

Технологии проектирования, разработки и внедрения баз данных в информационную систему предприятия

В книге обсуждаются роль и место баз данных в современных информационных системах, основные функции и архитектура СУБД, организация многопользовательского доступа к данным, обеспечение целостности данных, управление транзакциями, физическое хранение отношений, особенности построения индексов, основные черты коммерчески успешных моделей данных.

Рассматриваются жизненный цикл баз данных, технология проектирования реляционных баз данных на концептуальном, логическом и физическом этапах, базовые конструкции, используемые в SQL-ориентированных СУБД. Излагаются обязанности персонала, проектирующего и сопровождающего БД, требования пользователей к БД, особенности проектирования пользовательского интерфейса клиентских приложений, возможности интерактивной аналитической обработки данных OLAP, безопасность данных и способы противодействия угрозам, требования ГОСТ к документации БД.

Большое внимание уделяется перспективам развития баз данных, переход от централизованных к распределенным способам хранения данных, обсуждаются объектно-ориентированная и документ-ориентированная модели данных. Излагаются возможности языка XML для работы с слабо-структурированными данными.

Книга ориентирована на преподавателей и студентов ИТ-специальностей, а также на начинающих разработчиков БД.



Осипов Дмитрий Леонидович, кандидат технических наук, доцент кафедры прикладной математики и компьютерной безопасности института информационных технологий и телекоммуникаций Северо-Кавказского федерального университета. Специалист по информационным технологиям. Автор более 100 научных и методических работ, в том числе книг, посвященных разработке программного обеспечения и проектированию баз данных.

Интернет-магазин:
www.dmkpress.com

Оптовая продажа:
КТК «Галактика»
books@alians-kniga.ru



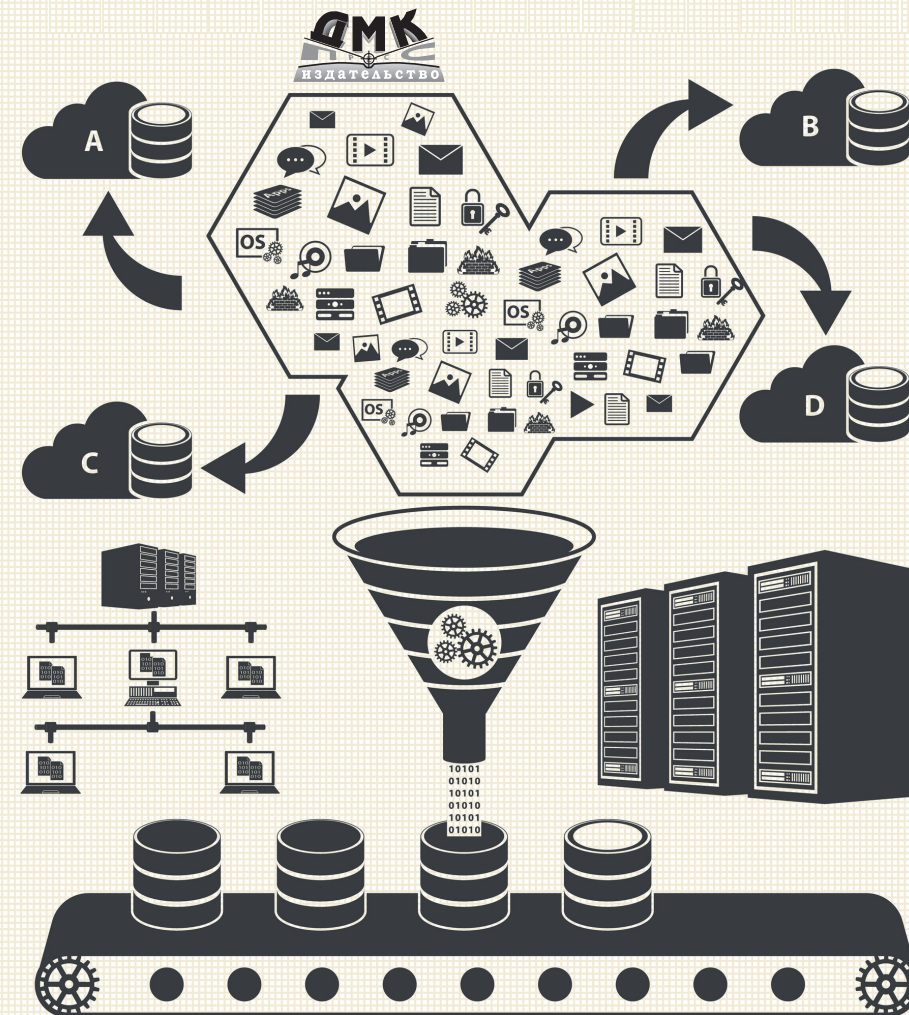
ISBN 978-5-97060-737-4



Технологии проектирования баз данных

Д. Л. Осипов

Технологии проектирования баз данных



Осипов Д. Л.

Технологии проектирования баз данных



Москва, 2019

УДК 004.65
ББК 32.972.134
074

074 Осипов Д. Л.

Технологии проектирования баз данных. – М.: ДМК Пресс, 2019. – 498 с.: ил.

ISBN 978-5-97060-737-4

Книга основана на материалах лекций и практических занятий, подготовленных автором и объединяет теоретические основы и практический аспект разработки современных баз данных (БД).

Основная задача издания — предоставить читателю профессиональную методику проектирования БД. Страницы книги проведут читателя по всем этапам жизненного цикла проекта баз данных от момента возникновения идеи разработки программного обеспечения до этапа ввода готового продукта в эксплуатацию, подробно объясняя каждый шаг.

Издание отличается глубиной и ясностью изложения материала, поэтому издание окажется полезным как для студентов и преподавателей ИТ-специальностей, так и для разработчиков баз данных и программистов, стремящихся самостоятельно освоить технологические приемы проектирования современных БД.

УДК 004.65
ББК 32.972.134

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

Материал, изложенный в данной книге, многократно проверен. Но, поскольку вероятность технических ошибок все равно существует, издательство не может гарантировать абсолютную точность и правильность приводимых сведений. В связи с этим издательство не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-5-97060-737-4

© Осипов Д. Л., 2019
© Оформление, издание ДМК Пресс, 2019

Оглавление

Введение	12
Соглашения.....	13
Глава 1. Эволюция баз данных	14
Электронные картотеки.....	16
Принцип построения систем файлов.....	17
Недостатки систем файлов	18
Пути устранения недостатков систем файлов.....	22
Что такое база данных?.....	23
Эволюция моделей БД.....	24
Необходимость моделирования	26
Иерархическая модель.....	27
Сетевая модель.....	30
Попытки разработки стандарта БД.....	32
Реляционная модель.....	34
Объектно-ориентированная модель	36
Слабоструктурированные данные	38
Документ-ориентированная модель	38
Резюме.....	39
Вопросы для самопроверки	39
Глава 2. Система управления базами данных.....	41
Функционал СУБД.....	42
Компоненты СУБД.....	45
Системный каталог	48
Архитектурные решения доступа к БД	48
Файл-сервер.....	49
Клиент-сервер	50
Распределенная система.....	54
Резюме.....	55
Вопросы для самопроверки	56
Глава 3. Персонал и пользователи БД.....	57
Администратор данных	59
Администратор базы данных.....	60
Разработчики баз данных.....	61
Прикладные программисты.....	61

Конечные пользователи.....	62
Резюме.....	63
Вопросы для самопроверки	63
Глава 4. Реляционная модель	65
Сущность и атрибуты.....	66
Тип данных и домен.....	69
Связь.....	71
Отношение.....	73
Ключи.....	76
Целостность данных.....	77
Целостность доменов	78
Целостность сущностей.....	79
Ссылочная целостность.....	80
Корпоративная целостность	80
Реляционная алгебра	81
Резюме.....	87
Вопросы для самопроверки	88
Глава 5. Технология разработки БД	89
Роль БД на предприятии	90
Жизненный цикл базы данных	94
Этап планирования разработки БД.....	96
Этап определения и анализа требований к системе	96
Этап проектирования БД	100
Этап выбора СУБД	104
Этап создания клиентского программного обеспечения.....	105
Этап тестирования и отладки.....	107
Этап реализации.....	109
Этап эксплуатации и сопровождения.....	110
Резюме.....	111
Вопросы для самопроверки	111
Глава 6. Концептуальное проектирование и ER-модель.....	112
Концептуальная модель БД	113
ER-модель.....	113
Типы сущностей и атрибуты.....	114
Связи в ER-модели.....	119
Вариации ER-моделей.....	127
Резюме.....	129
Вопросы для самопроверки	130

Глава 7. Логическое проектирование и нормализация	131
Первая нормальная форма	134
Функциональная зависимость атрибутов	137
Порядок определения первичного ключа	139
Вторая нормальная форма	141
Третья нормальная форма	142
Нормальная форма Бойса-Кодда	145
Четвертая нормальная форма	145
Пятая нормальная форма	147
Финал «гонки» нормальных форм	149
Резюме	149
Вопросы для самопроверки	150
Глава 8. Физическое представление данных	151
Двухуровневая модель хранения данных	151
Представление реляционных данных	153
Поля	154
Записи	156
Блоки	157
Файл	157
Модификация записей	158
Особенности представления объектов	158
Журнальная информация	159
Резюме	160
Вопросы для самопроверки	160
Глава 9. Индексирование	162
Индексы на основе хеширования	164
Индексы на основе В-деревьев	168
Битовые индексы	171
Правила назначения пользовательских индексов	172
Избирательность индекса	173
Резюме	175
Вопросы для самопроверки	175
Глава 10. Безопасность данных	176
Откуда исходят угрозы?	178
Политика безопасности	181
Правила защиты БД	182

Идентификация, аутентификация и авторизация.....	184
Криптографическая защита.....	185
Резервное копирование и восстановление.....	188
Аудит событий безопасности.....	189
Модернизация программного обеспечения.....	191
Безопасный доступ к данным.....	191
Экономическая оправданность.....	192
Резюме.....	192
Вопросы для самопроверки.....	193
Глава 11. Знакомимся с SQL.....	194
Возможности SQL.....	197
Типы данных SQL.....	198
Предопределенные типы.....	199
Непредопределенные типы.....	206
Константы.....	209
Преобразование данных.....	210
Операторы.....	212
Операция присваивания.....	213
Арифметические операторы.....	213
Логические операторы.....	214
Операторы сравнения.....	215
Проверка на неопределенность NULL.....	215
Конкатенация строк.....	216
Встроенные функции.....	216
Резюме.....	217
Вопросы для самопроверки.....	217
Глава 12. Манипулирование данными SQL.....	219
Запрос, инструкция SELECT.....	219
Псевдонимы имен столбцов и таблиц.....	222
Порядок сортировки, ORDER BY.....	223
Условие отбора данных, предложение WHERE.....	224
Агрегирующие функции.....	232
Группировка данных GROUP BY.....	233
Соединение таблиц в запросе SELECT.....	234
Вставка, инструкция INSERT.....	244
Модификация, инструкция UPDATE.....	246
Удаление, инструкция DELETE.....	248
Слияние данных, инструкция MERGE.....	249
Резюме.....	250
Вопросы для самопроверки.....	251

Глава 13. Определение данных средствами SQL	252
Базы данных (схемы)	252
Домены	255
Таблицы	256
Внешние ключи и связи между таблицами	258
Ограничения на значения столбцов	260
Столбец-перечисление	261
Столбец-множество	262
Временные таблицы	263
Модификация таблицы	264
Клонирование и копирование таблиц	265
Индексы	266
Изменение индекса	267
Удаление индекса	268
Представления	269
Изменение представления	272
Удаление представления	273
Модифицируемые представления	273
Резюме	274
Вопросы для самопроверки	275
Глава 14. Процедурный SQL	276
Элементы процедурного SQL	278
Переменные	278
Составной оператор BEGIN..END	281
Условные операторы	281
Циклы	284
Хранимые процедуры и функции	288
Вызов хранимой процедуры	291
Особенности работы с функциями	292
Изменение процедур и функций	294
Удаление процедур и функций	294
Триггеры	294
Контекстные переменные	297
Примеры триггеров	297
Курсоры	302
Примеры курсоров	305
Резюме	309
Вопросы для самопроверки	310
Глава 15. Регулярные выражения в запросах	311
Операторы для регулярных выражений	311

Основы синтаксиса.....	312
Регулярные выражения в запросах.....	319
Резюме.....	320
Вопросы для самопроверки	320
Глава 16. Управление транзакциями	322
Требования к транзакции	323
Состояние транзакции.....	324
Проблемы совместного доступа к данным	325
Управление параллельными транзакциями.....	326
Пессимистический подход	327
Оптимистический подход	330
Детализация уровня блокировок	333
Требования стандарта SQL.....	334
Явное управление транзакцией.....	335
Точки сохранения.....	338
Резюме.....	338
Вопросы для самопроверки	339
Глава 17. Определение прав пользователей	340
Идентификатор авторизации.....	341
Объекты защиты	342
Управление наборами привилегий	343
Предоставление привилегий	344
Лишение привилегий	347
Резюме.....	349
Вопросы для самопроверки	349
Глава 18. Интерактивная аналитическая обработка OLAP	350
Требования к OLAP-инструментам	351
Хранилище данных.....	353
OLAP-куб.....	355
Язык многомерных выражений MDX.....	356
Резюме.....	358
Вопросы для самопроверки	358
Глава 19. Расширяемый язык разметки XML	360
Корректность документа XML.....	361
Построение документа XML	361
Элементы документа	362
Атрибуты	363

