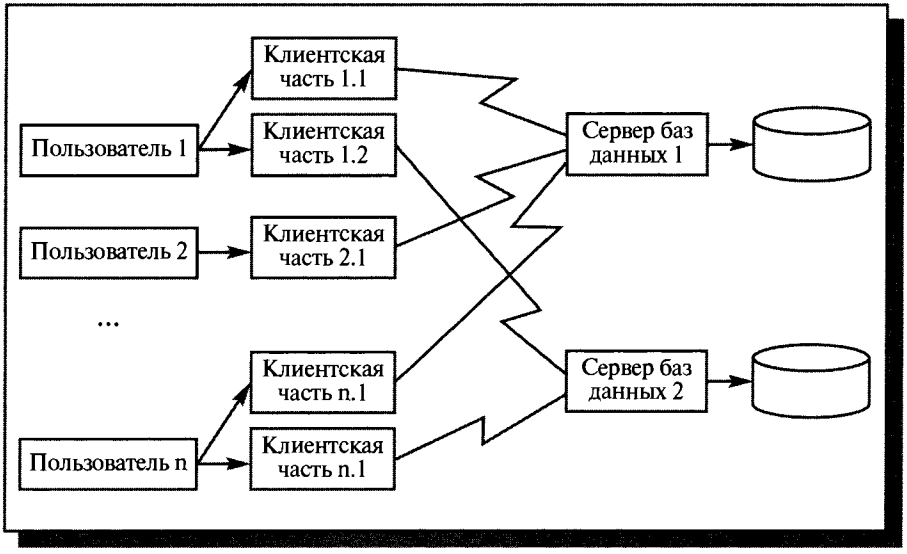


Рис. 18.8. Клиент-серверная архитектура СУБД

Конечно, это рисунок весьма условен. Под термином *пользователь* здесь, конечно, понимается некоторое приложение, с которым реально работает конечный пользователь (например, в этом приложении может быть реализован монитор прямого SQL). *Клиентская часть* СУБД – это тот системный компонент, с которым непосредственно взаимодействует *пользователь*. Данный компонент скрывает специфику реальных взаимодействий с серверной частью СУБД (например, используемые сетевые протоколы, если клиентская и серверная части СУБД разнесены по разным компьютерам сети). Наконец, *сервер*

баз данных представляет собой основную часть СУБД, где, собственно, и происходит выполнение операторов SQL и осуществляется доступ к базе данных.

Важно обратить внимание, что программные компоненты, представляющие *пользователя* и *клиентскую часть* СУБД, обычно выполняются на одном компьютере, а *сервер баз данных* работает на другом (серверном) компьютере. Но вполне может быть, что все три перечисленных программных компонента в действительности размещены на одном компьютере.

Установление соединений

Начиная со стандарта SQL/92, при разработке языковых средств стала приниматься во внимание клиент-серверная организация СУБД. Если говорить более точно, стал очевиден тот факт, что во всех существующих СУБД до начала работы приложения со средствами управления базой данных требуется выполнить некоторые предварительные инициирующие действия. В частности, необходимо создать контекст, в котором будет работать система баз данных. В некоторых реализациях этот контекст создается автоматически при запуске приложения, поскольку клиентская часть СУБД компонуется к приложению. В других случаях прикладная программа связывается с СУБД за счет наличия специализированных реализационно зависимых средств подключения к СУБД. Иногда контекст формируется на основе состояния системных переменных.

Очевидно, что для выработки языковых средств, которые не противоречили бы существующим реализациям, требовался компромисс. Этот компромисс выразился в том, что в SQL:1999 допускается установление связи приложения с СУБД по умолчанию, а также обеспечиваются средства явного управления соединениями. Общий подход состоит в следующем.

- Почти все операторы SQL (с небольшим числом исключений) могут выполняться только при наличии подключения клиентской части СУБД к серверу базы данных.
- Если соединение с сервером установлено и приложение пытается выполнить один из операторов SQL (для выполнения которых требуется соединение), то его выполняет та СУБД, с которой установлено соединение.
- Если приложение пытается выполнить один из операторов SQL (для выполнения которых требуется соединение), а соединение не установлено, то, прежде всего, требуется установить соединение. В SQL:1999 указывается, что такое соединение является соединением с СУБД по умолчанию. Что собой представляет это умолчание, определяется в реализации. После установления соединения упомянутый оператор SQL выполняется той СУБД, с которой установлено соединение.

- Если первым (до установки соединения) выполняемым оператором SQL является оператор CONNECT (это одно из исключений), то соединение по умолчанию не устанавливается, а происходит обращение к запрашиваемому серверу, и соединение устанавливается именно с ним.
- Можно выполнять оператор CONNECT для установления соединений со вторым, третьим и т. д. серверами, не разрывая ранее установленные соединения. Каждое вновь установленное соединение называется *текущим соединением (current connection)*, а все ранее установленные соединения – *отложенными соединениями (dormant connection)*.
- С каждым соединением ассоциирована сессия. Сессия, ассоциированная с текущим соединением, называется *текущей сессией (current session)*, а сессии, ассоциированные с отложенными соединениями, называются *отложенными сессиями (dormant session)*.*
- Если у приложения имеется несколько соединений, можно переключать их с помощью оператора SET CONNECTION.
- Для поддержания установленных соединений могут расходоваться значительные системные ресурсы. Поэтому может возникнуть потребность в ликвидации соединения. Это можно сделать с помощью оператора DISCONNECT. Все соединения, не ликвидированные явно до завершения работы приложения, ликвидируются системой автоматически. Попытка ликвидировать текущее соединение, в котором выполняется транзакция, расценивается как ошибка.
- В реализации определяется, можно ли переключать соединения во время выполнения транзакции. Однако если реализация это допускает, то, в соответствии со стандартом, все операторы, выполняемые в одной транзакции, но в разных соединениях, являются частью одной общей транзакции.

Операторы SQL для управления соединениями

Как отмечалось выше, в эту группу входят операторы CONNECT, SET CONNECTION и DISCONNECT.

Оператор CONNECT

Оператор определяется следующими синтаксическими правилами:

```
CONNECT TO connection_target
```

* Каждому соединению соответствует одна и только одна сессия. В сообществе SQL эти термины часто используются попеременно для обозначения одного и того же. Более строгие блюстители терминологии утверждают, что термин *подключение* относится к сетевому пути между клиентом и сервером, а *сессия* – это контекст, в котором работает SQL-сервер.

```
connection_target ::= SQL_server_name
                        [ AS connection_name ]
                        [ USER connection_user_name ]
                    | DEFAULT
```

Здесь SQL_server_name – это литерально заданная символьная строка, идентифицирующая сервер, к которому требуется подключиться. Смысл (и формат) этого имени определяется в реализации.

В необязательном разделе AS указываемое имя (connection_name) выступает в роли временного имени соединения, которое впоследствии может быть использовано в операторах SET CONNECTION и DISCONNECT. Если в операторе CONNECT раздел AS не содержится, то по умолчанию connection_name совпадает с SQL_server_name.

В необязательном разделе USER указываемое имя (connection_user_name) идентифицирует пользователя, от имени которого устанавливается соединение. При отсутствии раздела USER в качестве connection_user_name по умолчанию принимается текущий authID. В стандарте допускается, что реализация может ограничить возможные значения connection_user_name (например, потребовать, чтобы это имя всегда совпадало с текущим authID).

Эффект использования оператора в форме CONNECT TO DEFAULT почти не отличается от результата действия системы при отсутствии какого-либо явного требования соединения. (Напомним, что соединение по умолчанию неявно устанавливается при попытке выполнения первого оператора SQL, требующего соединения.) Однако имеется одно важное отличие. Если соединение по умолчанию устанавливается неявно, а затем вдруг прерывается из-за какой-то ошибки, то оно автоматически переустанавливается при выполнении следующего оператора SQL. Если же соединение по умолчанию устанавливается явным образом, то автоматическое повторное установление соединения после его разрыва не производится.

Оператор SET CONNECTION

Оператор определяется следующими синтаксическими правилами:

```
SET CONNECTION connection_object
connection_object::= { connection_name | DEFAULT }
```

Условием успешного выполнения операции является наличие отложенного установленного соединения с именем connection_name или отложенного установленного соединения по умолчанию. В этом случае

текущее соединение становится отложенным, а указанное отложенное соединение – текущим.

Оператор DISCONNECT

Оператор имеет следующий синтаксис:

```
DISCONNECT { connection_object | ALL | CURRENT }
```

Необходимым условием для возможности ликвидации соединения является отсутствие активной транзакции в этом соединении.

Если в операторе указывается `connection_object`, то соответствующее имя должно соответствовать установленному (текущему или отложенному) соединению. Если указывается `CURRENT`, то должно существовать текущее соединение.

Если оператор применяется к текущему соединению, то это соединение ликвидируется, и ни одно соединение не является текущим. В таком случае для продолжения работы необходимо установить текущее соединение при помощи операторов `CONNECT` или `SET CONNECTION`.

Если в операторе указывается `ALL`, то ликвидируются все соединения, включая текущее.

Заключение

В этой лекции были рассмотрены три темы, которые являются относительно независимыми, но относятся к средствам языка SQL, предназначенным для регулирования доступа пользователей к базам данных. На первый взгляд материал этой лекции проще материала предыдущих лекций, посвященных языку SQL. Наверное, это действительно так, если говорить про чисто языковую сложность соответствующих операторов SQL. Но в действительности (которую мы старательно обходили в основных разделах лекции) дело обстоит гораздо сложнее.

Как легко видеть, при распространении привилегий и ролей могут возникать произвольно сложные ориентированные графы связей между объектами базы данных, владельцами привилегий, привилегиями и ролями. Если изображать сплошными стрелками передачу привилегий, прерывистыми – передачу ролей, пунктирными – владение привилегиями, а точечными – владение ролями, то даже по отношению к одной привилегии *pr* для одного объекта *o* может появиться следующий граф связей (`userID` означает `authID`, отличный от имени роли), показанному на рис. 18.8.

Как мог появиться такой граф? Пользователь с `authID`, равным `userID1` (это мы предположили для упрощения, а вообще-то это могло

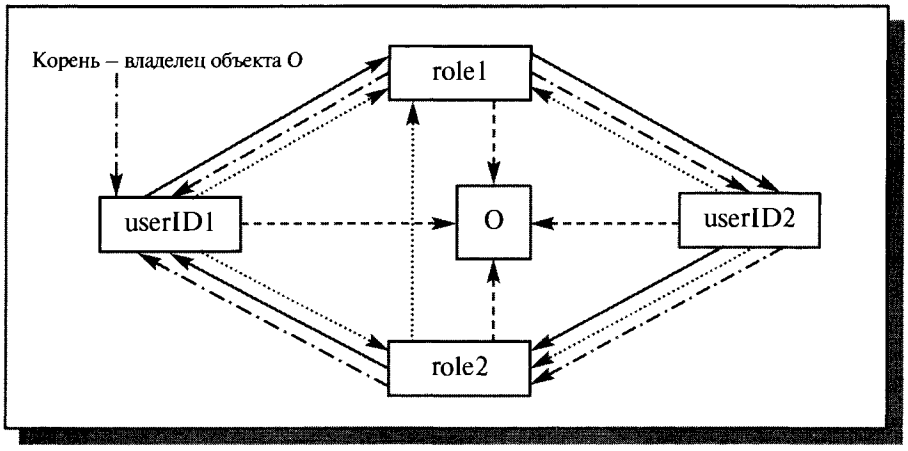


Рис. 18.8. Простейший граф идентификаторов пользователя, имен ролей, объектов и привилегий

быть и именем роли), создает объект *o*, становится его владельцем и тем самым обладателем привилегии *pr* по отношению к этому объекту. Пользователь `userID1` предоставляет полномочие *pr* роли `role1` (с правом передачи). Затем пользователю `userID1` предоставляется роль `role1` (с правом передачи), и он получает право исполнять эту роль. От имени роли `role1` полномочие *pr* передается пользователю `userID2` (с правом передачи), и этот же пользователь получает право исполнять роль `role1` (с правом передачи). Пользователь `userID2` передает роли `role2` роль `role1` и полномочие *pr* (с правом передачи). Наконец, от имени роли `role2` полномочие *pr* и сама роль `role2` передаются пользователю `userID1`.