Пространство имен.....	364
Определение типа документа DTD	367
XML Schemas.....	373
Элементы схемы	376
Атрибуты схемы	385
Подключение XML-схемы к документу.....	386
Поддержка XML в СУБД	386
Резюме.....	387
Вопросы для самопроверки	388
Глава 20. Клиент-серверные БД.....	389
Модель взаимодействия открытых систем.....	389
Клиент-серверные СУБД	393
Модели распределения функций	393
Резюме.....	396
Вопросы для самопроверки	397
Глава 21. Особенности разработки клиента БД.....	398
Выбор языка программирования.....	398
Технология доступа к данным ODBC.....	399
Технология доступа к данным ADO .NET	400
Технология доступа к данным FireDAC.....	402
Технология JDBC	404
Интерфейс клиента.....	405
Сколько людей, столько и мнений	406
Пользовательские критерии качества интерфейса	407
Рекомендации по проектированию	408
Резюме.....	410
Вопросы для самопроверки	411
Глава 22. Распределенные БД	412
Предпосылки децентрализации	412
Система управления распределенной базой данных.....	414
Правила распределенных БД от Криса Дейта	415
Аспекты проектирования распределенных БД	416
Фрагментация.....	417
Распределение	419
Репликация	419
Особенности управления системным каталогом.....	420
Распределенные транзакции	420
Преимущества распределенных БД.....	421

Недостатки распределенных БД.....	422
Резюме.....	423
Вопросы для самопроверки	424
Глава 23. Объектно-ориентированная модель данных.....	425
Предпосылки появления модели	425
Преимущества ООБД.....	427
Объектно-ориентированная терминология	428
Абстрагирование.....	430
Инкапсуляция.....	430
Модульность.....	431
Наследование.....	432
Идентификатор объекта	432
Манифест объектно-ориентированных СУБД.....	433
Стандарт ODMG.....	437
Что было сделано на практике?.....	437
Postgres	437
UniSQL.....	438
Cache	439
Versant Object Database.....	439
ObjectStore.....	440
Что пошло не так?.....	440
Недостатки ООБД.....	441
Объектно-реляционные СУБД.....	443
Резюме.....	444
Вопросы для самопроверки	445
Глава 24. Документ-ориентированные БД.....	446
Чем плоха нормализация?.....	446
БД ключ-значение	447
Документ-ориентированные БД.....	448
NoSQL.....	449
Распределенная обработка MapReduce	452
Сегментирование	453
Репликация	455
Когда следует использовать документ-ориентированную модель?	456
Резюме.....	456
Вопросы для самопроверки	457
Глава 25. Большие данные	458
Что такое «большие данные»?.....	459

Принципы работы с большими данными	461
Лямбда-архитектура	461
Apache Hadoop	463
Apache Storm	465
Apache Impala	466
Apache Kafka	466
NewSQL	467
Добыча данных	468
Резюме	469
Вопросы для самопроверки	470
Глава 26. Составление программной документации	471
Виды программных документов	472
Техническое задание	473
Пояснительная записка	475
Эксплуатационные документы	476
Руководство системного программиста	477
Руководство оператора	478
Документация в тексте программы	479
Резюме	480
Вопросы для самопроверки	481
Приложение 1. Модель БД «Склад»	482
Приложение 2. Пример XML-схемы	483
Приложение 3. Стандарты по единой системе программной документации	487
Список литературы	489
Предметный указатель	493

Введение

Вряд ли сегодня кому-то удастся назвать современную область знаний, в которой не нашли бы применения компьютерные базы данных (БД). Наука, образование, экономика, электронная коммерция, медицина, статистика, военное дело – список можно продолжать очень долго. Базы данных сопровождают человека на протяжении всей жизни. Появление на свет ребенка, учеба в школе и вузе, получение паспорта или водительских прав, посещение врача, покупки в магазинах, поиск книги в библиотеке, открытие счета в банке – в современном информационном обществе ни одно из этих и многих других событий не обходится без появления очередной электронной пометки в памяти многочисленных компьютеров.

Базы данных входят в десятку самых востребованных программных продуктов и служат источником неплохого заработка для профессиональных разработчиков. Судите сами, без хранения и учета данных сегодня обойтись весьма сложно. Многочисленные магазины, склады, страховые агентства, отделы кадров, бухгалтерии, учебные заведения и множество других предприятий и организаций остро нуждаются в разработанных специально для них БД. И спрос все еще превышает предложение.

Создать эффективную базу данных весьма непросто, даже если она предназначена для обслуживания незначительных объемов данных и подлежит эксплуатации на домашнем компьютере. Сложность проекта возрастает на порядок, когда возникает задача разработать жизнеспособный коммерческий продукт, с которым смогут одновременно работать десятки пользователей. Именно поэтому главная задача книги – вооружить читателя знаниями о технологии проектирования баз данных. Здесь вы подчерпнете всю необходимую информацию о:

- задачах, решаемых с помощью БД;
- архитектуре систем управления базами данных (СУБД);
- реляционной, объектно-ориентированной, документ-ориентированной (и ряде других) моделях данных;
- жизненном цикле проектов БД, этапах концептуального, логического и физического проектирования БД;
- особенностях физического хранения и правилах индексирования данных;

- многопользовательском доступе к данным и управлении транзакциями;
- особенностях организации доступа к БД и обеспечении безопасности данных;
- структурированном языке запросов SQL;
- применении в SQL регулярных выражений;
- расширяемом языке разметки XML;
- многомерном анализе данных;
- централизованных и распределенных БД;
- правилах разработки клиентских приложений БД;
- особенностях документирования проектов БД.

Самое главное, что получит читатель после изучения предложенного материала, – владение методологией работы с любой БД. Книга не ограничивает читателя в вопросе выбора целевой СУБД, поэтому ваша база данных может быть развернута как на основе простейших настольных систем, так и на фундаменте профессиональных многопользовательских клиент-серверных программных решений.

Соглашения

Для акцентирования внимания читателя на ключевых частях излагаемого материала такой текст отмечен особым форматированием:

Замечание

Таким способом в тексте книги выделен материал, который вы должны принять к сведению. Обычно это определения, комментарии или замечания.

Кроме того:

- впервые встречающиеся термины и определения выделены **полужирным шрифтом**;
- впервые встречающиеся аббревиатуры комментируются;
- код примеров, синтаксические конструкции даны моноширинным шрифтом;
- помимо содержания, книга включает подробный предметный указатель, позволяющий найти в тексте интересующую читателя информацию.

Глава 1

Эволюция баз данных

Очень сложно привести пример какой-либо области науки и техники, сделавшей за последние полстолетия столь же значимый рывок в своем развитии, как современная микроэлектроника и идущие с ней рука об руку информационные технологии (ИТ). Более того, ИТ развиваются столь динамично, что даже специалистам в этой области приходится едва ли не каждые 5–6 лет кардинально переучиваться, дабы успеть попасть в последний вагон улетающего вдаль с космической скоростью экспресса технологий разработки программного обеспечения.

Как человек сумел за столь короткий срок совершить столь большие шаги в области ИТ? Чтобы ответить на этот вопрос, стоит разобраться с тем, что происходило в те «старозаветные» времена, в которых компьютер упоминался лишь в произведениях писателей-фантастов. Для этого нам предстоит отмотать временную ленту всего на несколько десятилетий назад и попасть в эпоху, когда на смену механическому арифмометру стали приходиться первые вычислительные машины. Именно этот момент времени мы и станем считать началом всех начал для современных информационных технологий в целом и зарождающихся баз данных (БД) в частности.

Для полного погружения в эпоху нарисуем картину научно-вычислительной лаборатории 50-х годов XX века. Посреди огромного зала всеми цветами радуги переливается огромный ламповый монстр – электронно-вычислительная машина. А вокруг нее вьется рой инженеров, математиков и программистов. Инженеры меняют радиодетали, математики выдумывают формулы, а программисты замыкают круг – на основе предложенных математиками формул закладывают в ЭВМ программы, чтобы последние жгли электронные лампы. Огромное количество обслуживающего персонала, трудящегося во благо научно-технического прогресса, в некоторой степени роднило обслуживание ЭВМ с ходом

возведения Вавилонской башни (как с точки зрения процесса, так и по результату).

Технология разработки прикладного программного обеспечения тех времен был достойна кисти Сальвадора Дали. Наивный заказчик расчетов приходил к математику и просил, чтобы машина ответила – сколько, на ее взгляд, будет равняться $2 + 2$. Математик-алгоритмист в глубокой задумчивости рисовал алгоритм и передавал его программисту, перекладывающему алгоритм на низкоуровневый язык машинных команд. Для того чтобы ЭВМ смогла усвоить программу, последняя набивалась техниками на перфоленту или на перфокарты. Данные полдня загружались в память машины, затем она пару-другую раз зависала, шипела, кряхтела и, наконец, к всеобщей радости, выдавала распечатку. К этому времени результат уже никого не интересовал... Но согласитесь – сколь увлекателен сам процесс!

К 60-м годам прошлого века ЭВМ подверглись усовершенствованию настолько, что уже были способны не только воодушевлять писателей-фантастов и согревать воздух, но и производить некоторые полезные операции, в первую очередь связанные с утомительными математическими расчетами. Вас интересует таблица синусов с точностью до 18 знаков после запятой? Теперь уже не надо напрягать свой мозг – обращаемся за помощью в научно-вычислительную лабораторию, и всего лишь через час распечатка у вас в кармане. И не важно, что вместо синусов вам посчитали косинусы: главное – никаких умственных затрат, да и точность соблюдена!

Повысилась не только производительность процессоров, но и на качественно другой уровень перешли периферийные устройства. Теперь почти пропала необходимость вставки между пользователем и машинной гильдии разношерстных посредников. Человек, имеющий некоторое представление о способах общения с машиной, просто садился за терминал и получал возможность самостоятельно, без какой-либо посторонней помощи вгонять ЭВМ в ступор. Пользователи – люди разные, и далеко не всех интересовали расчеты траекторий баллистических ракет. Огромный пласт людей искал применения для машин в другой не менее важной области – области хранения и обработки больших массивов данных. Судите сами: XX век – век информации, в офисах корпораций, в правительственных учреждениях, в архивах и библиотеках хранились миллионы тонн бумаги с данными о чем угодно, начиная с роста поголовья пингвинов в Антарктике и заканчивая налоговыми декларациями от горячо любимых сограждан. Все это необходимо не просто хранить, а еще и максимально быстро обработать с возможно-

стью получения аналитических выводов, графиков и статистики. До сих пор все эти попытки систематизации колоссальных объемов данных, представленных на бумажных носителях, были в некотором родстве с сизифовым трудом. И вдруг, о чудо, наконец у человечества появился шанс вкатить камень на вершину горы!

Именно с этого момента в сферу обязанностей вычислительных машин, кроме проведения расчетов, вменили еще одну задачу – хранение и обработку больших объемов данных. Тем более что на смену перфолен-там и стримерам с магнитной пленкой стали приходить жесткие диски.

Определение

Данные (data) – информация, представленная в формализованном виде, пригодном для передачи, интерпретации или обработки с участием человека или автоматическими средствами.

Только бесконечно наивный пользователь (метко называемый в народе чайником) может предположить, что ЭВМ сразу же подставила человечеству свое плечо и в мановение ока все бумажные архивы превратились в во всех отношениях совершенные базы данных. Не тут-то было! Вычислительные машины – это лишь инструмент, который работает ровно так, как его научат программисты. Поэтому примерно на стыке 50-х и 60-х годов XX века перед программистами была поставлена задача научить машины обслуживать большие объемы данных.

Электронные картотеки

Мы с вами все учились сами, а некоторые из нас даже пытались учить других. Любой участник образовательного процесса, находящийся в здравом уме и доброй памяти, понимает, что научить можно только тому, что в совершенстве знаешь сам. А теперь ответьте на вопрос: «Что в начале 60-х программисты знали о хранении больших данных?» Прав окажется тот, кто скажет – практически ничего! Так что нет ничего удивительного в том, что на первых этапах становления такой области знаний, как базы данных, все пошло по особому пути, который получил название **систем, основанных на файлах** (file-based system), или просто **системы файлов**. К чему столько скепсиса? Поймете через пару страниц, а пока рассмотрим историю болезни прототипов современных БД – систем файлов.

Особенность человеческого образа мышления заключается в том, что при поиске решений сложных проблем в первую очередь мы пыта-

емся применить уже известные методики. Примеров такого подхода в истории предостаточно. Например, разработчики первых летательных аппаратов весьма настойчиво пытались обучить их махать крыльями – ведь именно так делали птицы. Эффективность подобного решения новаторы обычно проверяли сами, поэтому долго не жили. В большинстве своем программисты – также народ прямолинейный. Создатели прообразов баз данных при поиске решения огляделись вокруг и увидели, что везде, где есть бумажные архивы, данные хранятся в картотеках. Возьмем обычную городскую библиотеку тех лет. Здесь каждой книге соответствует отдельная бумажная карточка, в которой отражены данные об авторе, названии книги, годе издания, месте хранения и т. д. Карточки систематизированы по областям знаний и упорядочены по алфавиту. Любой грамотный посетитель, придя в библиотеку, за пару-тройку часов (перелопатив с тысячу карточек) выяснял, что интересующей его книги там нет, и с гордо поднятой головой уходил восвояси... Плохо это или хорошо, но на тот момент времени другого, более рационального решения проблемы архивного дела не существовало. Поэтому воодушевленные программисты, не теряя ни одной секунды на «лишние» размышления, самоотверженно приступили к обучению самолетов махать крыльями – взялись за лобовое проектирование электронных аналогов бумажных картотек.

Замечание

Почему системам, основанным на файлах, мы уделяем столько времени и не переходим сразу к изучению баз данных? По двум причинам. Во-первых, понимание сути недостатков, присущих файловым системам, позволяет избежать их в дальнейшем. Во-вторых, даже сегодня в спектре программного обеспечения есть место для приложений, построенных на основе файлов.

Принцип построения систем файлов

Рассмотрим в общих чертах идею построения систем, основанных на файлах. Допустим, что мы планируем создать программу, хранящую сведения о сотрудниках предприятия. Получивший столь ответственное задание программист поступает следующим образом.

Во-первых, он готовит структуру, подходящую для хранения данных. Для этого программист просматривает список персонала и выясняет максимальное число символов в фамилии, имени и отчестве (допустим, это значения: 20–15–15 байт). Затем программист выделяет ка-

кое-то число байтов для хранения названия должности, даты рождения и остальных подлежащих учету элементов структуры. Узнав все необходимые размерности, разработчик описывает структуру непосредственно в коде программы (рис. 1.1).

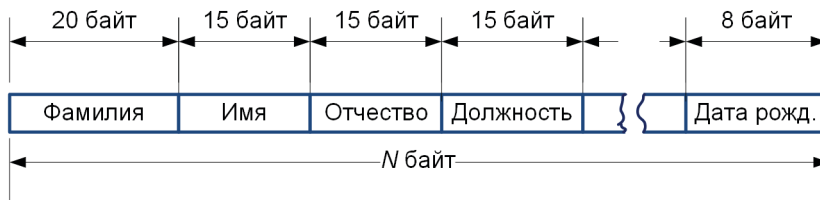


Рис. 1.1. Структура для хранения одной записи в файле

Во-вторых, программистом разрабатываются процедуры, осуществляющие основные операции по работе с файлом. Как минимум это добавление новой записи, редактирование, удаление и просмотр записи. Ни одну из этих операций невозможно осуществить без знания размерности и состава полей исходной структуры. При вставке новой строки в файл к нему следует добавить N байт, причем процедура добавления должна знать, что фамилия начинается с 1-го байта, имя – с 21-го и т. д. При просмотре файла необходимо осуществлять последовательные операции чтения порциями, пропорциональными размеру отдельной записи N байт; опять же, процедура чтения должна обладать информацией – сколько байт отводится тому или иному полю. В свою очередь, операции редактирования и удаления также не являются исключением из правил и нуждаются в знаниях об исходной структуре. К счастью, эти сведения искать не стоит – все данные о составе полей нашей программе хорошо известны, ведь определение структуры спрятано внутри нее.

Недостатки систем файлов

Поначалу у разработчиков систем, основанных на файлах, дела шли весьма неплохо. Пользователям очень нравилось, что работа с электронными картотеками была схожа с работой с бумажными архивами. В свою очередь, программистам нравилось то, что они сравнительно легко зарабатывают себе на жизнь.

Идиллия продолжалась недолго. Очень скоро, казалось на безоблачном горизонте, забрезжили грозовые тучи – стали проявляться отрицательные стороны лобового подхода разработчиков первых прототипов баз данных. Из всего сонма недостатков особо выделяются пять проблем [25]:

- 1) зависимость от данных;
- 2) разделение и изоляция данных;
- 3) избыточность данных;
- 4) несовместимость файлов;
- 5) разрастание количества приложений.

Зависимость от данных. Уже первая проблема, с которой столкнулись разработчики файловых систем, не предвещала ничего хорошего. Допустим, что нам требуется осуществить элементарную операцию – увеличить на один символ размер поля, отвечающего за хранение фамилии. Оказалось, что даже незначительное усовершенствование структуры влечет за собой ком дополнительных трудностей. Судите сами, изменение размера или состава полей структуры вынуждает нас не только переписать исполняемый файл, но и сочинить одноразовую программу-конвертор, которая должна преобразовать старые данные к новому формату.

Разделение и изоляция данных. Системы, основанные на файлах, объединяют в себе десятки отдельных файлов с данными. В одном файле хранятся данные о сотрудниках фирмы, в другом – сведения о клиентах, в третьем – перечень предоставляемых услуг, в четвертом – список заказов и т. д. Для извлечения логически связанных данных (допустим, о клиентах и их заказах) программисту приходилось выдумывать замысловатые алгоритмы синхронного чтения из двух файлов. С увеличением количества файлов, вовлекаемых в итоговый отчет, сложность возрастает в арифметической прогрессии. Задача извлечения взаимосвязанных данных из десятка файлов могла стать непосильной не только для программиста, но и для вычислительных машин тех времен. Масло в огонь подливало еще то, что данные могли быть разделены между отделами и службами предприятия – в отделе кадров находились данные о сотрудниках, в отделе продаж – о заказах и т. п. В 1960-х годах удельный вес предприятий, чьи машины были объединены локальными вычислительными сетями, стремился к нулю, посему данные были еще и изолированы друг от друга. А теперь представьте себе программиста тех времен, мечущегося по организации в надежде объединить разделенные данные в единое целое...

Избыточность (дублирование) данных. С появлением первой микропроцессорной техники многие руководители предприятий стали отказываться от покупок больших ЭВМ и начали отдавать свои предпочтения более дешевым мини-ЭВМ, расставляя их по отделам и службам

своих организаций. Подобное, во многом правильное решение имело и свои отрицательные стороны, одна из них – вынужденный отказ от централизованного хранения данных. Система децентрализованного хранения данных систем, основанных на файлах на нескольких машинах, приводила к тому, что одни и те же сведения повторялись на магнитных носителях многих мини-ЭВМ, разбросанных по учреждению. В этом случае даже был не столь страшен факт избыточности данных, сколь высока вероятность нарушения непротиворечивости данных предприятия – на всех машинах должны храниться идентичные копии данных, но ведь любое изменение данных на одной из ЭВМ никак не отражалось на остальных станциях до тех пор, пока данные не синхронизировали вручную.

В «доисторических» системах файлов избыточность данных вынужденно присутствовала и в рамках одного-единственного проекта. Допустим, что от нас потребуют дополнить программу «Дни рождений сотрудников» еще одним информационным полем – местом работы. В результате в файле появятся многократные дубликаты данных, например Петров – Бухгалтерия, Иванов – Бухгалтерия и т. д. Исследования файловых систем тех времен показали, что до 60 % хранящейся в них данных были избыточны. Учитывая астрономическую стоимость жестких дисков, это весьма непродуктивные расходы.

В свою очередь, избыточность данных порождала целый букет проблем и проблемок. Одна из них – **противоречивость данных**. На одной из рабочих станций предприятия хранится устаревший номер телефона вашего контрагента. На второй этого номера вовсе нет. На третьем компьютере, за счет ошибки оператора, там находится телефон его бабушки. В результате ни один из звонков не достигает цели. Как следствие, фирма терпит убытки.

Аномалии данных. Избыточные и противоречивые данные влекут за собой шлейф дополнительных неприятностей в лице: аномалий добавления новой записи, аномалий редактирования и аномалий удаления. Какая из аномалий способна принести больше печали в наш офис? Судите сами. Допустим, у нашей фирмы появился новый, не жалеющий денег оптовый покупатель. Данные нового клиента следует ввести сразу в несколько систем файлов (отдел сбыта, личная картотека главного менеджера, бухгалтерия и т. д.). Если все сделано безошибочно, то все в порядке... но если таких покупателей несколько, то можно гарантировать, что где-нибудь кто-нибудь спутает пару цифр в номерах счетов. В результате платеж уходит на чужой расчетный счет. После долгого

судебного разбирательства и уплаты неустоек вы, наконец, выясняете, в чем причина сбоя, но к этому времени вам уже все равно... С редактированием данных в избыточных системах дела также обстоят далеко не лучшим образом. В идеале можно нанять отдельного сотрудника, задачей которого станет регулярная пробежка по всем структурным подразделениям компании с целью исправить почтовый адрес (номер телефона, дату рождения, номер счета или что-нибудь в этом духе) главного спонсора фирмы. В результате поздравительная открытка, отправленная в канун очередного юбилея, не попадет к адресату, а в отместку «благодарный» юбиляр не перечислит вашей компании давно обещанные (и так необходимые сейчас) финансовые влияния. Впрочем, считайте, что вам крупно повезло, ведь уязвленное самолюбие может привести и к более серьезным последствиям... Аномалия удаления в состоянии принести не меньшие неприятности. Как вы думаете, как скажется на финансовом состоянии фирмы тот факт, что она станет выплачивать ежегодные премии давно уволенному менеджеру? Это печальное событие произойдет только потому, что в бухгалтерии забудут вычеркнуть всего одну строку с данными.

Несовместимость файлов. Структура файлов с данными определялась не только разработчиками программного обеспечения, но и языками программирования, состоящими на вооружении в тех или иных организациях. Построение файла, описанного на языке Algol, могло принципиально отличаться от структур, генерируемых средствами PL/1, ADA или какого-нибудь еще средства разработки приложений тех времен. Более того, дополнительные ограничения вносились из-за особенностей архитектурных решений, принятых в основу построения тех или иных ЭВМ. Помножьте это на специфичные черты различных операционных систем. Как следствие выходящие из-под «пера» программистов файлы зачастую становились несовместимыми, хотя и содержали практически идентичное описание данных.

Разрастание количества приложений. Сами по себе данные не представляют никакого интереса. Представьте, что у вас имеется файл с несортированными телефонными номерами жителей миллионного города – это хорошая новость. А теперь плохая – у вас нет средств упорядочивания и поиска данных. В результате цена таким данным – ломаный грош, ну-ка найдите номер телефона гражданина Иванова, потерявшегося где-то среди сотен тысяч других номеров... Данные надо не только хранить, но и уметь представлять пользователю в удобном формате. А пожеланий у пользователей не счесть, одним требуется,

чтобы списки заказов упорядочивались по алфавиту, другим – по дате заказа, третьим хотелось бы, чтобы имелась возможность сортировки записей по денежной сумме. Старуха, затребовавшая от Золотой рыбки перечень услуг (начиная от тривиального корыта и заканчивая царским тронem), – просто ангел в сравнении с запросами к данным у биржевого аналитика, главного бухгалтера завода или заведующего гипермаркетом. Идеи и пожелания пользователей сыплются на программиста как из рога изобилия, и никто, кроме него, не ведает, что каждый новый запрос приводит к цепной реакции – бесконечной переработке исходного приложения. В конце концов, головная программа обрастает скопищем утилит и «утилиток», что рано или поздно (а главное – необратимо) приведет проект к коллапсу.

Пути устранения недостатков систем файлов

Сегодня системы, основанные на файлах, практически не используются, исключение составляют состоящие из одного-двух файлов данных небольшие по числу записей хранилища. Дабы не повторить ошибки программистов тех времен, проектировщики БД сделали нужные выводы.

Во-первых, разработчики в принципе отказались от хранения физической структуры данных в коде приложений. Вместо этого описание данных стало выноситься в отдельное хранилище, называемое **системным каталогом** (system catalog). Таким образом, во всех современных БД, помимо собственно хранимых в них данных, еще имеются метаданные (данные о данных). Если внешняя программа обладает возможностью чтения метаданных, то она без труда сможет получить доступ к хранимой в БД информации.

Во-вторых, стали предприниматься активные попытки стандартизировать способы описания и хранения данных. Наличие единого для всех разработчиков стандарта значительно упростило доступ к данным.

В-третьих, возникла необходимость создания единого универсального языка, позволяющего производить с данными наиболее важные операции: вставки, редактирования, удаления и просмотра.

В результате на смену морально устаревшим системам файлов пришли свободные от недостатков своих предшественников базы данных.

Как видите, в составе баз данных основную роль играют структуры и описывающие эти структуры метаданные, кроме того, в БД задействуются индексы, представления, хранимые процедуры, триггеры и т. п.

А роль приложения сведена к минимуму, т. к. основная логика управления данными осуществляется силами самой БД. В свою очередь, в системах файлов приложения важны не менее самих данных, ведь, кроме них, никто не в состоянии предоставить доступ к данным.

Что такое база данных?

На протяжении всей главы мы смело оперировали таким важным понятием, как «база данных», но пока наполняли его интуитивным содержанием. Теперь настал тот час, когда мы введем определение БД. Сначала обратимся к стандарту.

В соответствии с ГОСТ 34.321–96, устанавливающим эталонную модель управления данными, под базой данных понимается «совокупность взаимосвязанных данных, организованных в соответствии со схемой базы данных таким образом, чтобы с ними мог работать пользователь» [6].

Несмотря на то что приведенное в ГОСТ определение следует считать официальным, на страницах этой книги никто не запрещает нам подойти к определению БД творчески и немного его доработать, при этом не искажая изначального смысла, заложенного в ГОСТ (тем более что со времен утверждения стандарта прошло немало времени).

Определение

База данных (database) – это организованная совокупность совместно используемых логически связанных данных и описаний этих данных, относящаяся к определенной предметной области, предназначенная для удовлетворения информационных потребностей организации.

Дополним нашу версию определения БД рядом комментариев.