Попробуйте теперь проследить, как будет выполняться операция

```
REVOKE pr ON o FROM role1 CASCADED
```

Какие узлы и дуги останутся в графе? Задача не очень сложная, но, очевидно, нетривиальная. И такого рода задачи приходится ежедневно решать администраторам больших и динамических SQL-ориентированных баз данных.

Теперь немного поговорим об управлении транзакциями. В стандарте SQL:1999 ничего не говорится о возможной реализации различных уровней изоляции. Конечно, это правильно, поскольку спецификация языка не должна накладывать какие-либо ограничения на реализации. Но, к сожалению, при использовании SQL-ориентированной СУБД некоторые знания о реализации механизма транзакций необходимы. Например, предположим, что имеются две транзакции `T1` и `T2`, выполняемые в

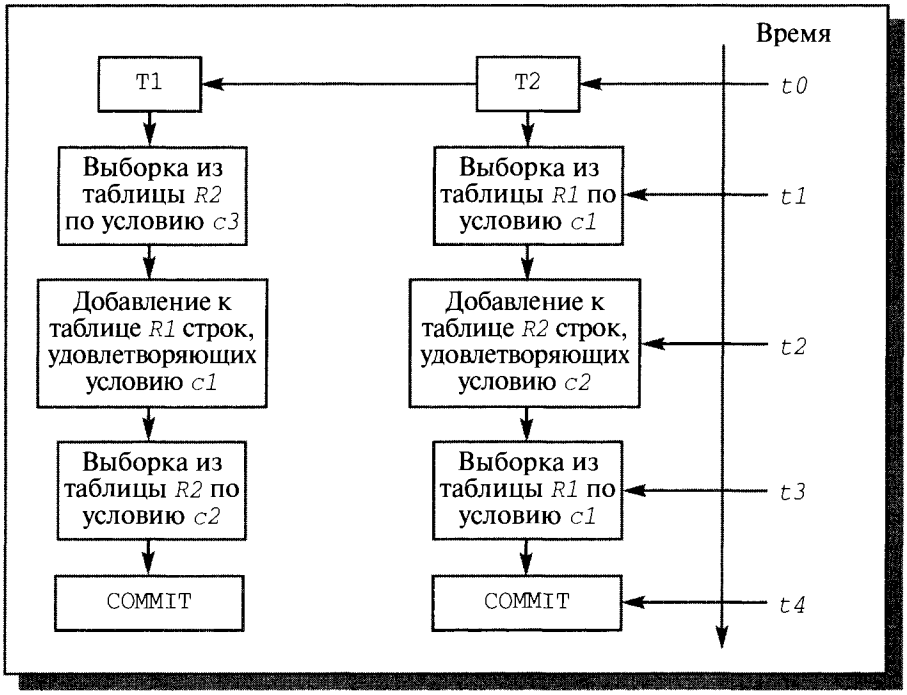


Рис. 18.9. Взаимные «фантомы»

режиме изоляции SERIALIZABLE. Предположим, что они должны работать по «симметричному» плану, показанному на рис. 18.9.

Транзакции работают в наивысшем режиме изолированности. Эффект их выполнения должен быть эквивалентен эффекту некоторого последовательного выполнения транзакций T1 и T2. Но попробуйте придумать какой-либо корректный способ одновременного выполнения этих транзакций, который привел бы к эффекту их последовательного выполнения. Другими словами, для грамотного использования механизма транзакций на уровне языка SQL необходимо знать, каким образом данный механизм реализован в используемой СУБД.

И, конечно же, знания о реализации абсолютно необходимы при использовании механизма подключений и сессий. Слишком много в этой части стандарта отдается на волю реализации.

Лекция 19. Язык баз данных SQL: объектные расширения

В последней лекции этого курса мы кратко изложим суть объектных расширений, которые включены в стандарт SQL:1999. Лекция основана не на официальном тексте стандарта (он очень формален и скучен), а на книге Джима Мелтона «Advanced SQL:1999. Understanding Object-Relational and Other Advanced Features» (Morgan Kaufmann Publishers, 2003), которая, по сути, является неформальным описанием семантики (rationale) соответствующей части языка. В указанной книге объектным расширениям языка SQL посвящено более 200 страниц. Естественно, наше изложение будет гораздо более кратким.

Ключевые слова: компания IBM, проект System R, объектно-реляционный подход к организации систем баз данных, объектно-реляционные системы управления базами данных (ОРСУБД), СУБД Ingres, компания Computer Associates, СУБД Postgres, СУБД PostgreSQL, объектно-ориентированные системы управления базами данных (ООСУБД), компания Illustra, СУБД Illustra, компания Informix, ОРСУБД Informix Universal Server, компания MCC, ООСУБД Orion, компания UniSQL, ОРСУБД UniSQL, манифест систем объектно-ориентированных баз данных, манифест систем баз данных следующего поколения, компания Oracle, ОРСУБД Oracle8, ОРСУБД DB2 Universal Database, объектная модель SQL, структурные, определяемые пользователями типы данных (User Defined Type, UDT), типизированные таблицы (Typed Table), минимальный пакет объектных свойств PKG006, полный пакет объектных свойств PKG007, индивидуальный тип (distinct type), структурный тип (structured type), оператор определения UDT, оператор `CREATE TYPE`, наследование структурных типов, максимальный структурный супертип, раздел представления в определении UDT, определение атрибута структурного UDT, метод-наблюдатель (observer), метод-мутатор (mutator), раздел `reference_scope_check` определения атрибута, инстанциируемый (instantiable) структурный тип, неинстанциируемый (not instantiable) структурный тип, раздел finality определения UDT, раздел спецификации ссылочного типа в определении структурного типа, уникальные идентификаторы экземпляров структурного типа, раздел `ref_cast_option` в определении структурного типа, раздел `cast_option` в определении индивидуального типа, раздел объявления сигнатур методов, первичный метод (original method), подменяющий метод (overriding method), метод экземпляра (`INSTANCE`), стати-

ческий метод (STATIC), метод-конструктор (CONSTRUCTOR), вызывное имя метода (invocable name), точное имя метода (specific name), характеристика метода (method characteristic), характеристика метода PARAMETER STYLE SQL, характеристика метода PARAMETER STYLE GENERAL, характеристика метода PARAMETER STYLE JAVA, детерминированный (DETERMINISTIC), недетерминированный метод (NOT DETERMINISTIC), характеристики связи метода с SQL, характеристика NO SQL, характеристика CONTAINS SQL, характеристика READS SQL DATA, характеристика MODIFIES SQL DATA, характеристика RETURN NULL ON NULL INPUT, характеристика CALLED ON NULL INPUT, определение типизированной таблицы, оператор CREATE TABLE, наследование типизированных таблиц, подтаблица, супертаблица, унаследованные столбцы подтаблицы, заново определенные столбцы подтаблицы, непосредственный супертип, определение элементов типизированной таблицы, максимальная супертаблица, самоссылающийся (self-referencing) столбец, опции столбцов (column options), ссылочное значение, ссылочный (REF) тип, механизмы генерации ссылочных значений, поддержка согласованности ссылок, раздел SCOPE, конструкция reference_scope_check, операция разыменования (dereferencing) в SQL:1999, разрешение ссылки (reference resolution) в SQL:1999, спецификация ONLY в операциях DELETE и UPDATE над типизированными таблицами, типизированные представления, представление, на которое можно ссылаться (referenceable view), объектное представление, суперпредставление, подпредставление, непосредственное суперпредставление, непосредственное подпредставление, собственное суперпредставление, собственное подпредставление, семейство подтаблиц, определение типизированного представления, оператор CREATE VIEW, базисная таблица представления, максимальное суперпредставление.

Введение

Как отмечалось в Лекции 11, язык SQL появился в середине 1970-х гг. при выполнении экспериментального проекта реляционной СУБД System R. Проект выполнялся в компании IBM, и это вполне естественно, потому что именно сотрудник IBM Эдгар Кодд предложил миру идею реляционных баз данных. От System R исходит большинство традиционных средств стандарта SQL:1999 (и SQL:2003), которые мы обсуждали в восьми предыдущих лекциях. Однако в этой лекции речь пойдет о возможностях современных вариантов SQL, которые не имеют отношения к System R (за исключением некоторых экспериментов по представлению сложных объектов средствами SQL) и, вообще говоря, к реляционной модели данных, а именно — о так

называемых *объектно-реляционных* расширениях языка. Чтобы у читателей не возникло впечатление, что объектные расширения появились в языке SQL внезапно, благодаря какому-то озарению разработчиков языка, мы начнем эту лекцию с небольшого (и крайне субъективного) очерка истории *объектно-реляционного подхода* к организации систем баз данных.

Истоки и краткая история объектно-реляционных баз данных

Пальму первенства в области объектно-реляционных систем управления базами данных (ОРСУБД) оспаривают два весьма известных специалиста в области технологии баз данных – Майкл Стоунбрейкер (Michael Stonebraker) и Вон Ким (Won Kim).

Первые ОРСУБД

Майкл Стоунбрейкер начал работать в области баз данных в начале 1970-х гг. прошлого века в университете Беркли. Его первым всемирно известным проектом была реляционная СУБД Ingres, которая существует и используется до сих пор в двух ипостасях – как свободно распространяемая система (*университетская* Ingres; код поддерживается в Беркли) и как коммерческая СУБД, принадлежащая компании Computer Associates. В исходном варианте СУБД Ingres отсутствовала поддержка языка SQL (поддерживался собственный язык запросов QUEL), но система уже обладала некоторыми уникальными чертами, которые, с небольшой натяжкой, можно было бы назвать *объектными* (например, в СУБД Ingres допускалось определение пользовательских процедур, выполняемых на стороне сервера). Кроме того, в проекте Ingres очень большое внимание уделялось управлению правилами.

В 1980-е гг. Майкл Стоунбрейкер возглавлял проект Postgres (вариант этой системы под названием PostgreSQL в настоящее время является весьма популярным свободно доступным продуктом). В Postgres были реализованы многие интересные средства: поддерживалась темпоральная модель хранения и доступа к данным, и в связи с этим был полностью пересмотрен механизм журнализации изменений, откатов транзакций и восстановления БД после сбоев; обеспечивался мощный механизм ограничений целостности; поддерживались ненормализованные отношения (работа в этом направлении началась еще в среде Ingres), хотя и довольно странным способом: в поле отношения мог храниться динамически выполняемый запрос к БД.

Одно свойство системы Postgres сближало ее со свойствами объектно-ориентированных СУБД (ООСУБД). В Postgres допускалось хранение в полях отношений данных абстрактных, определяемых пользователями

типов. Это обеспечивало возможность внедрения поведенческого аспекта в БД, т. е. решало ту же задачу, что и ООСУБД, хотя, конечно, семантические возможности модели данных Postgres были существенно слабее, чем у объектно-ориентированных моделей данных. Основная разница состояла в том, что в Postgres не предполагалось наличие языка программирования, одинаково понимаемого как внешней системой программирования, так и системой управления базами данных. Как и в Ingres, в исходном варианте Postgres не поддерживался язык SQL (имелся собственный язык запросов Postquel). Кстати, во времена Postgres Майкл Стоунбрейкер не использовал термин *объектно-реляционная система*, предпочитая называть свою СУБД *системой следующего поколения*.

В начале 1990-х гг. Стоунбрейкер создал компанию Illustra, основной целью которой был выпуск коммерческого варианта СУБД Postgress, получившего название Illustra. В этой системе поддерживались основные идеи Postgres, но уже присутствовала и поддержка языка SQL. В конце 1995 г. компания Illustra была поглощена компанией Informix, и это привело к выпуску в 1996 г. СУБД Informix Universal Server (см. ниже).

Имя Вона Кима стало широко известно во второй половине 1970-х гг., когда он примкнул к участию в экспериментальном проекте компании IBM System R. Наиболее известная ранняя работа доктора Кима была посвящена преобразованию SQL-запросов с целью превращения запросов с вложенными подзапросами в запросы с соединениями.

В 1980-е гг. Вон Ким работал в компании MCC, где успешно выполнил реализацию серии прототипов ООСУБД Orion. В этих прототипах были опробованы многие идеи объектно-ориентированных СУБД. Одной из интересных особенностей проекта было то, что в качестве основного языка программирования использовался объектный вариант известного функционального языка Lisp.

В конце 80-х гг. д-р Ким основал компанию UniSQL, выпустившую в 1991 г. первую версию продукта UniSQL, который Вон Ким стал называть *объектно-реляционной системой*. Трудно оценивать коммерческий успех этой СУБД. В настоящее время она принадлежит Корейской национальной телекоммуникационной компании и, по всей видимости, продолжает использоваться. Поскольку UniSQL была первой СУБД, официально называемой объектно-реляционной системой, приведем ее краткое описание.

UniSQL обеспечивала возможность построения так называемых *федеративных систем баз данных*. При этом обеспечивалось единое представление данных, которые могли храниться либо в базе данных, непосредственно управляемой UniSQL, либо в какой-либо из реляционных баз данных, управляемой СУБД Oracle, Informix, Sybase и т. д., либо в какой-либо дореляционной базе данных. Сервер UniSQL обеспечивал инте-

грированный доступ к данным, управляемым разными СУБД. Одна из возможных конфигураций использования системы показана на рис. 19.1.

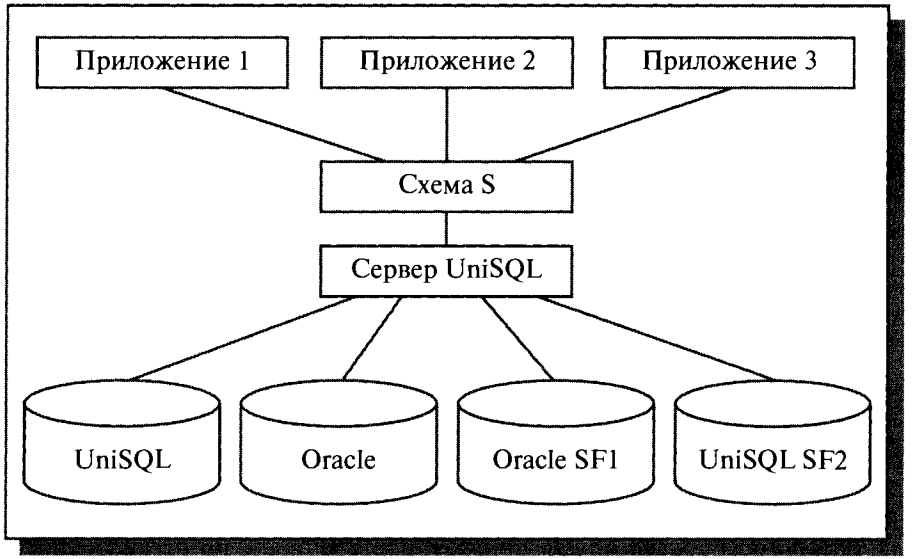


Рис. 19.1. Возможная конфигурация системы UniSQL

Как показывает рис. 19.1, сервер UniSQL позволяет представлениям работать через «глобальную» схему базы данных S, полученную из двух «фрагментарных» схем баз данных, которые управляются непосредственно UniSQL и СУБД Oracle.*

Разработчики UniSQL полагали, что построение полнофункциональной СУБД, основанной на принципиально новой модели данных, крайне проблематично. Был выбран подход к расширению реляционной модели, выражающийся в следующих четырех принципах:

- значениями атрибутов отношений могут быть не только литеральные значения, но и объекты;

* Вопросы интеграции данных выходят за пределы тематики этого курса. Однако следует сделать два замечания. Во-первых, проблематика обеспечения доступа к разнородным данным через некоторую глобальную, или концептуальную схему интересует сообщество баз данных в течение нескольких десятков лет. Существовали многочисленные попытки обеспечить интеграцию баз данных, представленных во всех возможных моделях (сетевой, иерархической, реляционной, объектно-ориентированной). С точки зрения теории решение проблемы возможно, но на практике это приводит к очень сложным с технической точки зрения реализациям, обладающим крайне низкой производительностью. Во-вторых, в МСС в 1980-е годы был создан весьма успешный прототип системы, интегрирующей SQL-ориентированные базы данных. Должно быть понятно, что такая интеграция существенно проще в техническом смысле, поскольку глобальная и фрагментарные схемы представлены в близких понятиях. Похоже, что проект UniSQL в большой степени базировался и на этой работе.

- значения атрибутов отношений не обязательно являются атомарными;
- при построении таблиц (классов) может использоваться механизм наследования;
- классы включают операции.

В созданной компанией системе поддерживалось расширение стандарта SQL – SQL/X, одновременно включающее и объектно-ориентированные, и реляционные возможности. В одном языке поддерживались возможности и определения данных, и манипулирования ими. В качестве языковых средств программирования приложений поддерживались языки C++ и Smalltalk.

Внедрение объектных расширений в основные РСУБД

В конце 1989 г. группа известных специалистов в области языков программирования баз данных опубликовала документ под названием *Манифест систем объектно-ориентированных баз данных* (для краткости будем называть его *Первым манифестом*). Основным поводом к написанию и публикации этого материала было то, что к тому времени существовал ряд систем управления базами данных, которые, по большому счету, объединяла только общая привязанность к объектно-ориентированным языкам программирования. В *Первом манифесте* была предпринята попытка дать определение системам объектно-ориентированных баз данных. Авторы стремились привести описание основных свойств и характеристик, которыми должна обладать система, претендующая на то, чтобы называться системой объектно-ориентированных баз данных. *Первый манифест* был написан академическими исследователями; почти все они являлись и являются профессорами различных университетов. Конечно, это нашло свое отражение в стиле Первого манифеста – очень мягком и умеренно рекомендательном (хотя по своему духу предложения этого манифеста были весьма радикальными).

Через год после публикации *Первого манифеста* вышел в свет *Манифест систем баз данных третьего поколения*, инициатором которого, очевидно, был Майкл Стоунбрейкер (хотя у документа формально имелось много авторов). Мы говорим об этом с уверенностью, поскольку в этом манифесте повсюду видны идеи Стоунбрейкера, использованные им в проектах Ingres и Postgres.

В некотором роде *Манифест систем баз данных следующего поколения* (для краткости мы будем называть его *Вторым манифестом*) стал ответом миру объектно-ориентированных баз данных со стороны мира SQL-ориентированных баз данных. Если *Первый манифест* был хотя и немного путанным, но все-таки носил научный характер, то *Второй манифест* является в большей степени инженерно-публицистическим документом.

Второй манифест можно расценивать как реакцию индустрии СУБД на неприятные для нее измышления науки.

Второй манифест (или, вернее, работы, приведшие к его появлению) имел важные последствия. В 1995 г. компания Informix (ныне входящая в состав IBM) купила компанию Майкла Стоунбрейкера Illustra*, и Стоунбрейкер стал техническим директором Informix. В начале 1996 г. компания Informix объявила о выпуске принципиально нового продукта Informix Universal Server, в котором, как утверждалось, лучшие черты Informix Online Server сочетались с развитыми объектными чертами, присущими Illustra.

К выпуску Informix Universal Server очень ревниво отнеслась компания Oracle, которая немедленно заявила, что у нее готов собственный объектно-реляционный продукт, по всем параметрам превосходящий систему компании Informix. Эта система, получившая название Oracle8, была выпущена в конце лета 1996 г.

Годом позже к группе производителей объектно-реляционных СУБД (ОРСУБД) примкнула компания IBM, выпустившая продукт DB2 Universal Database. Как выяснилось позже, все наиболее важные свойства этого продукта были реализованы еще в 1995 г. в СУБД DB2 for Common Servers. Просто компания IBM предпочла до поры не афишировать свои расширения.

Первые пару лет вокруг объектно-реляционных СУБД стоял большой шум. Позже выяснилось, что маркетинговые ожидания компаний-гигантов оказались преувеличенными. (В частности, это было одной из основных причин падения компании Informix.) Сегодня объектные расширения SQL-ориентированных СУБД предлагаются пользователям лишь в качестве дополнительных, хотя и важных возможностей.

Объектные расширения языка SQL были зафиксированы в стандарте SQL:1999. В той или иной мере эти расширения поддерживаются во всех трех перечисленных выше продуктах. В настоящее время ближе всех к стандарту находятся СУБД компании Oracle и DB2 компании IBM.

Объектная модель SQL

Объектные расширения SQL:1999 базируются на некоторой объектной модели, хотя эта модель в явном виде в стандарте не специфицируется. Объектная модель SQL** не является тождественной объектным моделям какого-либо объектно-ориентированного языка программирования или какой-либо объектно-ориентированной системы баз данных. Однако при определении объектной модели SQL участники процесса

* Компания Illustra была создана Стоунбрейкером для коммерциализации разработанной под его руководством свободно доступной СУБД Postgres.

** Конечно, это не модель данных в смысле Кодда.

стандартизации тщательно проанализировали ряд других языков и систем с целью выяснения достоинств и недостатков их объектных моделей. По мнению авторов стандарта SQL:1999, выработанная ими объектная модель похожа на объектную модель языка Java, но при этом адаптирована к природе языка SQL как языка СУБД с наличием стабильно хранимых метаданных и данных*.

Объектная модель SQL:1999 включает два основных отличительных компонента – *структурные, определяемые пользователями типы данных (User Defined Type – UDT)* и *типизированные таблицы (Typed Table)*. Первый компонент позволяет определять новые типы данных, которые могут быть гораздо более сложными, чем встроенные типы данных языка SQL. При определении структурного UDT требуется специфицировать не только содержащиеся в нем элементы данных, но и семантику типа данных, т. е. его поведение на основе интерфейса вызовов методов. Второй компонент – типизированные таблицы – позволяет определять таблицы, строки которых являются экземплярами (или значениями) UDT, с которым явно ассоциируется таблица. Во многих отношениях строка типизированной таблицы похожа на объект класса в объектно-ориентированной системе.

В стандарте SQL:1999 определены два пакета объектных свойств – минимальный (PKG006) и полный (PKG007), которым должны удовлетворять SQL-ориентированные ОРСУБД, претендующие на соответствие стандарту. Ниже будут перечислены свойства, включенные в каждый из пакетов, но смысл этих свойств будет понятен только после прочтения остальных разделов.

Пакет PKG006 включает всего пять свойств:

- свойство S023 (*«Basic structured types»*) – возможность определять UDT и их методы с ограниченными возможностями;
- свойство S041 (*«Basic reference types»*) – возможность определять и использовать ссылки на экземпляры UDT, входящие в типизированные таблицы;
- свойство S051 (*«Create table of type»*) – возможность создания типизированных таблиц;
- свойство S151 (*«Type predicate»*) – возможность определения точного типа (в иерархии типов) экземпляра UDT;
- свойство T041 (*«Basic LOB data type support»*) – возможность определения LOB-типов в смысле SQL (с необязательной поддержкой операций, кроме операций сохранения и полной выборки).

Пакет PKG007 содержит девять дополнительных свойств:

* Далеко не факт, что ориентация на язык Java была правильным решением. По мнению автора данного курса, причиной являются отнюдь не уникальные достоинства языка Java (обсуждение этого языка не является задачей автора), а то, что во время разработки стандарта SQL:1999 язык Java был особенно моден. Помимо прочего, заметим, что для языка Java (насколько известно автору) никогда не определялась формальная объектная модель.

- свойство S024 (*«Enhanced structured types»*) добавляет к свойству S023 ряд развитых возможностей, в число которых входят возможности кодирования методов на языках, отличных от SQL, сравнения экземпляров UDT и передача экземпляров UDT в качестве параметров различных процедур;
- свойство S043 (*«Enhanced reference types»*) расширяет свойство S041 возможностями определения ссылок с областью действия, автоматической проверки законности ссылок и т. д.;
- свойство S071 (*«SQL-paths in function and type name resolution»*) позволяет использовать путевые выражения SQL (*SQL-path*) в алгоритме разрешения типа;
- свойство S081 (*«Subtables»*) расширяет возможности свойства S051, допуская организацию иерархии таблиц, аналогичной иерархии типов соответствующих UDT;
- свойство S111 (*«ONLY in query expressions»*) обеспечивает возможность выборки *только* экземпляров указанного типа, без экземпляров любого из его подтипов;
- свойство S161 (*«Subtype treatment»*) позволяет информировать среду SQL о том, что некоторый экземпляр UDT в действительности является экземпляром указанного подтипа;
- свойство S211 (*«User-defined cast functions»*) разрешает определять подпрограммы, преобразующие экземпляры UDT к другим типам;
- свойство S231 (*«Structured type locators»*) способствует доступу к экземплярам UDT из прикладных программ;
- свойство S023 (*«Transform functions»*) позволяет определять подпрограммы, преобразующие значения UDT в значения предопределенных типов данных, и наоборот.