1. База данных представляет собой долговременное хранилище данных, структура которого определяется на этапе проектирования БД.
2. Находящиеся в БД данные являются общекорпоративным ресурсом, в котором вместо отдельных файлов, разбросанных по отделам и службам предприятия, данные собраны вместе с минимальной избыточностью.
3. В БД хранятся не только рабочие данные, описывающие какую-то предметную область, но и метаданные, предназначенные для

описания структуры самих данных. Такую важную способность БД часто называют самодокументированием.

4. Потребителем данных может выступать не только человек, но и программно-аппаратные комплексы (станки с числовым программным управлением, автоматизированные системы управления, робототехнические системы и т. д.), именно поэтому в определении говорится об удовлетворении информационных потребностей не просто пользователей, а организации в целом.

Эволюция моделей БД

Идея построения первых БД на принципах электронных картотек существовала сравнительно недолго и была отвергнута. Ограничения систем, основанных на файлах, вошли в непримиримое противоречие с возрастающими требованиями к хранению и обработке больших массивов данных, и, как следствие, проигравший должен был уйти с арены. Однако что можно было бы предложить взамен?

В 1960-х годах разработчикам программного обеспечения пришлось взять тайм-аут и задуматься над путями устранения возникших трудностей. Хорошим стимулом тому были грандиозные проекты второй половины века. Чего только стоила амбициозная цель НАСА первыми отправить астронавта на Луну. США, получившие поучительный щелчок по носу от СССР на первом этапе космической гонки, не жалели ни денег, ни ресурсов для взятия реванша. А борьба за космос – это не только битва реактивных двигателей. Здесь в равной степени важны достижения практически во всех областях науки и техники, и среди них далеко не на последнем месте стоят компьютерные технологии. Попробуйте представить себе астронавта с логарифмической линейкой, прикидывающего в уме, сколько надо подать топлива в форсунки двигателя, дабы в очередной раз не проскочить мимо спутника нашей планеты... Поэтому американскому космическому агентству потребовались производительные малогабаритные ЭВМ и современное программное обеспечение. В эту отрасль начали вкачиваться огромные денежные суммы. В результате над проектированием первых систем баз данных был сосредоточен внушительный спектр корпораций и научно-исследовательских лабораторий. Так что, если хотите, движущей силой современных баз данных в 1960-х годах стало уязвленное самолюбие, помноженное на отменное финансирование.

Спустя высадки человека на Луну минуло полвека, неплохой срок для подведения итогов развития баз данных. Возьмем на себя смелость утверждать, что за этот период базы данных шагнули вперед так же далеко, как и космонавтика. На рис. 1.2 отражены этапы развития моделей баз данных. Это так называемые модели реализации, т. е. модели, ориентированные на получение ответа на вопрос: «Каким образом следует описывать структуры данных?» Единственное исключение составляет понятийная модель «сущность-связь», это ближайший союзник реляционной модели, но отвечающий не за реализацию, а за логику будущей БД.

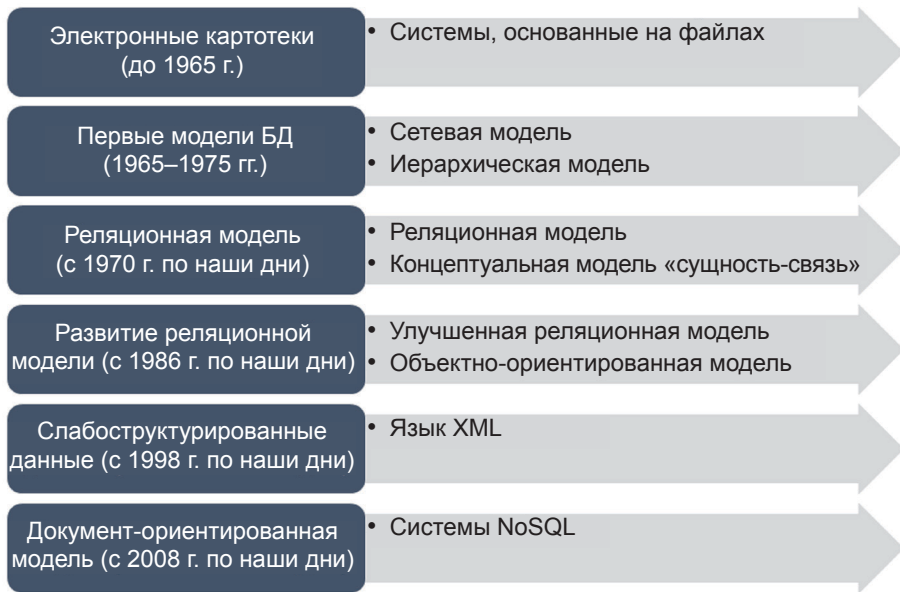


Рис. 1.2. Эволюция моделей реализации данных

Системы, основанные на файлах, лишь с большой натяжкой можно причислять к базам данных, поэтому мы отнесем их к предварительному этапу (если хотите – к этапу наивного проектирования). Появление первых моделей БД (сетевая и иерархическая модели) заменило файловые системы, этот условный «дореляционный» период продолжался с середины 60-х по первую половину 70-х годов XX века. Началом его конца стала публикация знаменитой статьи Э. Кодда о реляционных банках данных. Закат одного дня – это лишь прелюдия нового рассвета. Эпоха реляционной модели настала вместе с выходом в свет знаменитого прототипа реляционной базы данных – System-R.

Замечание

К моделям «дореляционного» периода относится еще один вид БД – системы с инвертированными списками. Указанная модель реализации БД не получила широкого распространения, поэтому в этой книге мы ее не рассматриваем.

Необходимость моделирования

Говоря об эволюции баз данных, мы уже несколько раз употребили термин «модель». Почему так акцентируется на этом внимание? Для чего нужен процесс моделирования при создании БД?

Построение различного рода моделей широко применяется в различных областях науки и техники. В авиастроении, прежде чем поднять в небо дорогостоящий самолет, его уменьшенная модель проходит исследование в аэродинамической трубе. Конструкторы летательного аппарата выясняют, насколько удачны контуры будущего истребителя или пассажирского лайнера, не рискуя жизнями экипажа. Физики и астрономы, изучая особенности протекания ядерных реакций на Солнце, не имеют даже теоретической возможности потрогать светило руками. Поэтому они моделируют термоядерную реакцию в микроскопических масштабах в своих лабораториях. Экономисты ведущих стран строят математические модели поведения финансовой системы страны, на которой апробируются их законодательные инициативы, – это гораздо безопаснее, чем опыты на своих согражданах. Несмотря на разную направленность, у всех рассмотренных выше случаев есть объединяющая черта – модель создается в том случае, когда проведение исследований на реальном объекте невозможно (по финансовым, экономическим, этическим, техническим или любым другим причинам).

Вне зависимости от моделируемого объекта создателям модели необходимо выполнить очень важное условие – модель должна быть адекватна реальности, иначе результаты исследования можно выбросить в мусорную корзину. Вполне естественно, что при подготовке модели самолета для продува в аэродинамической трубе не стоит тратить время на создание миниатюрных двигателей, прокладку электропроводки и размещение кресел внутри салона. Это не столь важно в сравнении с безукоризненным соблюдением пропорций фюзеляжа, крыльев и хвостового оперения летательного аппарата. Поэтому в процессе моделирования ученые отбрасывают малозначимые детали, уделяя первоочередное внимание ключевым характеристикам модели, в последнем примере с точки зрения аэродинамики.

На страницах этой книги мы еще много раз будем говорить о моделях баз данных и, в частности, о доминирующей сегодня реляционной модели. Теперь мы знаем, что модель базы данных призвана отражать реальные и виртуальные объекты окружающего мира, информацию о которых мы планируем хранить и обрабатывать в разрабатываемых информационных системах. Чем точнее модель отражает действительность – тем она полезнее и, как следствие, лучше база данных.

На сегодняшний момент времени подавляющее большинство современных баз данных ориентировано на реляционную модель. Это простейшие однопользовательские системы типа Microsoft Access и SQLite и сложные клиент-серверные системы SQL Server компании Microsoft, Informix, Oracle, InterBase, MySQL. Список можно продолжить... Но так было не всегда. До второй половины семидесятых годов прошлого века рынок баз данных был поделен примерно поровну между программными продуктами, исповедующими иерархическую или сетевую модель представления данных.

Иерархическая модель

Своим появлением на свет иерархическая модель данных обязана лунному проекту Apollo и двум американским компаниям: Rockwell International и IBM. Президент США Дж. Ф. Кеннеди пообещал высадить американского астронавта на Луну к концу 1960-х. Для доставки человека на поверхность спутника нашей планеты требовалось не только рассчитать траекторию полета, но и обработать огромное количество вспомогательных данных, начиная от расхода кислорода астронавтами и заканчивая учетом заказов на уборку помещений НАСА. Итак, в начале 1960-х North American Aviation (именно так в те годы именовалась Rockwell) стала генеральным подрядчиком по разработке программного обеспечения для обслуживания всего проекта. В результате пятилетнего труда специалисты компании предложили заказчику программу с мудреным названием «Обобщенный метод доступа и модификации» (Generalized Update Access Method, GUAM), способную хранить данные в виде перевернутой иерархической структуры (рис. 1.3). В это же время к Rockwell присоединился субподрядчик IBM, преследовавший еще одну заветную цель – разработать первый коммерческий проект баз данных. Совместное детище двух компаний появилось на свет во второй половине 1960-х и получило название «Информационно-управляющая система» (Information Management System, IMS). В отличие от всех своих конкурентов, проект IMS

приобрел одно уникальное по тем временам преимущество – система позволяла работать с данными не только на физическом, но и на логическом уровне. Новый программный продукт очень скоро стал столь популярен, что без него не обходился ни один крупный мейн-фрейм вплоть до конца 70-х годов прошлого века.

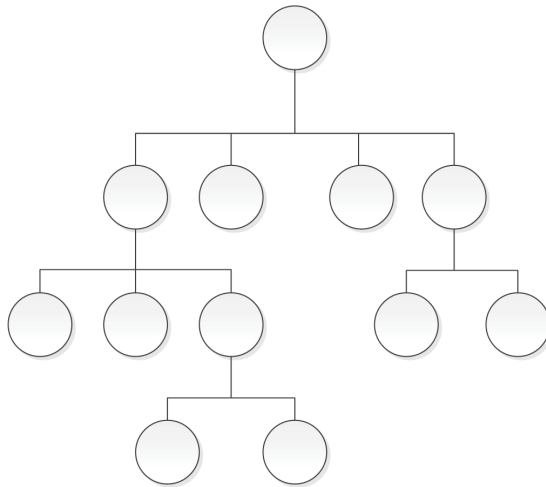


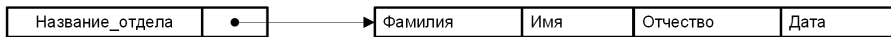
Рис. 1.3. Иерархическая модель данных

Суть иерархического хранения данных заключается в применении инвертированной древообразной структуры. В наивысшей точке располагался корень дерева, ниже – дочерние узлы (листья дерева). Все узлы связывались друг с другом благодаря сложной системе указателей. Каждый из узлов, в свою очередь, мог являться родительским по отношению к одному или нескольким нижерасположенным узлам, но у дочернего узла не может быть более одного родителя. Таким образом, в иерархической модели реализована одна из наиболее часто встречающихся в реальном мире связей между сущностями – связь «один ко многим». Например, в городе находится много компаний, в компании образовано много отделов, в отделе работает много сотрудников. Подобное решение существенно снижает остроту проблемы избыточности данных. Проект базы данных «отдел – сотрудник», реализованный средствами иерархической модели, мог бы выглядеть так, как представлено на рис. 1.4.

Это несколько усовершенствованное решение предусматривает описание дополнительных данных – места работы сотрудника. Заметьте, что узлы сотрудников выступают в качестве подчиненных по отноше-

нию к узлам отделов и не могут существовать без них. При желании схему можно и развить, например подчинив сотрудникам узлы с заказами, описанием работ, номерами счетов и т. п. Подобное развитие не требует кардинального изменения уже существующих узлов дерева, мы лишь добавляем очередные узлы на новом нижележащем уровне.

Модель



Пример



Рис. 1.4. Пример работы иерархической БД

По сравнению с системами файлов иерархическая модель весьма неплоха, вот только ключевые ее достоинства:

- 1) простота понимания структуры данных. Иерархическое построение данных интуитивно понятно, что существенно упрощает проектирование БД;
- 2) целостность данных. Иерархические БД представляли собой неразрозненные приложения и файлы. Все данные находились под контролем одной системы управления базами данных, которая не допускала некорректные действия с записями (например, удаление родительского узла, у которого оставались «осиротевшими» дочерние элементы);
- 3) независимость данных. Данные больше не принадлежат одному приложению. Наличие системного каталога позволяет работать с БД приложениям, умеющим читать метаданные;
- 4) безопасность данных. Контролируемый доступ к данным осуществляется с помощью СУБД, в которую закладывается предпочтительная политика безопасности.

Однако, поборов недостатки систем, основанных на файлах, иерархическая модель приобрела новые. Вот только некоторые из них:

- 1) ограничения в организации отношений между сущностями. Иерархическая модель позволяет организовать последовательную связь «один ко многим» между данными, но не в состоянии реализовать отношения «многие ко многим»;
- 2) структурная зависимость. Иерархическая структура предполагала, что физически данные также станут храниться в виде дерева. Серьезное изменение структуры (например, переподчинение узлов) могло привести к тому, что прикладные приложения теряли возможность навигации по данным;
- 3) сложность разработки прикладного программного обеспечения (ППО). Разработчик ППО должен знать особенности физического хранения данных, иначе он мог просто заблудиться в запутанной системе указателей.