Цели лекции

Следует заметить, что в этой, безусловно, перегруженной материалом лекции мы преследуем две основные цели. Первая цель состоит в том, чтобы показать читателям, что в средствах определения структурных типов SQL используются, по сути, все базовые возможности определения объектов базы данных и выборки данных, которые обсуждались в предыдущих лекциях. Более того, определенные пользователями типы в SQL являются объектами первого класса; UDT можно использовать в любой конструкции языка, в которой допускается применение предопределенного или конструируемого типа данных. Очень важно отдавать себе отчет в том, что наличие возможности определять пользовательские типы не делает язык SQL менее реляционным или более объектным. Эта возможность «всего лишь» фантастически повышает выразительную мощность языка.

Второй целью является демонстрация того, как на основе базовых механизмов языка удалось ввести дополнительные конструкции, которые действительно вплотную приближают SQL к объектному миру. Здесь, конечно, основную роль играет связка UDT и механизма типизированных таблиц, которые играют в SQL своеобразную совмещенную роль классов и коллекций объектов.

Может оказаться, что материал этой лекции покажется сложным, поскольку для его усвоения не помешало бы иметь предварительную подготовку в области полнотиповых языков программирования, объектно-ориентированных языков и систем баз данных и т. д. Хочется надеяться, что возникновение трудностей при изучении лекции не отпугнет читателей от этой темы, а напротив, послужит стимулом к изучению дополнительной литературы.

Возможна и другая опасная ситуация. Краткость и некоторая формальность изложения может создать ложное впечатление тривиальности объектных расширений SQL. В этом случае могу посоветовать перечитать предыдущие лекции курса, относящиеся к SQL, считая, что везде, где используются предопределенные или конструируемые типы, применяются некоторые UDT, а в тех случаях, где имеются таблицы, связываемые естественным соединением, используются типизированные таблицы. Думаю, это позволит оценить мощь новых возможностей SQL.

Введя этот необходимый контекст, перейдем к описанию соответствующих механизмов SQL:1999.

Определяемые пользователями типы

Один из основных упреков по адресу языка SQL, звучавший, в частности, в Первом манифесте, заключался в отсутствии каких бы то ни было возможностей хранить в базе данных данные, тип которых являлся бы не предопределенным, а определяемым пользователями. Отрицательные последствия отсутствия такой возможности признавались и во Втором манифесте. В SQL:1999 этот дефект был устранен. Как отмечалось в лекции 11, в стандарте поддерживается возможность определения пользователями двух разновидностей UDT – *структурных типов (structured type) и индивидуальных типов (distinct types)*.

Индивидуальные типы

Напомним, что индивидуальным типом называется UDT, основанный на единственном встроенном типе (например, `INTEGER`). Значения такого типа нельзя *прямо* использовать в операциях соответствующего базового типа, однако допускается явное приведение значений индивидуального типа к базовому типу. Поясним это на примерах.

Пусть заданы следующие определения индивидуальных типов:

```
CREATE TYPE EMP_NO AS INTEGER FINAL;
CREATE TYPE DEPT_NO AS INTEGER FINAL;
CREATE TYPE PRO_NO AS INTEGER FINAL;
```

Таблицу EMP можно определить следующим образом (упрощенный вариант):

```
CREATE TABLE EMP (
   EMP_ID EMP_NO,
   EMP_NAME VARCHAR(20),
   DEPT_ID DEPT_NO,
   PRO_ID PRO_NO);
```

Такое определение таблицы приведет к тому, что хотя все три индивидуальных типа определены на одном и том же базовом типе INTEGER, попытка выполнить запрос

```
SELECT EMP_NAME
FROM EMP
WHERE EMP_ID > DEPT_ID;
```

будет отвергнута системой (и это правильно, поскольку, скорее всего, запрос задан по ошибке). Но если действительно требуется сравнивать идентификаторы служащих с идентификаторами их отделов, то можно воспользоваться конструкцией явного приведения типа:

```
SELECT EMP_NAME
FROM EMP
WHERE CAST (EMP_ID TO INTEGER) > CAST (DEPT_ID TO INTEGER);
```

Аналогичным образом будет отвергнут запрос

```
SELECT EMP_NAME, EMP_ID + 5
FROM EMP
WHERE DEPT_ID > 630;
```

Чтобы указать системе, что действительно требуется выполнить операции целочисленного сложения и сравнения над значениями индивидуальных типов, запрос нужно переписать следующим образом:

```
SELECT EMP_NAME, CAST (EMP_ID TO INTEGER) + 5
FROM EMP
```

```
WHERE CAST (DEPT_ID TO INTEGER) > 630;
```

У читателей могут возникнуть два законных вопроса:

- почему, вопреки обыкновению, мы не привели формальные синтаксические правила операции определения индивидуального типа?
- что означает ключевое слово FINAL в приведенных примерах определения индивидуальных типов?

На оба эти вопроса достаточно дать один (возможно, неожиданный) ответ. С формальной точки зрения индивидуальный тип данных является частным случаем структурного типа данных. Обе разновидности UDT определяются единым синтаксисом, который мы обсудим в следующих подразделах. В частности, ключевое слово FINAL играет важную роль в определении структурного типа, указывая на тот факт, что этот тип может использоваться только для создания объектов, а не для порождения новых типов на основе механизма наследования. При определении индивидуальных типов механизм наследования не используется, и поэтому в определении любого индивидуального типа должно присутствовать ключевое слово FINAL.* Далее, поскольку индивидуальный тип является частным типом структурного типа, для индивидуального типа можно определять методы.

В своих книгах главный редактор стандартов SQL Джим Мелтон постоянно подчеркивает семантическое сходство понятий индивидуального типа данных и домена в смысле SQL (лекция 11). Более того, утверждается, что в следующих версиях стандарта SQL использование доменов будет сначала объявлено нежелательным, а потом и вовсе будет запрещено. Но я полагаю, что сделать это совсем непросто.

Напомним, что в случае использования SQL-домена:

(a) в определении домена указывается базовый встроенный тип данных и, возможно, ограничение допустимых значений, которое распространяется на любой столбец, определенный на данном домене;

(b) для значений столбца, определенного на домене, допускаются все операции, разрешенные для базового типа.

Естественно, эти возможности могут использоваться текущими пользователями стандарта SQL. В то же время в случае использования индивидуального типа данных:

(a) в определении индивидуального типа указывается только базовый тип данных; если столбец определяется на индивидуальном типе данных, то для него обязательно придется специфицировать собственное ограничение целостности;

* Кстати, не очень понятно, по каким причинам в стандарте SQL не поддерживается наследование для индивидуальных типов. Конечно, этот механизм существенно более полезен для структурных типов, но его вполне можно было бы реализовать и для индивидуальных типов.

(b) для значений столбца, определенного на индивидуальном типе данных, не допускаются операции соответствующего базового типа (если не использовать операцию явного приведения типов).

Здесь явно имеются противоречия, для сглаживания которых требуется модифицировать понятие индивидуального типа данных.

Определение структурных типов

Общий синтаксис оператора определения UDT (индивидуального или структурного) определяется следующими правилами:

```
UDT_definition ::= CREATE TYPE UDT_name
                        [ subtype_clause ]
                        [ AS representation ]
                        [ instantiable_clause ]
                        finality
                        [ reference_type_specification ]
                        [ ref_cast_option ]
                        [ cast_option ]
                        [ method_specification_commalist ]
```

Имя определяемого пользователем типа данных имеет, в общем случае, традиционную для SQL трехзвенную структуру – `имя_каталога.имя_схемы.имя_типа`. Раздел подтипизации задается в следующем синтаксисе:

```
subtype_clause ::= UNDER UDT_name
```

Если этот раздел присутствует в определении UDT, то в нем указывается имя ранее определенного UDT, атрибуты и методы которого будут наследоваться определяемым структурным типом. Структурные типы, определяемые без использования наследования, называются *максимальными супертипами* (поскольку у любого из таких типов супертип отсутствует).* В определениях максимального структурного супертипа или индивидуального типа должен присутствовать раздел представления (AS):

```
representation ::= predefined_type
                 | (attribute_definition_ commalist)
```

Если в разделе представления указывается имя предопределенного встроенного типа, то определяется индивидуальный тип. Указание спис-

* Как уже отмечалось ранее, раздел подтипизации может присутствовать только при определении структурного UDT.

ка определений атрибутов соответствует определению структурного типа. Заметим, что раздел представления может отсутствовать. В этом случае должен присутствовать раздел подтипизации, и представление заново определяемого структурного типа полностью наследуется из определения структурного UDT, имя которого указано после ключевого слова UNDER.

Определение атрибута структурного UDT

Определение атрибута имеет следующий синтаксис:

```
attribute_definition ::= attribute_name data_type
                         [ reference_scope_check ]
                         [ default_clause ]
                         [ collate_clause ]
```

Имя определяемого атрибута должно отличаться от имен всех других атрибутов определяемого типа, включая имена атрибутов, наследуемых от супертипа, и имена атрибутов типа данных определяемого атрибута. Тип данных может быть любым допустимым в SQL типом данных (включая конструируемые типы ARRAY* и ROW, а также UDT), кроме самого определяемого структурного типа и его супертипов**.

Для атрибута можно объявить значение по умолчанию. Если типом данных атрибута является встроенный тип данных, то значение атрибута объявляется в том же синтаксисе, что и значение столбца по умолчанию в определении таблицы (см. лекцию 12). Если типом данных атрибута является UDT (индивидуальный или структурный), тип ROW или ссылочный тип (см. следующий пункт), то единственным допустимым значением по умолчанию является неопределенное значение (NULL). Если же типом данных атрибута является тип ARRAY, то значением по умолчанию может быть NULL или пустое значение-массив (указывается как ARRAY[]).

Для каждого определения атрибута, в котором типом атрибута является структурный тип, система автоматически генерирует пару методов, имена которых совпадают с именем атрибута. Первый метод является *наблюдателем* (*observer*). Он вызывается без явных параметров и выдает значение указанного атрибута в значении того структурного типа, к которому применяется. Второй метод является *мутатором* (*mutator*). Он вызывается с одним явным параметром – значением типа атрибута, применяется к

* А в стандарте SQL:2003 и MULTISET.

** Последнее ограничение является непонятным. Его можно обойти, например, следующим образом. Пусть структурный тип T' определяется как подтип типа T, и мы хотим включить в представление типа T' атрибут a типа T. Тогда предварительно определим тип T'' как подтип типа T в точности с тем же представлением. Тогда ничто не помешает определить в представлении типа T' атрибут a типа T''.

некоторому местоположению (столбцу, переменной или параметру), где находится значение определяемого структурного типа, и этот вызов приводит к тому, что значение заменяется новым значением того же типа с измененным соответствующим образом значением данного атрибута.

Присутствие в определении атрибута раздела `reference_scope_check` возможно (и требуется) в том и только в том случае, когда типом определяемого атрибута является ссылочный тип. Более подробно мы обсудим суть этой спецификации в следующем разделе. Пока лишь кратко заметим, что этот раздел указывает системе, должна ли она проверять, что каждое значение данного атрибута является ссылкой на существующий экземпляр указанного структурного типа, и должна ли система вызывать ссылочное действие при удалении экземпляра, на который ведет ссылка.*

Можно определить *инстанциируемый (instantiable)* или *неинстанциируемый (not instantiable)* структурный тип:

```
instantiable_clause ::= INSTANTIABLE
                      | NOT INSTANTIABLE
```

Для неистанциируемого типа конструктор не определяется, и поэтому создать значение этого типа невозможно.** Такие типы применимы только для определения инстанциируемых подтипов. Назначение неинстанциируемых типов состоит в моделировании абстрактных концепций, на которых основываются более конкретные концепции. Неинстанциируемые типы могут быть типами атрибутов других структурных типов, типами столбцов, переменных и т. д. Однако в соответствующем местоположении всегда должно находиться либо значение инстанциируемого подтипа данного неинстанциируемого типа, либо неопределенное значение. При отсутствии явной спецификации по умолчанию тип считается инстанциируемым.

Обязательный раздел `finality` указывает на возможность или невозможность определения подтипов определяемого структурного типа:

---

* Мы вынуждены следовать терминологии стандарта SQL, которая иногда бывает довольно нечеткой. В частности, по отношению к структурным типам используются термины *значение (value)* во вполне стандартном смысле; *местоположение (site)* как расширенное понятие переменной (нечто, содержащее значение структурного типа); *экземпляр (instance)*. Последний термин в объектной терминологии обычно используется в том же смысле, что *объект класса*. В случае SQL это строка типизированной таблицы (см. следующий раздел).