Ко всему прочему иерархическая модель не была стандартизирована. Как следствие всегда существовала проблема переносимости данных между приложениями различных разработчиков.

Замечание

Иерархическая организация данных очень удобна при осуществлении поиска данных, запрашиваемые сведения уже сгруппированы по соответствующим ветвям дерева, и нам остается лишь спускаться от корня схемы к требуемому листу.

Сетевая модель

Практически параллельно с North American Aviation и IBM свои собственные разработки вела еще одна серьезная американская корпорация – General Electric. Ею в 1964 году был выпущен релиз первой сетевой базы данных IDS (Integrated Data Store). Так же, как и разработчики иерархической модели, General Electric работала в условиях отсутствия утвержденного стандарта на описание среды БД.

Заметим, что когда мы говорим о «сетевой» модели, в виду имеется особенность организации данных, а не способ объединения компьютеров в сеть. В данном случае за основу была взята не древовидная модель данных, а структура, построенная на основе графа (рис. 1.5).

По сравнению с иерархической, сетевая модель имеет одно серьезное преимущество: она позволяла назначать произвольное количество связей между узлами графа. Поэтому она в состоянии создавать базы данных, более точно отражающие связи реального мира, в частности в сетевых БД без особого труда можно было формировать отношения

«многие ко многим» или замыкать связь узла на себя самого. При удалении записи в сетевой БД уничтожается только этот соответствующий ей элемент и связь с ним, все остальное остается на месте. Это разительное отличие от иерархически организованной БД, ведь здесь при удалении одного узла удалению подлежала вся нижерасположенная ветвь дочерних элементов или требовалось провести относительно сложную процедуру переподчинения дочерних узлов. Впрочем, простота перестроения структуры графа имеет и свои подводные камни, ведь любое непродуманное изменение связей может привести к нарушению целостности данных.

Замечание

Ключевое отличие между иерархической и сетевой моделями данных заключается в том, что в древообразных структурах дочерний узел может обладать только одним родительским узлом, а в сетевой модели ограничение на количество родительских элементов отсутствует.

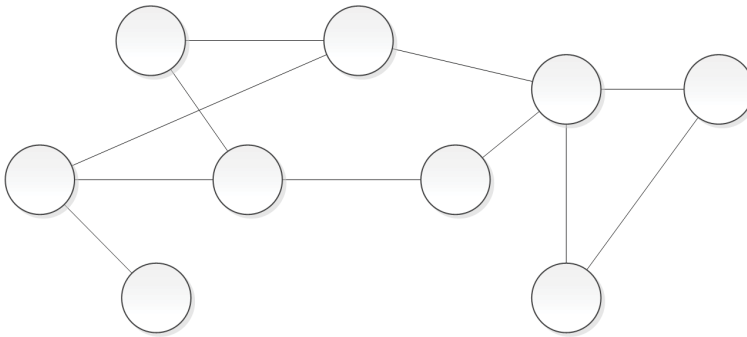


Рис. 1.5. Сетевая модель данных

К сожалению, сетевая модель также не свободна от недостатков:

- большое количество произвольных связей повышает сложность схемы БД и как следствие вызывает дополнительные трудности при обеспечении целостности данных;
- сложность разработки прикладного программного обеспечения.

Замечание

Говоря о сетевой модели данных, обязательно следует упомянуть заслуги руководителя проекта IDS Чарльза Вильяма Бэчмана (Charles William Bachman), за свой вклад в науку в 1973 году он был удостоен премии Тьюринга.

Говоря о сравнительных характеристиках рассмотренных иерархической и сетевой моделей, сразу отметим, что они свободны от недостатков систем, основанных на файлах. Все это достигнуто благодаря тому, что иерархическая и сетевая модели предусматривают существование некоторого внешнего хранилища (системного каталога), в котором размещаются описания данных. Современные системы управления базами данных размещают в нем всю информацию, необходимую для нормальной жизнедеятельности базы данных, включая описание используемых типов данных, ограничений на вводимые данные, отношений, связей между отношениями и многое-многое другое. Благодаря наличию системных метаданных достигается архиважный для базы данных фактор физической независимости приложений от данных. Теперь грамотно спроектированное приложение после ознакомления с системным каталогом способно получать доступ практически к любому набору данных.

Попытки разработки стандарта БД

Во многом недостатки первых БД были предопределены вакуумом в области стандартов. Разработчики программного обеспечения фактически работали в голом поле, полностью полагаясь на свою интуицию. Поэтому уделим несколько минут внимания знакомству со сложившейся к середине 1960-х годов обстановкой в области стандартизации БД.

А ситуация напоминала извечный философский спор о первичности курицы или яйца. Корректный стандарт, определяющий способы хранения и описания данных в будущих БД, не мог появиться на пустом месте, для этого был необходим опыт реализации и эксплуатации подобных баз. Одновременно с этим любая попытка построения СУБД в условиях отсутствия стандартов была обречена на провал. Существовали ли выход из такого положения? Оказывается, да.

Острое желание победить в лунной гонке, помноженное на завидные финансовые возможности, позволило США пойти по пути одновременного создания курицы и яйца. Компании North American Aviation, IBM, General Electric и десятки их помощников уже приступили к проектированию пилотных версий БД. Одновременно с этим в 1965 году при непосредственном покровительстве правительства США на конференции CODASYL (Conference on Data Systems Languages) была сформирована рабочая группа List Processing Task Force, переименованная в 1967 году в группу Data Base Task Group (DBTG). Всю свою энергию DBTG посвятила вопросам определения спецификаций среды, которая допускала бы

разработку баз данных и управление данными. Группа сумела написать два научных отчета, промежуточный в 1969 и итоговый в 1971 году. Заметьте, что к этому времени астронавты уже оставили свои следы на луне.

Несмотря на упорные старания группы DBTG, ее труд не смог стать стандартом, отчеты DBTG не удовлетворили Национальный институт стандартизации США (American National Standard Institute, ANSI). Вместе с тем работа ни в коем случае не прошла даром и подготовила фундамент для будущих исследований. Сложно в двух предложениях резюмировать более чем пятилетнюю работу специалистов, но отметим следующее. Группа DBTG признала необходимость использования двухуровневого подхода, построенного на основе использования системного представления и пользовательских представлений. Если уж совсем просто – БД должны создаваться так, чтобы пользователю даже не имело смысла интересоваться, каким образом данные хранятся физически, ему надо лишь обладать возможностью ими пользоваться.

В 1975 году Комитет планирования стандартов и норм SPARC (Standards Planning and Requirements Committee) Национального института стандартизации США предложил свое видение модели БД – 3-уровневую архитектуру ANSI/X3/SPARC. Ближе всего к данным предлагалось ввести первый уровень абстракции – внутренний (рис. 1.6). На внутренний уровень возлагались задачи управления физическим хранением данных в памяти ЭВМ. Над внутренним уровнем расположился очередной уровень абстракции – концептуальный. Этот уровень уже не нес ответственности за хранение данных на носителях информации. Комитет SPARC предлагал возложить на концептуальный уровень обязанности за описание логической структуры всех данных, в частности логический смысл хранимых в БД данных, связи между данными, ограничения на данные, правила поддержки целостности и т. п. Наконец, третий по счету слой архитектуры ANSI/X3/SPARC получил название внешнего уровня. Внешний уровень отвечал за организацию интерфейса между человеком и БД. Конечный пользователь нашего программного продукта может быть высококлассным программистом, а может и оказаться «чайником». Уровень должен уметь предоставить и тому, и другому соответствующие его потребностям и умениям инструменты по доступу и управлению данными.

Таким образом, архитектура ANSI/X3/SPARC развивает задумку группы DBTG. И там, и здесь во главу угла ставится обеспечение независимости от данных. Независимость означает, что изменения на нижних уровнях никак не влияют на верхние уровни. Только в последнем случае

комитет SPARC пошел дальше и предложил три слоя абстракции. Благодаря этому модель обеспечивала не только физическую, но и логическую независимость от данных.

Впрочем, дополнительный уровень не решил проблему, и модель ANSI/X3/SPARC все равно не сумела стать стандартом. Несмотря на то что труды группы DBTG и комитета SPARC не привели разработчиков к запланированному результату и не стали стандартом, предложенные ими модели оказали значительное влияние на специалистов-практиков, в особенности на создателей первой реляционной БД, получившей название System-R.

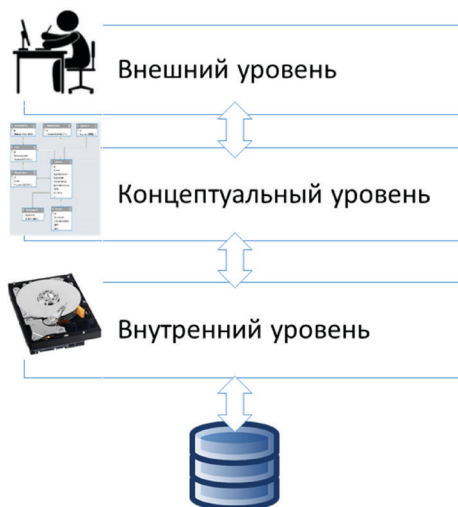


Рис. 1.6. Трехуровневая архитектура ANSI/X3/SPARC

Реляционная модель

Довольно скоро на смену иерархической и сетевой моделям пришла принципиально новая модель данных – реляционная. Реляционная модель в следующих главах будет рассмотрена весьма подробно, а пока упомянем лишь то, что она основана на принципе выявления подлежащих описанию в БД сущностей и связей между ними. По сравнению со своими товарками она отличается высокой гибкостью, здесь программист гораздо в меньшей степени ломает голову над механизмом управления данными. Эта задача автономно решается ядром СУБД. Реляционные БД снабжены специализированным языком SQL, который достаточно легок в освоении. Таблицы реляционной модели жестко структурированы, что упрощает их обслуживание.

Теория реляционной модели в первую очередь опиралась на вышедшую в 1970 году статью¹ Э. Кодда (E. F. Codd), а первый существенный прикладной результат пришел в 1976-м. В этом году в исследовательской лаборатории корпорации IBM, расположенной в городе Сан-Хосе штата Калифорния, на свет появился прототип современных реляционных СУБД – проект System-R. Руководителем проекта был Мортон Астрахан (Morton M. Astrahan).

Проект System-R преследовал цель доказать практичность реляционной модели, что достигалось посредством реализации предусмотренных ею структур данных и требуемых функциональных возможностей. На основе этого проекта был разработан структурированный язык запросов (в ту пору названный SEQUEL), который несколько позднее (в 1986 г.) стал стандартом SQL.

На базе System-R впоследствии (в 1975–1979 гг.) был создан первый успешный коммерческий реляционный продукт фирмы IBM – DB2. Говоря о DB2, нельзя не упомянуть одного из ее авторов – Криса Дж. Дейта (Chris J. Date). На сегодняшний день это ведущий специалист по реляционной модели данных, по всему миру широко известна и многократно переиздавалась его книга «Введение в системы баз данных» [19], на которой выросло не одно поколение программистов баз данных.

Практически параллельно с System-R над схожим проектом в Калифорнийском университете работали Майкл Стоунбрейкер (Michael Stonebraker) и Юджин Вонг (Eugene Wong). Проект получил название INGRES (INteractive GReaphics REtrieval System) и был доведен до реально работающей одноименной СУБД. Позднее на фундаменте INGRES была разработана СУБД Postgres (от лат. *post* и *Ingres*), которая, в свою очередь, стала родоначальником очень популярной сегодня СУБД PostgreSQL.

Как и все в нашем мире, реляционная модель данных далеко не идеальна. По существу, это компромиссное решение между потребностью отражать сущности реального мира и связи между ними и ограниченными возможностями математической теории множеств, переложенной на программный код. Любой компромисс предполагает появление множества малопривлекательных ограничений. Так, из-за борьбы с избыточностью данных в реляционной БД (процесс нормализации) сведения «размазываются» по нескольким отношениям (таблицам). Как следствие для получения сводного отношения приходится собирать данные по крохам и осуществлять множество соединений. Чем больше соединений следует провести, тем больше временных и ресурсных затрат.

¹ Codd E. F. A Relational Model of Data for Large Shared Databanks. Communications of the ACM, 06.1970. P. 377–387.

Другая проблема реляционной модели заключается в особенностях организации связи между отношениями. Для моделирования всего многообразия взаимодействия между сущностями реального мира в нашем распоряжении имеется лишь одна конструкция «один ко многим». Для создания отношения «многие ко многим» приходится выкручиваться за счет введения дополнительной ассоциативной таблицы и применения двух типов связей «один ко многим». Но на этом проблема построения связей не исчерпывается. Реляционная модель попросту бессильна, когда следует отразить смысловую нагрузку связи. Что делать, когда семантика связи между сущностями различается? Мы с вами видим разницу между глаголами «обладает», «подчиняется» или «управляет», а реляционная модель нет...

Замечание

В отличие от однотипных записей реляционных БД, записи в иерархической и сетевой моделях могут обладать различной структурой, т. е. содержать различное число разнотипных полей.