** Мы снова используем обороты, принятые в стандарте SQL. Заметим, что, хотя смысл неинстанциируемого типа должен быть интуитивно понятен, приведенное определение является очень нечетким. Классическое (не вполне строгое) понятие типа данных основывается на паре <множество_значений, набор_операций>. Поэтому нельзя создать значение типа, можно только выбрать его из соответствующего множества значений. Поэтому, строго говоря, в типе данных не может присутствовать «метод-конструктор», а может иметься (или не иметься) операция выборки значения. У неинстациируемых типов такая операция отсутствует.

```
finality ::= FINAL | NOT FINAL
```

При определении индивидуального типа всегда требуется указывать FINAL. При определении структурного типа в SQL:1999 необходимо указать NOT FINAL. Это требование не обосновано, и в следующих версиях стандарта SQL будет разрешено определять структурные типы, от которых невозможно наследование.

Раздел спецификации ссылочного типа

Хотя типизированные таблицы обсуждаются в следующем разделе, мы вынуждены немного забежать вперед, чтобы ввести синтаксис и пояснить смысл раздела reference_type_specification определения структурного типа. Строки типизированных таблиц обладают всеми характеристиками объектов в объектно-ориентированных системах, включая уникальные идентификаторы, которые могут использоваться для ссылок из других компонентов среды. В SQL:1999 поддерживаются три различных механизма присваивания уникальных идентификаторов экземплярам структурных типов, ассоциированных с такими таблицами (для всех строк таблицы, ассоциированной с данным структурным типом, используется один и тот же механизм). Уникальные идентификаторы экземпляров структурного типа могут представлять собой следующее:

- значения, генерируемые системой автоматически (system_generated_representation);
- значения некоторого встроенного типа SQL, которые должны генерироваться приложением при сохранении экземпляра структурного типа как строки типизированной таблицы (user_generated_ representation);
- значения, порождаемые из одного или нескольких атрибутов структурного типа (derived_representation).

Соответственно, синтаксис раздела reference_type_specification определяется следующими правилами:

```
reference_type_specification ::= system_generated_representation
                               | user_defined_representation
                               | derived_representation
system_generated_representation :== REF IS SYSTEM GENERATED
user_defined_representation :== REF USING predefined_type
derived_representation ::= REF USING (commalist_of_attributes)
```

Раздел reference_type_specification может присутствовать только в определении максимального структурного супертипа, т. е. соответствующая спецификация наследуется всеми подтипами этого супертипа.

При отсутствии в определении супертипа явного раздела reference_type_specification по умолчанию предполагается наличие раздела REF IS SYSTEM GENERATED.

Разделы спецификации функций явного преобразования типов

Если в определении структурного типа присутствует раздел reference_type_specification и он имеет вид user_generated_representation, то в определении структурного типа должен присутствовать и раздел ref_cast_option (тем самым, раздел ref_cast_option может присутствовать только в определении максимального структурного супертипа). Спецификации этого раздела используются для преобразования предоставленных приложением значений встроенного типа в значения типа REFERENCE (REF), необходимые для реального выполнения ссылок на строки типизированной таблицы, и обратного преобразования. Синтаксис раздела определяется следующими правилами (подробнее см. в следующем разделе):

```
ref_cast_option ::= cast_to_ref
                  | cast_to_type
cast_to_ref ::= CAST (SOURCE AS REF) WITH identifier
cast_to_type ::= CAST (REF AS SOURCE) WITH identifier
```

Раздел cast_option может присутствовать только в определении индивидуального типа. Спецификации раздела обеспечивают возможности преобразования значений индивидуального типа в значения базового встроенного типа, и наоборот. Раздел имеет следующий синтаксис:

```
cast_option ::= cast_to_distinct
              | cast_to_source
cast_to_distinct ::= CAST (SOURCE_TO_DISTINCT) WITH identifier
cast_to_source ::= CAST (DISTINCT_TO_SOURCE) WITH identifier
```

Раздел объявления сигнатур методов

В разделе method_specification_commalist объявляются сигнатуры методов, ассоциируемых с определяемым структурным типом. Раздел определяется следующими синтаксическими правилами:

```
method_specification ::= original_method_specification
                       | overriding_method_specification
                       | static_field_method_specification
```

```
original_method_specification ::= partial_method_specification
                                  [ SELF AS RESULT ]
                                  [ SELF AS LOCATOR ]
                                  [ method_characteristic_list ]
overriding_method_specification ::=
     OVERRIDING partial_method_specification
partial_method_specification :==
     [ INSTANCE | STATIC | CONSTRUCTOR ] METHOD method_name
     SQL_parameter_declaration_list
     return_clause
     [ SPECIFIC specific_method_name ]
method_characteristic ::= language_clause
                        | parameter_style_clause
                        | deterministic_clause
                        | SQL_data_access_indication
                        | null_call_clause
specific_method_name ::= [ schema_name . ]
qualified_identifier
static_field_method_specification ::=
     STATIC METHOD method_name ( )
     RETURNS data_type
     [ SPECIFIC specific_method_name ]
     external variable name character_string_literal
```

Как показывает синтаксис, имеются возможности определять *первичные методы* (original_method_specification), неприменимые к любому супертипу определяемого структурного типа. Если определяемый тип является подтипом некоторого другого типа, то можно также определить *подменяющие методы* (overriding_method_specification). Подменяющий метод имеет то же имя и тот же список аргументов, что и метод, определенный в некотором супертипе определяемого типа.

Исходный метод может быть определен как *метод экземпляра* (INSTANCE), *статический метод* (STATIC) или *метод-конструктор* (CONSTRUCTOR). Методы экземпляра действуют над экземплярами определяемого типа. Статические методы не используют экземпляры типа и не влияют на них; такие методы действуют над самим типом. Наконец, методы-конструкторы используются для инициализации экземпляров типа. Поскольку у неинстанциируемого типа не может быть экземпляров, для него могут быть определены только статические методы. Если при определении первичного метода его разновидность не указывается, этот метод считается методом экземпляра.

В сигнатуре метода указывается имя, по которому этот метод будет вызываться (*вызывное имя – invocable name*). Кроме того, можно указать *точное имя метода (specific name)*, которое может использоваться для уникальной идентификации метода, если его вызывное имя перегружено. Если у метода имеются какие-либо параметры, отличные от неявного параметра SELF, то в определении должен присутствовать заключенный в скобки список пар <имя_параметра, тип_параметра>, разделяемых запятыми. Поскольку методы являются функциями, требуется указать тип возвращаемого значения. Методы могут возвращать значения любого допустимого в SQL типа, даже структурного типа, ассоциированного с методом.

Наконец, у каждого метода имеется набор *характеристик метода* (method_characteristic). Методы могут быть написаны на языке SQL (более точно, на SQL/PSM) или на любом из языков программирования, поддержка которых предусмотрена в стандарте SQL (Ada, C/C++, COBOL, Fortran, MUMPS*, Pascal, PL/1). Язык Java поддерживается в стандарте в несколько иной манере, чем другие языки. Список параметров метода может быть определен в стиле, более соответствующем стилю SQL-подпрограмм (каждый параметр может принимать неопределенное значение, и не требуется параметр кода возврата). Для этого в качестве характеристики метода нужно указать PARAMETER STYLE SQL. Можно определить список параметров в стиле, более близком стилю различных языков программирования (к параметру, который может принимать неопределенное значение, должен быть добавлен дополнительный параметр-индикатор, и должен быть явно определен выходной параметр кода ответа). В этом случае метод должен иметь характеристику PARAMETER STYLE GENERAL. Наконец, для методов, тела которых будут написаны на языке Java, нужно указать характеристику PARAMETER STYLE JAVA.**

Любой метод может быть *детерминированным* (DETERMINISTIC) или *недетерминированным* (NOT DETERMINISTIC). Детерминированный метод всегда возвращает один и тот же результат, если вызывается с одним и тем же набором аргументов при одном и том же состоянии базы данных. По умолчанию методы считаются недетерминированными.

У каждого метода имеется характеристика, указывающая связь этого метода с SQL. Можно указать следующие варианты:

- метод не содержит операторов SQL (NO SQL);

* Теперь этот язык называется M. Вокруг этого языка и его реализаций имеется, в частности, целое семейство СУБД, основанных на так называемой M-технологии. Судя по всему, наиболее успешной представительницей этого семейства является СУБД Cache известной компании InterSystems.

** Этот абзац, в частности, показывает, как много нужно знать технических (и не только технических) подробностей, чтобы реально освоить технику определения UDT в среде SQL.

- метод содержит операторы SQL, но не обращается к базе данных (CONTAINS SQL);
- метод может производить выборку из базы данных, но не обновляет базу данных (READS SQL DATA);
- в методе допускаются обновления базы данных (MODIFIES SQL DATA).

По умолчанию принимается характеристика CONTAINS SQL. Наконец, для каждого метода можно определить его реакцию на аргументы, являющиеся неопределенными значениями. Если указывается RETURN NULL ON NULL INPUT, то метод всегда возвращает неопределенное значение, если значение любого из его аргументов является неопределенным (независимо от того, что написано в теле функции, реализующей метод). Если же указывается CALLED ON NULL INPUT (или если характеристика явно не задана), то метод всегда явно выполняется (т. е. происходит вызов соответствующей функции) при вызове с любым набором аргументов.

Типизированные таблицы

В предыдущем подразделе уже упоминалась возможность определения *типизированных таблиц*, основанных на некотором структурном типе. Далее мы приведем и поясним соответствующие синтаксические правила, введем понятие иерархии типизированных таблиц и связь этой иерархии с иерархией структурных типов, а также обсудим соотношение понятия строки типизированной таблицы с понятием объекта в ООБД.

Определение типизированной таблицы

С точки зрения синтаксиса оператор определения типизированной таблицы является частным случаем оператора создания базовой таблицы CREATE TABLE, обсуждавшегося в лекции 12 (там мы не имели возможности рассматривать этот частный случай). Типизированные таблицы определяются в следующем синтаксисе:

```
typed_table_defintion ::= CREATE TABLE typed_table_name
                          OF UDT_name
                          [ UNDER typed_table_name ]
                          [ (typed_table_element_list) ]
```

Первой существенной особенностью оператора создания типизированной таблицы является обязательное наличие раздела OF, в котором указывается имя ранее определенного структурного типа. Строки типи-

зированной таблицы являются экземплярами ассоциированного с таблицей структурного типа.

Подтаблицы и супертаблицы

Далее, при определении типизированной таблицы можно объявить ее подтаблицей некоторой другой типизированной таблицы (имя супертаблицы указывается в разделе UNDER). Таблица *R'* является собственной подтаблицей супертаблицы *R*, если *R'* не совпадает с *R* (в этом случае таблица *R* является собственной супертаблицей подтаблицы *R'*). Супертаблица должна быть ассоциирована со структурным типом, являющимся непосредственным супертипом* определяемой подтаблицы. Каждый столбец указанной супертаблицы наследуется подтаблицей; наследуются и характеристики столбцов супертаблицы – значения по умолчанию, ограничения целостности и т. д. Эти столбцы называются *унаследованными столбцами* подтаблицы, и они соответствуют атрибутам UDT подтаблицы, унаследованным от UDT супертаблицы. Кроме того, подтаблица будет содержать по одному столбцу для каждого собственного атрибута ассоциированного структурного типа. Такие столбцы подтаблицы называются *заново определенными*.

Как это принято в SQL, столбцы типизированной таблицы имеют порядковые номера. При этом унаследованные столбцы нумеруются до заново определенных столбцов и имеют те же номера, которые имели столбцы супертаблицы.

Определение элементов типизированной таблицы

Заключительным компонентом определения типизированной таблицы является конструкция typed_table_element_list, являющаяся обобщением конструкции table_element_list, которая используется в определении обычной базовой таблицы (см. лекцию 12). Элемент списка элементов типизированной таблицы определяется следующим синтаксическим правилом:

```
type_table_element ::= table_constraint_definition
                     | self-referencing_column_specification
                     | column_options
```

* Тип *T* является *непосредственным супертипом* типа *T'* в том и только том случае, когда *T* является супертипом *T'*, и не существует такого типа *T''*, что *T* является супертипом *T''*, и *T''* является супертипом *T'*.

Как видно из этого правила, в определении типизированной таблицы разрешается указывать табличные ограничения целостности. Если определяемая таблица является подтаблицей некоторой супертаблицы, то в ней не допускается определение ограничения первичного ключа (PRIMARY KEY). Однако если определяется максимальная супертаблица, то в ее определении допускается спецификация PRIMARY KEY (с указанием одного или нескольких столбцов) или спецификация ограничения UNIQUE (с указанием одного или нескольких столбцов) в комбинации с указанием NOT NULL. В определении типизированной таблицы могут также содержаться спецификации ссылочных ограничений целостности. Ссылки могут вести как на типизированную, так и на обычную таблицу.

«Самоссылающийся» (self-referencing) столбец специфицируется в следующем синтаксисе:

```
REF IS column_name { SYSTEM GENERATED | USER GENERATED | DERIVED }
```

Эта спецификация не может входить в определение подтаблицы. Спецификация должна присутствовать в определении максимальной супертаблицы, и самоссылающийся столбец, определенный в максимальной супертаблице, наследуется любой ее подтаблицей. Семантика самоссылающихся столбцов обсуждается в следующем пункте.

Последней разновидностью элемента типизированной таблицы являются *опции столбцов* (column_options). Опции столбца можно указывать только для заново определенных столбцов — для унаследованных столбцов это не допускается. Соответствующая конструкция имеет следующий синтаксис:

```
column_name WITH OPTIONS ::= scope_clause
                           |default_clause
                           |column_constraint_definition_list
                           |collate_clause
```

Раздел scope_clause может входить в опции только заново определяемого столбца с типом REF (подробности в следующем подразделе). Для заново определяемого столбца некоторого типа символьных строк можно указать раздел collate_clause, чтобы задать желаемый порядок на соответствующем наборе символов. Если требуется указать значение столбца по умолчанию, отличное от значения по умолчанию соответствующего атрибута, ассоциированного с определяемой таблицей структурного типа, можно воспользоваться опцией default_clause. Наконец, для заново определяемого столбца можно указать одно или несколько ограничений, включая проверочные ограничения (см. лекцию 12).

Ссылочные значения и REF-типы

Понятия ссылочных значений и ссылочных (REF) типов являются, по существу, неразделимыми. В SQL:1999 ссылочный тип может использоваться в качестве типа данных столбцов обычных таблиц, атрибутов структурных типов, SQL-переменных и параметров – словом, везде, где можно использовать другие типы данных SQL. Значения местоположения ссылочного типа всегда являются ссылочными значениями строк типизированных таблиц (т. е. значениями самоссылающихся столбцов этих строк).

Для удобства повторим синтаксис спецификации ссылочного типа:

```
reference_type_specification ::= system_generated_representation
                               | user_defined_representation
                               | derived_representation
system_generated_representation :== REF IS SYSTEM GENERATED
user_defined_representation :== REF USING predefined_type
derived_representation ::= REF USING (commalist_of_attributes)
```

Механизмы генерации ссылочных значений

В SQL:1999 и SQL:2003 обеспечиваются три механизма назначения уникальных идентификаторов экземплярам структурных типов, ассоциированных с типизированными таблицами. Во всех типизированных таблицах, ассоциированных с данным структурным типом, должен использоваться один и тот же механизм. Предоставляются следующие альтернативы выбора ссылочных значений, которые могут являться:

- значениями некоторого встроенного типа SQL (user_defined_representation), которые должны генерироваться приложением каждый раз при сохранении экземпляра структурного типа как строки типизированной таблицы;
- значениями, порождаемыми из одного или нескольких атрибутов структурного типа;
- значениями, автоматически генерируемыми системой.

Как отмечалось в разделе «Определяемые пользователями типы», при определении любого максимального структурного супертипа явно или неявно задается спецификация ссылочного типа. Спецификация ссылочного типа наследуется всеми подтипами этого супертипа. При определении типизированных таблиц необходимо указать соответствующую спецификацию самоссылающегося столбца (конечно, эта спецификация логически избыточна, и, по всей вероятности, в следующих версиях стандарта SQL это требование будет ослаблено). Хотя соотношение

между альтернативами спецификации ссылочного типа и спецификации самоссылающегося столбца очевидно, приведем его явно (рис. 19.2).

reference_type_specification	self-referencing_column
REF USING predefined_type	USER GENERATED
REF FROM commalist_of_attributes	DERIVED
REF IS SYSTEM GENERATED	SYSTEM GENERATED

Рис. 19.2. Спецификации ссылочного типа и самоссылающегося столбца

Если для некоторого структурного типа выбран вариант пользовательской генерации ссылочных значений, то ответственность за поддержание уникальности таких значений лежит на пользователе. Конечно, ограничения PRIMARY KEY или UNIQUE, определенные на уровне максимальной супертаблицы семейства типизированных таблиц, могут гарантировать отсутствие в любой таблице этого семейства дублирующих ссылочных значений, но в SQL:1999 отсутствуют какие-либо средства, предотвращающие повторное использование ссылочных значений из удаленных строк в самоссылающихся столбцах новых строк.

Преобразование задаваемых пользователем ссылочных значений к ссылочному типу

В этом случае в определении структурного типа может присутствовать конструкция ref_cast_option (вернее, она должна присутствовать в определении соответствующего максимального супертипа). Синтаксис этой конструкции приводился в предыдущем разделе, но для удобства мы его повторим здесь:

```
ref_cast_option ::= cast_to_ref
                  | cast_to_type
cast_to_ref ::= CAST ( SOURCE AS REF ) WITH identifier
cast_to_type ::= CAST ( REF AS SOURCE ) WITH identifier
```

Чтобы пояснить эту конструкцию, предположим, что в определении структурного типа указано REF USING INTEGER. Тогда соответствующие приложения отвечают за то, чтобы обеспечить глобально уникальные целые значения самоссылающегося столбца во всех строках всех типизированных таблиц, ассоциированных с этим структурным типом. Но прило-

жения обеспечивают значения целого типа, а типом данных самоссылающегося столбца является некоторый ссылочный тип.

Для решения именно этой проблемы и предназначена конструкция ref_cast_option. В этой конструкции вводятся имена двух SQL-функций, первая из которых служит для преобразования ссылочных значений, обеспечиваемых приложением, к соответствующему REF-типу при вставке или обновлении строк типизированной таблицы (SOURCE AS REF). Вторая функция преобразует значения REF-типа к соответствующему встроенному типу данных при выборке строк из типизированной таблицы (REF AS SOURCE). Система автоматически генерирует обе подпрограммы, и конструкция ref_cast_option позволяет лишь назначить подпрограммам имена. Если конструкция ref_cast_option явно не включается в определение структурного типа с REF USING predefined_type, то имена подпрограммам назначаются системой. Единственным преимуществом явного назначения имен является возможность явного вызова этих функций при написании SQL-операторов, содержащих выражения REF-типа, которые нужно привести к соответствующему встроенному типу. Заметим, что такие функции невозможно написать вручную, поскольку правила отображения зависят от реализации SQL.

Если для структурного типа выбирается альтернатива порождения ссылочных значений, то система использует для порождения ссылочных значений значения неявно указанных столбцов (соответствующих явно указанным атрибутам ассоциированного структурного типа). При этом остаются все упомянутые выше проблемы, хотя в таком случае явно требуется объявление ограничений PRIMARY KEY или UNIQUE для соответствующего набора столбцов.

Наконец, при выборе последней альтернативы (системно-генерируемые ссылочные значения) каждой строке, вставляемой в типизированную таблицу, ассоциированную с соответствующим структурным типом, присваивается уникальный идентификатор. Это значение сохраняется в самоссылающемся столбце и может быть использовано любым приложением для уникальной идентификации данной строки на всем протяжении жизни таблицы.

Спецификация ссылочного типа при объявлении столбцов и атрибутов

Самоссылающиеся столбцы всегда имеют REF-тип. Конкретный REF-тип зависит от двух факторов:

- структурного типа, ассоциированного с типизированной таблицей: REF-тип всегда связан с некоторым структурным типом;

- выбранного способа генерации ссылочных значений; эта информация задается в определении структурного типа и не присутствует в спецификации ссылочного типа.

Для объявления местоположения ссылочного типа используется следующий синтаксис:

```
reference_type ::= REF (referenced_type) [ SCOPE table_name ]
referenced_type ::= UDT_name
```

UDT_name должно задавать имя типа (referenced_type), на экземпляры которого будут указывать значения ссылочного типа. REF-тип может использоваться в качестве типа атрибута структурного типа, и в этом случае referenced_type может быть тем же самым, что и определяемый структурный тип. Во все остальных случаях referenced_type должен являться некоторым существующим структурным типом.

В необязательном разделе SCOPE задается имя типизированной таблицы. Ассоциированным структурным типом этой таблицы должен быть referenced_type REF-типа, в спецификации которого содержится данный раздел SCOPE. Хотя можно было бы ожидать, что значение REF-типа можно использовать для ссылки на строки типизированных таблиц, ассоциированный структурный тип которых является собственным подтипом указанного referenced_type, в SQL такая разновидность ссылок не допускается. Ассоциированный структурный тип таблицы, на строки которой указывают значения REF-типа, должен быть в точности тем же, что и referenced_type этого REF-типа. Но, конечно, можно объявить REF-тип, у которого referenced_type является ассоциированным структурным типом подтаблицы, хотя самоссылающийся столбец этой подтаблицы необходимо наследуется от максимальной супертаблицы семейства таблиц.

Поддержка согласованности ссылок

Никакое ссылочное значение никогда не идентифицирует какую-либо строку, кроме той, с которой оно было ассоциировано с самого начала. Если эта строка удаляется, то значение ничего не идентифицирует и никогда не может быть связано с другой строкой. Из этого следует, что система должна каким-либо образом узнавать о том, идентифицирует ли данное ссылочное значение какую-то хранимую строку или ничего не идентифицирует (является висящей ссылкой). Но как система может это узнать, не потратив множество ресурсов? Отчасти здесь может помочь раздел SCOPE. В этом разделе указывается одна таблица, в которой строки должны существовать для всех значений данного местоположения, типом

данных которого является некоторый REF-тип. (В будущих версиях стандарта SQL, по всей видимости, будет разрешено указывать в разделе SCOPE список имен типизированных таблиц или даже использовать некоторую конструкцию, означающую «все таблицы, ассоциированные с данным структурным типом».)

Итак, если определяется столбец таблицы, поле строчного типа или атрибут структурного типа, и типом этого местоположения является REF-тип, то можно специфицировать раздел SCOPE. Однако если такой раздел действительно указывается, то требуется также указать, нужна ли проверка ссылочных значений. Для этого служит конструкция reference_scope_check, определяемая следующим синтаксическим правилом:

```
reference_scope_check ::= REFERENCES ARE [ NOT ] CHECKED
                          [ ON DELETE referential_action ]
```

Если указывается REFERENCES ARE NOT CHECKED или если раздел SCOPE не задается, то в определяемом местоположении можно хранить любое ссылочное значение, независимо от того, является ли оно значением самоссылающегося столбца какой-либо таблицы, на строку которой предположительно указывает ссылка. В этом случае система не гарантирует, что ссылочное значение действительно указывает на строку (но, конечно, это значение должно быть значением правильного типа – REF-типа указанного структурного типа).

Если же указывается REFERENCES ARE CHECKED, то каждый раз при сохранении значения в определяемом столбце, поле или атрибуте система обращается к указанной в разделе SCOPE таблице, чтобы убедиться в том, что в ней имеется строка, значение самоссылающегося столбца которой совпадает с сохраняемым ссылочным значением. Кроме того, если указывается REFERENCES ARE CHECKED, то можно также указать ссылочное действие, которое должно выполняться при удалении строки, идентифицируемой ссылочным значением. Как обычно (см. лекцию 12), возможными ссылочными действиями являются RESTRICT, CASCADE, SET NULL и NO ACTION. Если ссылочное действие явно не указывается, по умолчанию принимается NO ACTION. (Для поля строчного типа (ROW TYPE) и атрибута структурного типа допускается только NO ACTION.)

Заметим, что если раздел SCOPE включается в определение атрибута структурного типа, то в конструкции column_options столбца типизированной таблицы, соответствующего данному атрибуту, раздел SCOPE присутствовать не может – это считается синтаксической ошибкой.