Спустя пять лет после того, как мир впервые услышал о реляционной модели, ученый Питер Чен Пин-Шен (Peter Pin-Shan Chen) дополнил теорию моделирования данных доктора Кодда весьма удобным описанием модели, получившей название ER-модель. Об этой модели мы поговорим в 6-й главе книги, а пока отметим то, что Питер Чен предложил простой и эффективный способ представления сущностей, атрибутов и связей между ними в виде наглядных диаграмм. В отличие от сетевой, иерархической и реляционной моделей (относящихся к классу моделей реализации), модель Чена принадлежит к разряду концептуальных (понятийных). Другими словами, ER-модель отражает логическую природу представления данных.

Объектно-ориентированная модель

В середине 1980-х годов на рынке программных продуктов стали активно появляться программные средства, построенные на основе объектно-ориентированной парадигмы. В ту пору все разработчики как заклинание повторяли новые термины: абстрагирование, инкапсуляция, модульность, иерархичность. Теперь на верхушку пирамиды вознесен Его Величество Объект. Объект в технологии объектно-ориентированного программирования выступает удобной абстракцией объектов из реального мира. Он позволяет описывать все, начиная от

авторучки и заканчивая тепловозом. Главное, чтобы программист грамотно сконструировал исходный класс. Программный объект обладает свойствами, методами и способен реагировать на события. Одним словом, достоинств столько, что многие начали говорить о том, что настал час задуматься о замене реляционной модели данных на перспективную объектно-ориентированную.

Одним из первых теоретических изысканий считается семантическая модель данных (Semantic database model, SDM), предложенная М. Хаммером (M. Hammer) и Д. Маклеодом (D. McLeod) и опубликованная в статье в 1981 году.

Желание создать что-то новое было столь острым, что в первой половине 90-х годов группой под названием Object Database Management Group были предложены рекомендации по созданию объектно-ориентированных баз данных ODMG-93. Примерно тогда начали появляться мнения о закате реляционной модели данных. Однако до сегодняшнего дня реализовать рекомендации ODMG-93 в полном объеме не удалось. Основная причина этого – отсутствие развитого математического аппарата, способного описывать данные в формате, приемлемом для объектно-ориентированной теории.

С точки зрения точности моделирования реального мира объектно-ориентированная модель данных по всем статьям сможет превзойти реляционную, ведь в идеале она должна быть способной адекватно выразить информационные структуры любой сложности. То же самое можно сказать и о реализации связей между объектами – вариантов не счесть. Самым главным достоинством СУБД очередного поколения может стать тот факт, что проведение логических операций над данными многократно усилится за счет возможностей ООП. Но, к сожалению, даже спустя три десятилетия мы не увидели коммерчески успешного программного продукта с этикеткой «объектно-ориентированная СУБД». Так что пока можно говорить только о взаимной интеграции реляционной модели и технологии ООП.

Замечание

Наибольшие противоречия между специалистами по базам данных вызывает объектно-ориентированная модель баз данных (ООМБД). Ряд авторов утверждает, что ООМБД (object-oriented database model, OODM) приходит на смену реляционной и представляет собой СУБД 3-го поколения, а другие, например Крис Дж. Дейт, вместо термина «объектно-ориентированная модель» используют термин «объектно-реляционная система» [19], тем самым подчеркивая, что объектно-ориентированные

системы лишь дополнили реляционную модель. Так что слухи о кончине реляционной модели оказались сильно преувеличены. И на сегодняшний день реляционная модель является доминирующим направлением развития всех современных систем баз данных. Практическим подтверждением слов К. Дейта является реляционная СУБД Oracle 11.0, поддерживающая элементы объектно-ориентированного подхода.

Существенный вклад в разработку объектно-реляционной системы внес американец Майкл Стоунбрэкер (Michael Stonebraker). Майкл в 1970-е работал над проектом Ingres, в 80-е возглавлял проект Postgre, в 90-е ряд его идей был реализован в СУБД Informix Universal Server. Стоит отметить, что Стоунбрэкер (так же, как и Крис Дж. Дейт) с крайней осторожностью применял термин «объектно-ориентированная модель», предпочитая вместо него говорить о системах баз данных будущего поколения.

Слабоструктурированные данные

В тех ситуациях, когда на первое место выступает не столько решение задачи хранения записей, сколько простота организации и универсальность обмена данными, стоит обратить внимание на идею работы со слабоструктурированными данными. В качестве естественного способа хранения подобных данных выступает обычный текстовый формат. Что можно придумать проще?

Для обслуживания слабоструктурированных данных в 1998 году разработан отдельный стандарт – расширяемый язык разметки (eXtensible Markup Language, XML). Особо подчеркнем, что слабоструктурированные данные в формате XML не нуждаются в услугах отдельных СУБД, а для обработки XML разработан специальный прикладной интерфейс программирования.

Документ-ориентированная модель

Документ-ориентированная (document-oriented) модель данных – явление сравнительно новое, появившееся в начале 2000-х. Данная модель выступает едва ли не антиподом реляционной теории и опирается на идею хранения иерархических структур данных, в которой каждый элемент представляет собой целостный, а не «размазанный» по двумерным отношениям, как в реляционной модели, документ. Документы хранятся не в таблицах, а в специальных хранилищах документов (document store). Основным инструментом по доступу и управлению данными выступает декларативный язык NoSQL (Not only SQL).

Документ-ориентированные СУБД свободны от ключевого недостатка их реляционных коллег – существенных затрат на сборку данных из нескольких таблиц в единое целое. В этом просто нет необходимости, ведь документ никто и не собирался разделять. Обратной стороной медали выступает ограниченность синтаксических конструкций по управлению данными языка NoSQL.

Классическими примерами документ-ориентированных СУБД можно считать MongoDB, CouchDB, Couchbase, MarkLogic.

Замечание

Рассмотренные в главе модели БД являются наиболее востребованными, но далеко не единственными. Помимо них, можно упомянуть хранилища «ключ–значение», графовую и столбцовую модели.

Резюме

За сравнительно недолгий период существования модели реализации баз данных совершили несколько эволюционных шагов – от первых систем, основанных на файлах современных моделей. Отказ от систем файлов и переход к базам данных позволили получить ряд существенных преимуществ: непротиворечивость данных, целостность данных, совместное использование данных, контролируемая избыточность и высокая безопасность данных.

Совместно с моделями реализации развивалась и теория баз данных, появлялись на свет принципиально новые языки программирования (например, ставший сегодня стандартом структурированный язык запросов SQL, языки NoSQL и NewSQL) и особый вид программного обеспечения – системы управления базами данных (СУБД).

Вопросы для самопроверки

1. Перечислите ограничения, присущие системам, основанным на файлах.
2. Что стало причиной большинства ограничений систем, основанных на файлах?
3. Для чего предназначен системный каталог?
4. Что такое метаданные?
5. Что понимается под термином база данных?
6. Поясните, почему база данных является моделью.
7. Прокомментируйте этапы развития моделей данных.

8. Почему в вопросе хранения данных так важна стандартизация?
9. Каковы состав и назначение уровней архитектуры ANSI/SPARC?
10. Перечислите достоинства и недостатки иерархической модели данных.
11. Перечислите достоинства и недостатки сетевой модели данных.
12. Для чего нужны слабоструктурированные данные?
13. Что поставлено в основу объектно-ориентированной модели данных?
14. В чем заключается ключевое отличие в подходе хранения данных между реляционной и документ-ориентированной моделями?

Глава 2

Система управления базами данных

В «старозаветные» времена, когда доминирующим способом хранения и обслуживания данных выступали системы, основанные на файлах (электронные картотеки), разработчики этих систем были вынуждены создавать программный пакет полностью от «А» до «Я». Программистам приходилось определять структуры, выдумывать методы хранения и доступа к записям. С появлением более совершенных моделей данных для управления данными стали разрабатывать специальное программное обеспечение, получившее название системы управления базами данных (СУБД).

Вне зависимости от того, на основе какого подхода спроектирована современная БД, ее существование немыслимо без системы управления базами данных. Прямо или косвенно СУБД используется администраторами, разработчиками БД, программистами и обычными пользователями. Для этого СУБД предоставляет определенный набор инструментов, упрощающих проектирование, администрирование БД и обеспечивающих доступ к данным.

Так что такое СУБД? Предлагаю нашу работу начать с разъяснения этого термина. В [6] предлагается под СУБД понимать «совокупность программных и языковых средств, обеспечивающих управление базами данных». Мы вновь позволим себе небольшую вольность и немного расширим понятие.

Определение

Система управления базами данных (Database Management System, DBMS) – это комплекс программных средств, с помощью которого можно создавать и поддерживать базу данных, а также осуществлять к ней контролируемый доступ пользователей.

Введение в определение СУБД понятия «контролируемый доступ» очень важно, т. к. акцентирует внимание на то, что на СУБД возлагаются обязанности не просто по управлению базами данных, но и по:

- предотвращению несанкционированного доступа к данным;
- контролю за многопользовательским (параллельным) доступом;
- поддержке целостности данных;
- восстановлению данных.

Существует множество показателей, по которым можно классифицировать СУБД, но основной отличительный признак – модель реализации данных. Нам уже известно, что существуют сетевая, иерархическая, реляционная, объектно-ориентированная и другие модели. Про особенности перечисленных моделей мы уже поговорили в предыдущей главе, поэтому сразу перейдем к очередному параметру. Различают персональные и многопользовательские СУБД. Персональные системы предназначены для создания небольших БД, устанавливаемых на одном компьютере, поэтому их часто называют настольными. В противовес персональным, многопользовательские системы предназначены для обслуживания БД, находящихся в совместном владении несколькими пользователями. Есть и другие классификационные признаки, например по способам разработки приложений БД (ручное кодирование и автоматическая генерация форм), по возможностям определения данных, особенностям обработки транзакций, используемой ОС, по экономическим параметрам.

Функционал СУБД

В самом общем случае работу СУБД можно описать следующим образом.

1. Пользователь запрашивает у системы разрешение на доступ к данным.
2. СУБД анализирует запрос и проверяет права пользователя на осуществление запрашиваемой операции.
3. СУБД выполняет требуемые действия с данными и при необходимости возвращает результат пользователю.

На первый взгляд все просто, но давайте заглянем в функциональные обязанности СУБД глубже.

Ключевой функционал СУБД был сформулирован Коддом еще в начале 1980-х годов¹. На первом этапе насчитывалось 8 функций, позднее Кодд и другие исследователи неоднократно расширяли и уточняли этот перечень. Так что теперь можно говорить о нескольких десятках функций, служб и сервисов, которыми обязана обладать современная СУБД. В этой книге мы не станем пытаться объять необъятное и прокомментировать только основополагающие функции СУБД.

1. **Доступность данных.** Предоставление пользователю возможности вставлять, редактировать, удалять и извлекать данные из БД. При осуществлении любой из операций пользователь не должен вникать в особенности физической реализации системы, т. е. все операции должны быть прозрачны.
2. **Метаописание данных.** СУБД должна предоставлять системный каталог, в котором содержится: описание хранимых в БД данных; описание связей между данными; ограничения целостности данных; регистрационные данные пользователей и другая служебная информация. Благодаря метаданным БД становится доступной внешним приложениям, упрощается понимание смысла данных, усиливаются меры безопасности, может выполняться аудит информации.
3. **Управление параллельностью.** Реализация механизма одновременного многопользовательского (параллельного) доступа к обрабатываемым данным с гарантией корректного обновления этих данных. Умение предоставить нескольким пользователям совместный доступ к разделяемым ресурсам – это едва ли не самая сложная задача, решаемая СУБД. СУБД должна суметь избежать конфликта совместного доступа двух или большего числа пользователей к одним и тем же строкам таблицы, или, по крайней мере, исключить какие-либо нежелательные последствия при возникновении конфликта.
4. **Обработка данных в рамках транзакции.** СУБД гарантирует, что БД будет всегда находиться в непротиворечивом состоянии вне зависимости от любых сбоев при проведении операций обновления данных. Для этого операции с данными (в первую очередь вставки, редактирования и удаления) объединяются в единый блок, называемый транзакцией. Все операторы транзакции должны быть выполнены корректно и полностью, только в этом

¹ Codd E. F. (1982) The 1981 ACM Turing Award Lecture: Relational database: A practical foundation for productivity. Comm. ACM, 25(2), 109–117.

случае в БД будут зафиксированы изменения. В противном случае осуществляется автоматический откат транзакции, т. е. состояние БД будет восстановлено на момент времени, предшествующий вызову транзакции. Далее в книге мы рассмотрим много примеров транзакций, а пока ограничимся одним, наиболее близким каждому из нас. Представьте себе, что вы снимаете какую-то сумму наличных денег в одном из банкоматов вашего города. Вы уже ввели свой код доступа, указали требуемую сумму, эта сумма денег списана с вашего электронного счета и «бежит» по проводам из банка к вам в руки. И вдруг из-за сбоя питания, или отказа коммуникационного оборудования, или по какой-то другой технической причине команда на выдачу денег в банкомат не дошла. Что в результате? Вы потеряли свои деньги? К счастью, нет. Программное обеспечение банковских платежных систем увидит, что одна из операций транзакции завершена некорректно. Поэтому все изменения, сделанные указанной транзакцией, подлежат отмене – списанные электронные деньги вновь вернутся к вам на банковский счет. Вам остается повторить операцию получения наличных в другом банкомате.