Выборка данных из типизированных таблиц

Приведем несколько примеров операций выборки данных из типизированных таблиц, а также кратко обсудим операции обновления таких таблиц. Для этого сначала определим структурные типы EMP_T, PROGRAMMER_T и DEPT_T, а также соответствующие типизированные таблицы (упрощенный вариант).

```
CREATE TYPE EMP_T AS (
   EMP_NAME VARCHAR(20),
   EMP_BDATE DATE,
   EMP_SAL SALARY,
   DEPT REF (DEPT))
   INSTANTIABLE
   NOT FINAL
   REF IS SYSTEM GENERATED
   INSTANCE METHOD age ()
      RETURNS DECIMAL (3,1);
CREATE TYPE PROGRAMMER_T UNDER EMP_T AS (
   PROG_LANG VARCHAR (10))
   INSTANTIABLE
   NOT FINAL;
CREATE TYPE DEPT_T AS (
   DEPT_NO INTEGER,
   DEPT_NAME VARCHAR(200),
   DEPT_MNG REF (EMP))
   INSTANTIABLE
   REF IS SYSTEM GENERATED
   NOT FINAL;
CREATE TABLE EMP OF EMP_T
   (REF IS DEPT_ID SYSTEM GENERATED,
   DEPT WITH OPTIONS SCOPE DEPT);
CREATE TABLE PROGRAMMER OF PROGRAMMER_T UNDER EMP;
CREATE TABLE DEPT OF DEPT_T
   (REF IS EMP_ID SYSTEM GENERATED,
   DEPT_MNG WITH OPTIONS SCOPE EMP);
```

Следует отметить, что с типизированными таблицами можно работать, как с обычными таблицами*. Поэтому, в частности, возможен следующий запрос.

* По крайней мере, в той же синтаксической форме.

Пример 19.1. Найти имена всех служащих, размер заработной платы которых меньше 20000.00.

```
SELECT EMP_NAME
FROM EMP
WHERE EMP_SAL < 20000.00;
```

В соответствии с семантикой SQL:1999, при выполнении запроса из примера 19.1 сначала будет произведена выборка имен служащих, удовлетворяющих условию, из таблицы EMP, затем – из таблицы PROGRAMMER, и эти промежуточные результаты будут скомбинированы в окончательный результат путем применения операции объединения (UNION). Но предположим, что нас интересуют только те служащие, получающие зарплату, не превышающую 20000 руб., которые не являются программистами (**пример 19.2**). Тогда можно применить формулировку запроса, в которой присутствует спецификация ONLY:

```
SELECT EMP_NAME
FROM ONLY (EMP)
WHERE EMP_SAL < 20000.00;
```

Естественно, в запросах к типизированным таблицам можно использовать ссылки.

Пример 19.3. Найти имена и названия отделов, где работают служащие, размер заработной платы которых меньше 20000.00.

```
SELECT EMP_NAME, DEPT -> DEPT_NAME
FROM EMP
WHERE EMP_SAL < 20000.00;
```

В SQL:1999 операция «->» называется *операцией разыменования* (*dereferencing*), но в обиходе ее можно считать операцией перехода по ссылке (в нашем примере DEPT ссылается на DEPT_NAME). Можно неформально трактовать ссылочные значения как указатели на строки типизированных таблиц.

Может показаться неожиданным, что запрос из примера 19.3 выбирает значения из таблицы DEPT, хотя в разделе FROM этого запроса она даже не упоминается. Дело в том, что выполнение операции разыменования фактически приводит к выполнению соединения таблиц EMP и DEPT, делая в запросе столбец DEPT_NAME «видимым».

Конечно, в запросе допускаются многократные переходы по ссылкам, так что можно сформулировать следующий запрос:

Пример 19.4. Найти имена служащих и имена руководителей их отделов для служащих, получающих зарплату, не превышающую 20000.00.

```
SELECT EMP_NAME, DEPT -> DEPT_MNG -> EMP_NAME
FROM EMP
WHERE EMP_SAL < 20000.00;
```

Как показывает следующий пример, в запросах можно использовать вызовы методов над строками, к которым производится переход по ссылке.

Пример 19.5. Найти имя и возраст руководителя отдела 605.

```
SELECT DEPT_MNG -> EMP_NAME, DEPT_MNG -> age ()
FROM DEPT
WHERE DEPT_NO = 605;
```

Наконец, имеется возможность полностью выбрать экземпляр структурного типа, идентифицируемый ссылочным значением (в SQL:1999 это называется *разрешением ссылки – reference resolution*).

Пример 19.6. Получить полные данные о руководителе отдела 605.

```
SELECT DEREF (DEPT_MNG)
FROM DEPT
WHERE DEPT_NO = 605;
```

В этом случае результатом запроса будет являться таблица, включающая один столбец структурного типа EMP_T. Единственным значением этого столбца будет экземпляр (значение) этого структурного типа, соответствующий служащему-руководителю отдела 605.

Операции обновления типизированных таблиц выполняются очевидным образом. Операция INSERT вставляет указанные строки в указанную таблицу. Операции DELETE и UPDATE удаляют или модифицируют строки в иерархии таблиц, корнем которой является указанная таблица, если в операции не содержится ONLY. Если же специфицировано ONLY, то удаляются или модифицируются только строки указанной таблицы.

Типизированные представления

Наряду с типизированными базовыми таблицами в SQL:1999 поддерживаются типизированные представления, иначе называемые *представлениями, на которые можно ссылаться (referenceable views)*. Иногда такие представления также называют *объектными представлениями*, поскольку данные, видимые через представление, соответствуют строкам типизированных таблиц, поведение которых во многом похоже на поведение объектов в объектно-ориентированных системах. Между типизированными базовыми таблицами и типизированными представлениями имеется большое сходство, но есть и несколько отличий, связанных с различиями базовых таблиц и представлений.

В SQL в связи с объектными представлениями вводится ряд терминов – «*суперпредставление*», «*подпредставление*», «*непосредственное суперпредставление*», «*непосредственное подпредставление*», «*собственное суперпредставление*» и «*собственное подпредставление*». Смысл этих терминов полностью аналогичен смыслу соответствующих терминов для типизированных базовых таблиц. Термин *семейство подтаблиц* применяется как к типизированным таблицам, так и к типизированным представлениям.

Определение типизированного представления задается в следующей синтаксической форме:

```
view_definition ::= CREATE VIEW table_name
                        OF UDT_name
                      UNDER table_name
                      (view_element_commalist)
                    AS query_expression
                    [ WITH [ levels_clause ] CHECK OPTION ]
view_element ::= self_referencing_column_specification
                 | column_name WITH OPTIONS scope_clause
```

Указываемое `UDT_name` должно быть именем существующего структурного типа. Как и в определении обычных представлений, в разделе `AS` указывается выражение запроса. В случае типизированных представлений это выражение запроса должно основываться на единственной типизированной таблице (базовой таблице или представлении). Эта типизированная таблица должна быть ассоциирована с тем же структурным типом, что и определяемое представление. Такую таблицу иногда называют *базисной таблицей* представления.

Типизированное представление можно определить как подпредставление другого типизированного представления. В этом случае структурный тип, ассоциированный с определяемым представлением, должен яв-

ляться непосредственным подтипом структурного типа, который ассоциирован с суперпредставлением, специфицируемым в разделе UNDER. Базисная таблица определяемого представления должна являться собственной подтаблицей или собственным подпредставлением – не обязательно непосредственным – базисной таблицы непосредственного суперпредставления определяемого представления.

В определение типизированного представления может входить один или несколько элементов column_name WITH OPTIONS scope_clause. Если представление определяется как подпредставление другого типизированного представления, то в его определении не должна содержаться спецификация самоссылающегося столбца. Если определяется максимальное суперпредставление (т. е. в определении не содержится раздел UNDER), то эта спецификация может присутствовать. Если спецификация присутствует, то она может содержать только конструкции USER GENERATED или DERIVED (из этого следует, что нельзя определить типизированное представление, в ассоциированном структурном типе которого присутствует спецификация REF IS SYSTEM GENERATED). При указании USER GENERATED степень определяемого представления на единицу больше числа атрибутов ассоциируемого структурного типа; дополнительным столбцом является самоссылающийся столбец. В случае указания DERIVED дополнительный столбец не появляется, поскольку значение самоссылающегося столбца порождается из тех же столбцов, из которых порождается значение самоссылающегося столбца базисной таблицы.

Заключение

Если обратиться к истории, выяснится, что попытки расширения функциональности СУБД, изначально основанных на реляционном подходе, предпринимались уже на ранних стадиях разработки таких систем. Классическими примерами являются проекты System R компании IBM, где разработчики пытались обеспечить возможности работы со сложными объектами путем расширения SQL, и Ingres (университет Беркли), где Майкл Стоунбрейкер предлагал механизм определения пользовательских типов данных на основе представлений и хранимых процедур. Однако новый толчок к расширению SQL-ориентированных СУБД объектными свойствами был получен со стороны объектного мира после публикации *Первого манифеста*.

В ответном *Втором манифесте* представители индустрии развитых СУБД утверждали, что имеются реальные возможности добиться желаемой функциональности без коренной ломки традиционной технологии. Идеи *Второго манифеста* были воплощены в жизнь в нескольких ведущих SQL-продуктах, и использование объектных расширений позволило са-

мим поставщикам обеспечить ряд законченных функциональных расширений своих систем. Однако ожидания большого спроса со стороны пользователей на сами инструменты объектных расширений не оправдались. Некоторые известные специалисты из области баз данных считают, что для этого еще не пришло время.

Развитие объектно-реляционного подхода нашло отражение в языке SQL. Гигантский стандарт SQL:1999 позволяет хотя бы сопоставлять отдельные реализации, хотя ни одна компания полностью его не поддерживает. Как можно заметить, разработчики стандарта SQL пошли на существенно большее сближение с объектно-ориентированным подходом к организации систем баз данных, чем это предполагалось во *Втором манифесте*. В особенности это проявляется в механизмах типизированных таблиц, ссылочных типов и ссылочных значений: типизированные таблицы похожи на экстенты классов, а ссылочные значения – на объектные идентификаторы. Однако во многом это сходство является внешним – за путевыми выражениями в стиле ODMG по-прежнему скрываются операции соединения таблиц.

Данная лекция содержит весьма разнообразный материал, объединенный только общей идеей расширения РСУБД объектными возможностями. К сожалению, это вынужденное разнообразие, поскольку, на мой взгляд, большая часть расширений выполнялась без предварительной проработки не только общей модели, но даже и концепции языка. В результате мы можем оказаться в ситуации, когда язык SQL в лучшем случае будет полностью понятен только главному редактору стандарта.

И последнее замечание, на котором мы закончим этот курс. Несмотря на некоторую критику в адрес языка SQL, высказанную в начале лекции 11, мы потратили на обсуждение этого языка половину курса и больше его практически не критиковали. Не означает ли это, что язык все-таки очень хорош или что автор питает к нему особую привязанность? Конечно же нет! В языке SQL имеется множество слабых мест, неточностей и даже прямых ошибок. Если задаться целью продемонстрировать все промахи языка SQL, то этот курс никогда бы не закончился, а его читатели так и не узнали бы, что представляет собой язык в целом.

При всех недостатках у SQL имеются два неоспоримых преимущества. Во-первых, за 30 лет существования языка к нему привыкли (и даже сроднились с ним) тысячи профессионалов в области баз данных. Как говорится, лучше плохое, да свое. Во-вторых (и это проверено многолетней практикой) язык SQL допускает эффективную и масштабируемую реализацию, и даже объектные расширения языка не вызывают какой-либо деградации производительности систем. Одним словом, нам предстоит еще долгая совместная жизнь с SQL, и, чтобы она была удачной, нужно хорошо знать и достоинства, и недостатки этого языка.