5. **Обеспечение целостности данных.** Все содержащиеся в БД данные должны быть корректны и непротиворечивы. Это означает, что данные в таблицах могут модифицироваться только в соответствии с утвержденными правилами. В самом общем случае можно говорить о существовании трех правил поддержания целостности данных: целостность доменов, целостность отношений, целостность связей между отношениями. Кроме того, разработчик имеет возможность описывать свои собственные бизнес-правила, которые мы станем называть корпоративными ограничениями.
6. **Восстановление данных.** В случае непредвиденных ошибок и сбоев, приведших к повреждению или разрушению данных, СУБД должна обладать возможностью восстанавливать пострадавшие данные. В первую очередь эта функция реализуется с помощью процедур резервного копирования.
7. **Обмен данными.** СУБД обязана поддерживать современные сетевые технологии и предоставлять доступ к БД удаленным персональным компьютерам.
8. **Контроль за доступом к данным.** Доступ к данным могут осуществлять только зарегистрированные в СУБД пользователи в соответствии с назначенными администратором СУБД им правами.

Еще раз отметим, что список обязанностей СУБД на этом далеко не заканчивается. Современные системы предоставляют нам средства мониторинга, сервисы статистического анализа, утилиты экспорта/импорта данных, развитые средства проектирования БД и прикладного программного обеспечения, инструменты реорганизации файлов данных и индексных данных, службы аудита и многое другое. Однако все дополнительные сервисы и службы выполняют вспомогательную роль, а основы жизнедеятельности СУБД определяются перечисленными выше функциями.

Компоненты СУБД

СУБД – это сложный вид программного обеспечения, над созданием которого работают большие коллективы высококвалифицированных программистов. На современном рынке программного обеспечения конкурирует около трех десятков коммерческих СУБД. Из малых систем, рассчитанных на одного пользователя, сегодня наибольшей популярностью пользуются Microsoft Access и SQLite. В перечень получивших широкое признание многопользовательских реляционных СУБД входят: Oracle, SQL Server компании Microsoft, InterBase, Firebird, MySQL, PostgreSQL, Informix и т. д. Несмотря на то что все перечисленные программные продукты предназначены для решения практически одних и тех же задач, входящие в перечисленные СУБД программные компоненты и взаимосвязи между ними далеко не идентичны. Поэтому любая попытка обобщить все известные технические решения построения СУБД и на основе этого обобщения построить структурную схему типичной СУБД, пусть даже высокой степени абстракции, несколько наивна. Тем не менее мы попытаемся это сделать.

Для того чтобы СУБД оказалась в состоянии предоставлять минимальный набор из рассмотренных выше восьми сервисов, она должна состоять из набора компонент, представленных на рис. 2.1.

Над верхним уровнем системы расположены потребители услуг СУБД:

- администратор БД;
- программисты;
- конечные пользователи.

Администратор БД отвечает за планирование и физическую реализацию проекта. Он создает основные объекты БД, определяет правила поддержания целостности и непротиворечивости данных, управляет политикой безопасности, анализирует процесс эксплуатации проекта,

контролирует производительность системы – одним словом, поддерживает жизнедеятельность БД. В распоряжении администратора имеются разнообразные средства проектирования БД, эти средства могут входить в состав СУБД в виде дополнительных программных модулей, а могут быть поставлены и сторонними разработчиками.

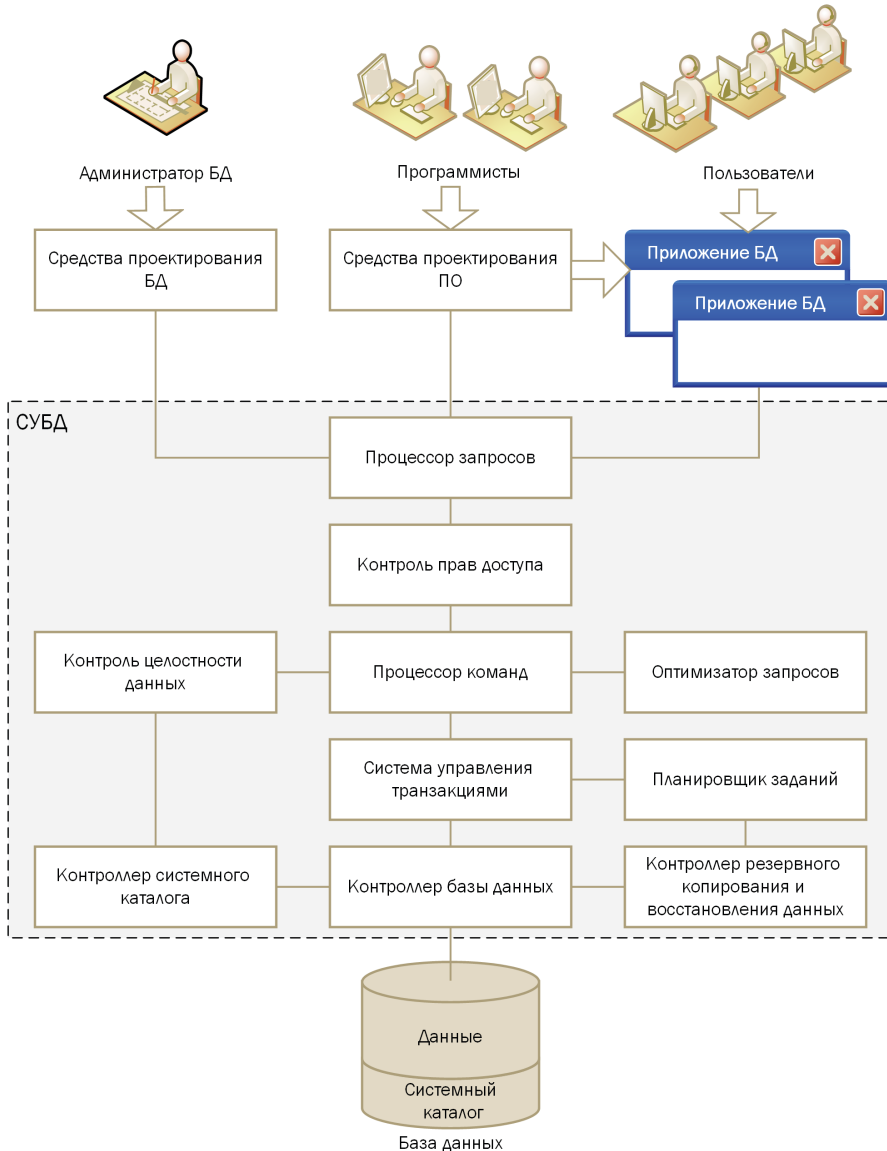


Рис. 2.1. Обобщенная структура СУБД

Программисты совместно с администратором трудятся над физическим созданием проекта БД. Но прикладных программистов в большей степени интересует не сама концепция проекта БД, а способы донесения этой концепции до конечного пользователя. Поэтому область интересов программистов смещена в сторону разработки клиентских приложений и отчетов. Основным инструментом прикладного программиста выступают многочисленные среды проектирования, как правило, не ниже 4-го поколения.

Основным потребителем услуг СУБД выступает обычный пользователь – в его интересах создаются БД и прикладное программное обеспечение. Среди пользователей есть и неплохие специалисты, чьи советы способны помочь разработчикам БД, а на другом конце спектра находятся пользователи, с трудом обнаруживающие нужные кнопки на клавиатуре. Последние могут даже и не понимать, что они работают с БД, расположенной на другом компьютере.

Основным средством общения между людьми и базой данных выступает структурированный язык запросов SQL. Поэтому средства проектирования БД, ПО и клиентские приложения БД отправляют в адрес СУБД инструкции на языке SQL. Эти команды поступают на процессор запросов, который преобразует их в набор низкоуровневых команд, понятных ядру СУБД.

К базе данных могут обращаться различные категории пользователей, от руководителя предприятия до случайного посетителя. Одним из них разрешается просматривать и редактировать любые данные, другим могут быть доступны лишь выборочные сведения, третьим вообще не стоит даже знать о существовании БД. Именно поэтому в составе большинства СУБД имеется модуль, отвечающий за контроль прав доступа. В простейшем случае модуль следит за тем, чтобы к БД присоединялись только авторизованные пользователи, для этого полученные во время регистрации пароль и логин потенциального клиента сверяются с хранящимися в системном каталоге учетными записями. В современных многопользовательских СУБД система безопасности значительно сложнее и включает в себя комплекс программных и аппаратных средств защиты.

Убедившись, что инструкция поступила от доверенного лица, модуль контроля прав доступа передает ее в распоряжение процессора команд. В первую очередь процессор убеждается, что поступившая команда не противоречит ограничениям целостности данных. Это зона ответственности модуля контроля целостности данных. На время проверки команды на соответствие ограничениям целостности доменов,

сущностей и связей задействуется контроллер системного каталога. Именно он имеет возможность собирать метаданные, в которых прячется техническое описание нашей БД. Поняв, что угрозы целостности нет, процессор передает команду оптимизатору запросов. Задача оптимизатора – найти наиболее эффективный способ выполнения поступивших команд. Наконец, оптимизированная команда компилируется и передается во власть системе управления транзакциями. Система управления транзакциями, во-первых, отвечает за полное и корректное выполнение блока команд и, во-вторых, совместно с планировщиком заданий обеспечивает параллельную многопользовательскую обработку данных. Наконец, блок команд передается в распоряжение контроллеру баз данных. Задача модуля заключается в организации взаимодействия СУБД с файлами БД и файлами системного каталога. При этом для осуществления стандартных операций ввода-вывода задействуются возможности операционной системы.

Системный каталог

В отличие от систем, основанных на файлах, хранивших описание своих данных внутри исполняемых файлов, все современные СУБД обладают системным каталогом, в котором хранятся следующие сведения:

- 1) описание поддерживаемых типов данных;
- 2) описание развернутых БД (схемы данных) и входящих в них объектов (домены, таблицы, представления и т. д.);
- 3) сведения об ограничениях целостности;
- 4) имена и права пользователей, имеющих доступ к данным;
- 5) разнообразная статистика.

Благодаря системному каталогу легко определить смысл данных, что (в отличие от систем, основанных на файлах) позволит пользователям понять назначение и состав БД.

Замечание

Обычно системный каталог физически реализуется в виде отдельной базы данных с системными таблицами по умолчанию, скрытыми от обычных пользователей.

Архитектурные решения доступа к БД

Поиски эффективных архитектурных решений организации доступа к данным начались с первых дней работы с БД. Все решения так или ина-

че отталкивались от текущего состояния электронной вычислительной техники. Во времена доминирования больших вычислительных машин доступ к самым первым БД базировался на принципе телеобработки. База данных и СУБД размещалась в памяти центральной ЭВМ, к этой ЭВМ подключались терминалы. Так как терминальные устройства особым умом и сообразительностью не отличались (они не обладали собственными вычислительными возможностями), то вся обработка данных осуществлялась только в процессоре центральной ЭВМ. Системы на основе телеобработки доминировали до начала 1970-х годов, но с изобретением микропроцессоров стали постепенно уступать свои позиции и к сегодняшнему дню практически не используются.

Файл-сервер

Бурное развитие микропроцессорной техники в 70-х годах прошлого века привело к появлению сравнительно дешевых микро-ЭВМ. Теперь руководители предприятий вместо покупки одной большой дорогостоящей ЭВМ начали приобретать несколько менее производительных, но вполне приемлемых для решения задач, стоящих перед их коллективами. Микро-ЭВМ объединялись в простейшие одноранговые локальные сети, в которых каждая из машин обладала равными правами со своими соседями. Одна из ЭВМ (с накопителями на жестких магнитных дисках наибольшего размера) назначалась файловым сервером. На выданной в совместное пользование сетевой папке сервера, кроме обычных документов, размещали и файлы баз данных (рис. 2.2).

Для того чтобы этой БД могла воспользоваться какая-нибудь из рабочих станций, она обращалась к файловому серверу, перекачивала все файлы БД в свою память, вносила правки и возвращала файлы на прежнее место. Такой способ многопользовательского доступа к БД обладал всего одним преимуществом – простотой реализации. Все остальное было сплошными недостатками:

- на каждой рабочей станции необходимо устанавливать полную копию СУБД;
- во время работы с данными сетевой трафик возрастает до пиковых значений;
- управление параллельностью обработки данных и поддержку целостности реализовать было практически невозможно, в то время как один пользователь сохранял свою порцию данных, другой уже вносил в нее изменения.

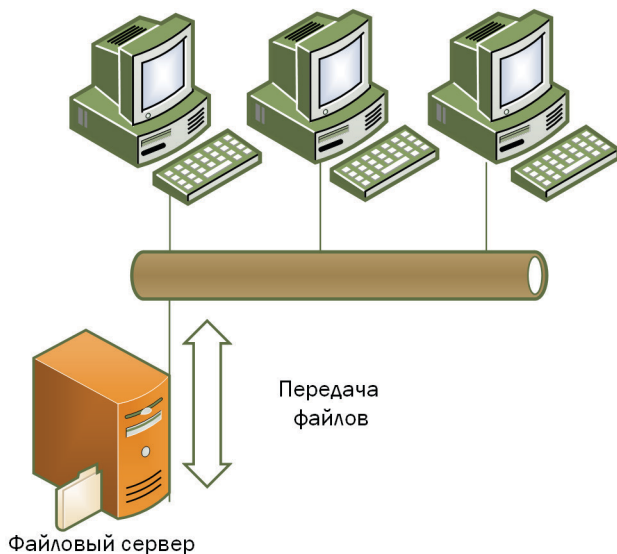


Рис. 2.2. Архитектура «файл-сервер»

Замечание

Существуют весьма продвинутые файл-серверные решения. В них сервер предоставляет клиентам доступ таким образом, что они могут осуществлять операции чтения и записи с дисками и памятью сервера так, как будто это их собственные диски и ОЗУ. Примерами таких сервисов могут стать сокеты (sockets) и именованные каналы (named piped).