Дополнительная литература

1. К. Дейт. Введение в системы баз данных. – 2-е изд.– М.: Наука, 1980.
2. К. Дейт. Введение в системы баз данных. – 6-е изд. – М.; СПб.: Вильямс, 2000.
   Обратите внимание, что я не рекомендую читать в связи с введением в реляционную модель данных седьмое издание этой книги Дейта, вышедшее на русском языке в издательстве «Вильямс» в 2001 г. По моему мнению, лучше всего реляционная модель была представлена в четвертом и пятом изданиях, которые на русском языке не издавались. Седьмое же издание мне менее всего нравится с методической точки зрения. В этих книгах Кристофера Дейта можно найти варианты реляционной алгебры, отличные от изложенных в данном курсе. Для полноты картины этот материал стоит изучить.
3. К. Дейт. Введение в системы баз данных. – 7-е изд. – М.; СПб.: Вильямс, 2001. – 8-е изд. – М.; СПб.: Вильямс, 2005.
   Пожалуй, материал по поводу проектирования реляционных баз данных путем нормализации лучше всего изложен Дейтом именно в седьмом и восьмом изданиях его основной книги. С моей точки зрения, он слишком сильно оценивает «научность» и строгость этой дисциплины, но это несущественно. Кроме того, при чтении соответствующих глав этих книг не следует обращать внимания на некоторые особенности изложения, проистекающие из Третьего манифеста, как и на погрешности перевода.
4. К. Дейт. Руководство по реляционной системе DB2. – М.: Финансы и статистика, 1988.
   С моей точки зрения, это очень хорошая книга. Она переведена М.Р. Когаловским. Было очень интересно проследить, как идеи System R реализовались в полноценном коммерческом продукте. Кроме того, тогда было странно, что компания IBM отказалась от некоторых удачных идей (например, тогда в DB2 не поддерживались триггеры).
5. К. Дейт, Хью Дарвен. Основы будущих систем баз данных. Третий манифест. – М: Янус-К, 2004.
   Авторы считают эту книгу просто изложением реляционной модели данных в современном понимании. Скромно замечу, что мне выпала честь переводить и редактировать перевод этой книги.
6. Д. Мейер. Теория реляционных баз данных. – М.: Мир, 1987.
   Это единственная переведенная на русский язык книга, посвященная лишь теоретическим вопросам проектирования реляционных баз данных. Я не слишком рекомендую читать эту книгу сразу целиком (если, конечно, удастся ее найти), поскольку написана она достаточно сум-*

* Научным редактором перевода является выдающийся российский математик и теоретик баз данных М.Ш. Цаленко.

бурно, но просмотреть ее полезно. Это позволит оценить стиль типичного теоретика баз данных и уровень сложности теории.

7. Вендров А.М. CASE-технологии. Современные методы и средства проектирования информационных систем. – М.: Финансы и статистика, 1998.
8. Вендров А.М. Проектирование программного обеспечения экономических информационных систем. – М.: Финансы и статистика, 2000.
   В книгах А.М. Вендрова дается широкий обзор современных технологий проектирования информационных систем. Автор руководствуется собственным практическим опытом, и в его изложении сглаживаются терминологические и концессионные барьеры между разными подходами. Кроме того, эти книги изначально написаны на русском языке, а не переведены с английского, поэтому читаются без затруднений.
9. Фаулер М., Скотт К. UML в кратком изложении. Применение стандартного языка объектного моделирования. – М.: Мир, 1999.
   Многие мои коллеги считают, что это лучшая книга про UML, переведенная на русский язык. Она написана четко, без повторов и философских отклонений от темы. Конечно, книга не может улучшить язык, но она помогает понять его в том виде, в каком он существует. Кстати, переводил книгу А.М. Вендров.
10. Буч Г. Объектно-ориентированный анализ и проектирование с примерами приложений на C++. – 2-е изд. – М.: Бином; СПб.: Невский диалект, 1999.
11. Буч Г., Рамбо Д., Джекобсон А. Язык UML: руководство пользователя. – М.: ДМК, 2000.
    Было бы просто неприлично не предложить в качестве дополнительной литературы по UML книги отцов-основателей этого языка. Хотя, честно говоря, я не в восторге от этих книг (как в русском переводе, так и на языке оригинала). Конечно, каждая книга полностью соответствует своему названию, но почему-то эти книги очень трудно и скучно читаются подряд, и в них очень трудно отыскать конкретную информацию.
12. М.Р. Когаловский. Энциклопедия технологий баз данных. – М.: Финансы и статистика, 2002.
    Одна из немногих книг, которые полезно всегда иметь под рукой специалистам в области баз данных.
13. Гектор Гарсиа-Молина, Джеффри Ульман, Дженифер Уидом. Системы баз данных. Полный курс. – М., С.-Петербург, Киев: Вильямс, 2003.
    Это действительно полный курс. Во многих университетах мира именно по этой книге читаются курсы по тематике баз данных.

Классические статьи в русских переводах

1. Кодд Э.Ф.. Реляционная модель данных для больших совместно используемых банков данных. – СУБД, № 1, 1995.
   http://www.osp.ru/dbms/1995/01/01.htm
   Первое широко доступное описание исходной реляционной модели, сделанное ее изобретателем.
2. Кодд Э.Ф. Расширение реляционной модели для лучшего отражения семантики. – СУБД, № 5, 1996.
   http://www.osp.ru/dbms/1996/05/163.htm
3. Злуф М. М. Query-by-Example: язык баз данных. – СУБД, № 3, 1996.
   http://www.osp.ru/dbms/1996/03/149_print.htm
4. Чен П.П. Модель «сущность-связь» – шаг к единому представлению данных. – СУБД, № 3, 1995.
   http://www.osp.ru/dbms/1995/03/271.htm
   Я очень рекомендую прочитать эту классическую статью, изданную впервые в 1976 г. Мне кажется, что она во многом проясняет процесс развития семантических диаграммных моделей.
5. Чамберлин Д.Д., Астрахан М.М., Эсваран К.П., Грифитс П.П., Лори Р.А, Мел Д.В., Райшер П., Вейд Б.В.
   SEQUEL 2: унифицированный подход к определению, манипулированию и контролю данных. – СУБД № 1, 1996.
   www.osp.ru/dbms/1996/01/48_print.htm
6. Аткинсон М. и др. Манифест систем объектно-ориентированных баз данных. — СУБД, № 4, 1995.
   http://www.osp.ru/dbms/1995/04/23.htm
7. Стоунбрейкер М. и др. Системы баз данных третьего поколения: манифест. — СУБД, № 2, 1995.
   http://www.osp.ru/dbms/1995/02/23.htm
8. Дарвин Х., Дейт К.. Третий манифест. — СУБД, № 1, 1996.
   http://www.osp.ru/dbms/1996/01/23.htm
   Это один из ранних вариантов (хотя и не самый ранний) Третьего манифеста. Насколько я помню, эта статья не произвела большого впечатления. Тогда мы еще наблюдали большой энтузиазм относительно системы объектно-ориентированных баз данных. Казалось, что они действительно смогут заменить системы реляционных баз данных.
9. Чемберлин Д. Анатомия объектно-реляционных баз данных. – СУБД, № 1-2, 1998.
   http://www.osp.ru/dbms/1998/01-02/003.htm

Неклассические статьи и другие материалы, доступные в Internet

1. Кузнецов С.Д. Операционная система UNIX.
   http://www.citforum.ru/operating_systems/unix/contents.shtml
   Свободно доступные материалы курса. Помимо прочего, можно ознакомиться с основными идеями операционной системы Multics и более подробно разобраться с организацией файловой системы ОС UNIX
2. http://www.multicians.org/
   Официальный сайт любителей ОС Multics. Здесь можно прочитать многие оригинальные статьи, написанные разработчиками системы во время реализации проекта.
3. Кузнецов С.Д. Третий манифест Дейта и Дарвена. – Открытые системы, № 4, 2000.
   http://www.osp.ru/os/2000/04/061.htm
   В этой статье пересказываются, обсуждаются и комментируются основные идеи, изложенные в книге Криса Дейта и Хью Дарвена «The Third Manifesto: Foundation for Future Database Systems».
4. Кузнецов С.Д. Третий манифест Кристофера Дейта и Хью Дарвена: немного формализма.
   http://www.osp.ru/os/2000/07-08/058.htm
   Это дополненный комментариями пересказ двух глав из книги Криса Дейта и Хью Дарвена «The Third Manifesto: Foundation for Future Database Systems». В частности, в статье содержится описание Алгебры A в авторском изложении.
5. Кузнецов С.Д. Основы современных баз данных. Лекция 8. Ingres: общая организация системы, основы языка Quel. http://www.citforum.ru/database/osbd/glava_32.shtml
6. Кузнецов С.Д. Развитие идей и приложений реляционной СУБД System R. http://www.citforum.ru/database/articles/art_27.shtml
7. Воссоединение SQL в 1995 г.: люди, проекты, политика. /Под ред. Пола МакДжонса, перевод Кузнецов С.Д.
   http://www.citforum.ru/database/digest/sql1.shtml
8. Кузнецов С.Д. Основы современных баз данных. Лекция 14. Стандартный язык баз данных SQL.
   http://www.citforum.ru/database/osbd/
   В этой главе материалов моего старого курса можно найти описание средств определения схемы, предусмотренных в стандарте SQL/89. Иногда бывает полезно сравнить старое и новое.
9. Чтобы получить исчерпывающую информацию о стандарте любого языка, нужно читать текст его стандарта. Это всегда непросто, поскольку текст любого стандарта представляет собой сухой технический документ. Это непросто и потому, что официально распространяемые

издания стандартов достаточно дорого стоят, и их нужно специально заказывать. Однако в случае языка SQL дела обстоят несколько более благоприятно. В Internet всегда можно найти тексты проектов следующего стандарта SQL, которые в данное время обсуждаются. В частности, на сайте http://www.wiscorp.com/SQLStandards.html можно найти проект стандарта SQL:200n, практически полностью включающий SQL:1999 и SQL:2003.

10. Журнальные публикации К. Дейта (DBPB, Intelligent Enterprise, сайт Фабиана Паскаля).
http://www.intelligententerprise.com/dbpdsearch.shtml
http://www.intelligententerprise.com/info_centers/database/date.shtml
www.dbdebunk.com
В Internet можно найти много публикаций Дейта, хотя все, что имеется в свободном доступе, датируется 1990-ми годами. Статьи последних лет доступны на сайте Фабиана Паскаля за умеренную плату.
11. Кузнецов С.Д. Дубликаты, неопределенные значения, первичные и возможные ключи и другие экзотические прелести языка SQL.
www.citforum/htdocs/database/articles/art_5.shtml
Эта небольшая статья была написана в то время, когда я понял, что потенциальное отсутствие возможного ключа в таблице, потенциальное наличие в таблице строк-дубликатов, наличие неопределенных значений и другие странные явления в языке SQL имеют общую природу.
12. Johnson Tom. The Fault with Defaults. Database Programming & Design On-Line, vol.11, № 2, February 1998,
www.dbpd.com/9802xtra.htm. (Имеется пересказ Кузнецов С.Д., www.citforum/htdocs/database/digest/dig_0402.shtml).
13& Кузнецов С.Д. Объектно-реляционные базы данных: прошедший этап или недооцененные возможности?
http://citforum.ru/database/articles/ordbms10/

Лучшие (непереведенные на русский язык) книги про SQL

На русском языке издано много книг по SQL (как переводных, так и написанных отечественными авторами). Но ни одна из них мне не нравится. Вот три книги, посвященные стандарту SQL, которыми пользуюсь я сам. К сожалению, на русский язык они не переведены.

1. Date C.J. with Darwen Hugh. A Guide to the SQL Standard. Fourth edition. Addison-Wesley Longman, 1997.
2. Melton Jim, Symon Alan R.. SQL:1999. Understanding Relational Language Components. Morgan Kaufmann Publishers, 2002
3. Melton Jim. Advanced SQL:1999. Understanding Object-Relational and Other Advanced Features. Morgan Kaufmann Publishers, 2003

Учебное издание

Кузнецов Сергей Дмитриевич

ОСНОВЫ БАЗ ДАННЫХ

Учебное пособие

Литературный редактор *Е. Петровичева*
Корректоры *Л. Зимилова, Ю. Голомазова*
Компьютерная верстка *Ю. Волшмид*
Обложка *М. Автономова*

Подписано в печать 31.07.2007. Формат 60x90 $^{1}/_{16}$.
Гарнитура Таймс. Бумага офсетная. Печать офсетная.
Усл. печ. л. 30,5. Тираж 2000 экз. Заказ № 5922

ООО «ИНТУИТ.ру»
Интернет-Университет Информационных Технологий, www.intuit.ru
Москва, Электрический пер., 8, стр.3.
E-mail: admin@intuit.ru, http://www.intuit.ru

ООО «БИНОМ. Лаборатория знаний»
Москва, проезд Аэропорта, д. 3
Телефон: (499) 157-1902, (499) 157-5272
E-mail: Lbz@aha.ru, http://www.Lbz.ru

При участии ООО «ЭМПРЕЗА»

Отпечатано в ОАО «ИПК «Ульяновский Дом печати»
432980, г. Ульяновск, ул. Гончарова, 14