Сегодня работа с БД в режиме файл-сервер популярностью не пользуется, хотя до сих пор существуют программные продукты, ориентированные на индивидуального пользователя и способные предоставить подобный сервис, например Microsoft Access.

Клиент-сервер

Факт появления клиент-серверных систем тесным образом связан с парадигмой открытых систем, активно эволюционирующих с начала 1980-х годов. Программисты пришли к выводу о необходимости распределения задач между элементами системы, в данном случае между двумя типами процессов, которые могут выполняться на различных объединенных в вычислительную сеть компьютерах. Сервер отвечает за предоставление каких-либо услуг клиентскому процессу. Клиент запрашивает у сервера определенные услуги и ресурсы.

В клиент-серверных системах БД размещается на отдельном наиболее производительном компьютере, на этом же компьютере разворачивается сервер (это и есть СУБД). На клиентских станциях достаточно установить сравнительно нетребовательное к ресурсам пользовательское ПО и настроить сетевой доступ к серверу СУБД. Работа клиент-серверных систем принципиально отличается от работы в системах «файл-сервер». Теперь вместо перекачки файлов с БД клиентский компьютер отправляет серверу запрос, построенный на основе языка SQL. Получив и обработав инструкцию SQL, сервер возвращает клиентскому компьютеру результаты ее выполнения, например выборку определенных данных (рис. 2.3).

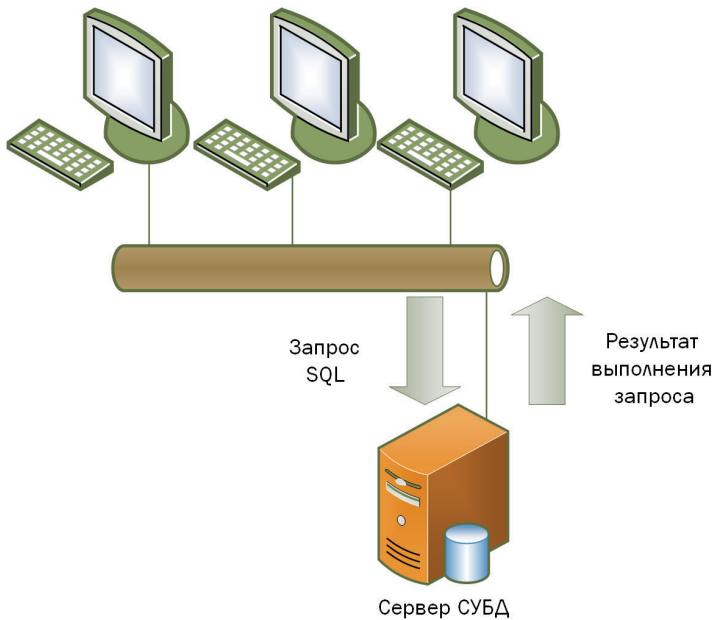


Рис. 2.3. Архитектура «клиент-сервер»

Уже несколько десятилетий клиент-серверное построение БД является доминирующим. Причин тому несколько:

- 1) значительно повышается доступность БД. Сервер представляет собой открытую систему, поэтому клиентские компьютеры могут функционировать под управлением различных операционных систем и с различным ПО;
- 2) выделенный сервер СУБД в состоянии обеспечить параллельную многопользовательскую обработку данных;

- 3) основные правила поддержки целостности и непротиворечивости данных описываются на одном сервере СУБД, клиентские приложения никаким образом не в состоянии обойти эти правила;
- 4) экономно расходуется пропускная способность компьютерных сетей;
- 5) благодаря централизованному хранению данных сравнительно просто поддерживать единые для всех правила безопасности БД;
- 6) наличие стандарта на основной язык общения SQL обеспечивает широкие возможности доступа к серверу БД из программного обеспечения различных производителей;
- 7) упрощены вопросы обслуживания и администрирования БД.

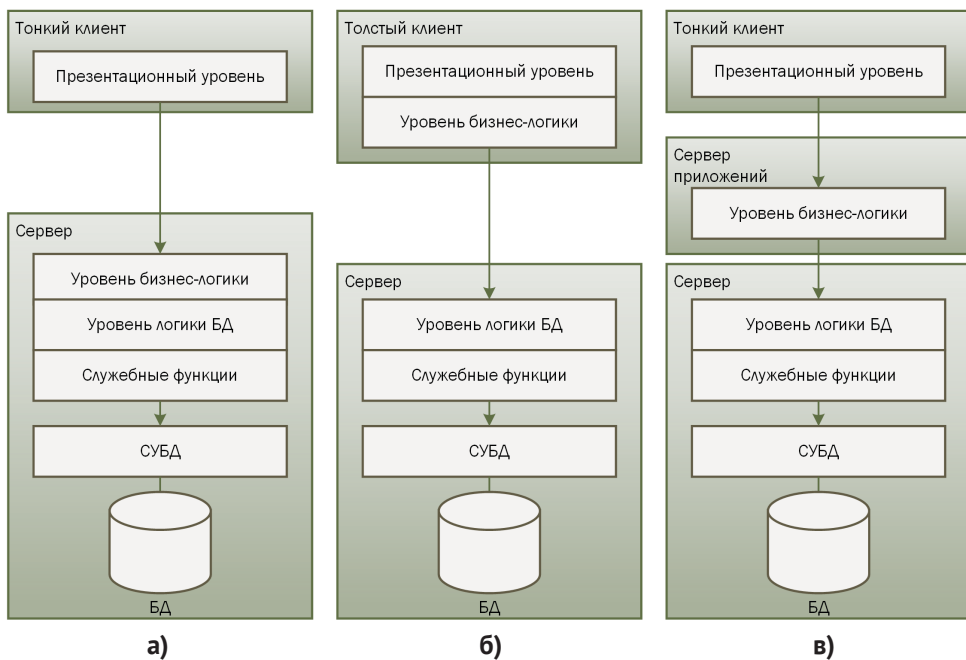


Рис. 2.4. Популярные модели распределения функций между сервером и клиентом

Предусмотрено несколько подходов к распределению обязанностей между сервером и клиентом. В простейшем случае (рис. 2.4(а)) работает модель **тонкий клиент** (thin client). Это тот вариант, когда клиентское ПО максимально упрощено и представляет собой только интерфейсную часть, состоящую из форм ввода и просмотра данных. Тонкий клиент фактически отвечает лишь за презентацию данных, поэтому можно

сказать, что он не умеет думать и вся программная логика сосредоточена на стороне сервера. Есть и обратное решение – **толстый клиент** (fat client). В противовес своему тонкому собрату, толстый клиент старается максимально облегчить работу сервера (рис. 2.4(б)). Например, реализует дополнительные бизнес-правила, производит сортировку полученных данных на клиентской стороне, выполняет вспомогательные расчеты, проверяет синтаксис инструкций SQL и т. п. Проектирование полнофункционального толстого клиентского приложения значительно сложнее, чем разработка его упрощенного собрата. Но в конечном счете мы получаем существенную прибавку в производительности, а это весьма важно в больших многопользовательских БД.

К настоящему времени существует несколько сотен СУБД, предназначенных для работы по модели клиент-сервер. В России наибольшей популярностью пользуются MySQL, Oracle, SQL Server компании Microsoft, InterBase компании Embarcadero, PostgreSQL, Informix компании IBM, NetWare SQL фирмы Novell и т. д.

Замечание

Более подробно об архитектурном решении клиент-сервер мы поговорим в главе 20.

Многоуровневые решения

Предусмотрены и более сложные модели распределения функций между сервером и клиентом. Например, между сервером СУБД и клиентом может появиться еще одно дополнительное звено – **сервер приложений** (рис. 2.4(в)). Сервер приложений обычно отвечает за соблюдение бизнес-правил БД, для этого реализуется несколько прикладных функций, которые представляются клиенту в виде отдельных служб. Клиент не может обратиться к СУБД напрямую, вместо этого он запрашивает интересующий его сервис у сервера приложений.

При создании промежуточного звена между клиентом и сервером разработчики часто пользуются технологией Архитектура брокера общих объектных запросов (Common Object Request Broker Architecture, CORBA). Технология CORBA создавалась специально для систем с распределенной обработкой информации. Ядром системы выступает брокер объектных запросов (Object Request Broker, ORB), выполняющий посредническую функцию между клиентом и сервером.

Изюминка CORBA в том, что интерфейсная часть всех описанных в ней компонентов отделена от их реализации. Благодаря этому

связывающиеся при посредничестве CORBA сервер и клиентские приложения могут разрабатываться на любых языках программирования и функционировать под управлением различных ОС, лишь бы соблюдалось единственное условие – они должны уметь общаться на языке определения интерфейсов (Interface Definition Language, IDL).

Многоуровневые проекты БД могут создаваться и на альтернативных технологиях, в частности на основе распределенной многокомпонентной модели объектов (Distributed Component Object Model, DCOM), протоколе простого доступа к объектам (Simple Object Access Protocol, SOAP), предназначенного для обмена данными в распределенной децентрализованной среде, технологии сокетов и т. д.

Несмотря на то что трехзвенная архитектура позволяет создавать гибкие и универсальные решения, она используется значительно реже, чем двухзвенная. Причин тому несколько, но основная из них в том, что создание многоуровневых клиент-серверных проектов под силу только хорошо подготовленным разработчикам, как следствие подобные проекты стоят значительно дороже, по сравнению с классической архитектурой «клиент-сервер».

Распределенная система

Системы управления распределенными базами данных (СУРБД) – одно из направлений развития современных технологий хранения данных (см. главу 22). Подобные системы предполагают организацию работы с данными, распределенными между несколькими серверами, которые, в свою очередь, могут быть удалены друг от друга на значительные расстояния (рис. 2.5). Любой из серверов должен обладать возможностью обрабатывать как локальные запросы пользователей в своей подсети, так и слаженно работать с внешними запросами, поступающими из других подсетей. В свою очередь, клиент вправе получать одновременный доступ к интересующим его данным, физически размещенным на разных серверах (в идеале с логической точки зрения клиент вообще может полагать, что он работает с единой, а не распределенной БД).

Распределенные системы могут включать как однотипные (гомогенные), так и разнотипные (гетерогенные) БД. СУБД обслуживают фрагменты БД, между которыми может осуществляться частичная или полная репликация данных. Кроме того, могут использоваться специальные серверы, предназначенные для управления всей системой (например, на них может содержаться глобальный системный каталог,

схема фрагментации данных между СУБД, средства обработки распределенных запросов и т. д.).

Замечание

На данный момент существует множество коммерческих и бесплатных традиционных СУБД, предназначенных для работы в архитектуре клиент-сервер, однако вы не найдете в продаже программный продукт с условным названием «СУРБД». Дело в том, что проектирование и внедрение распределенных БД является весьма сложным процессом, который выполняется высокопрофессиональными разработчиками в интересах конкретного заказчика.

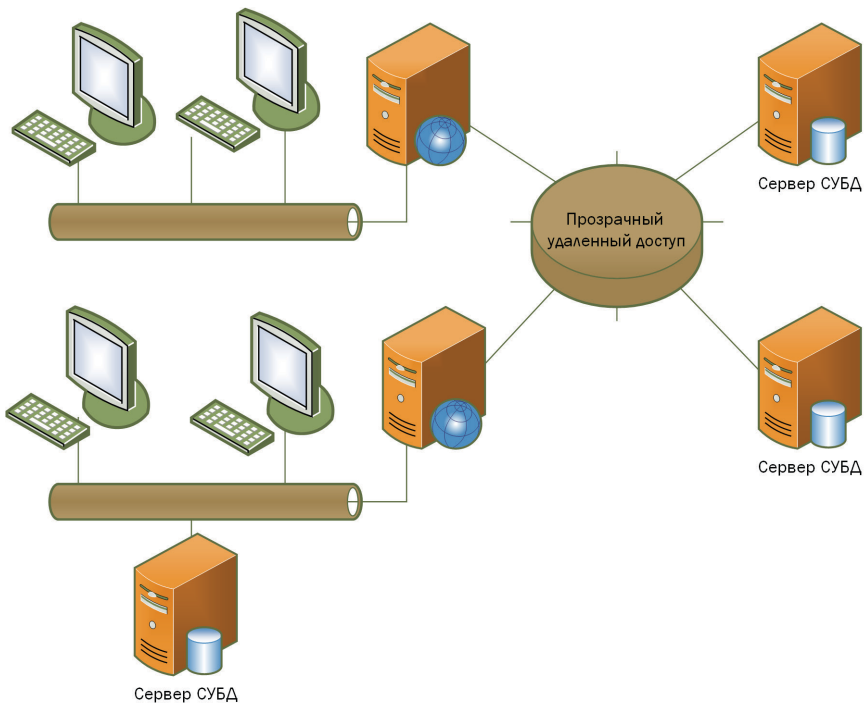


Рис. 2.5. Архитектура распределенной обработки

Резюме

Состав и особенности функционирования любой сложной системы, в том числе и СУБД, определяются стоящими перед ней задачами. Еще на заре разработки реляционной модели доктор Эдгар Фрэнк Кодд сформулировал функциональные обязанности классической СУБД, чем

определил вектор развития этой области информационных технологий. За более чем 40 лет существования реляционных баз данных идеи Кодда совершенствовались учеными, инженерами и программистами, в результате к сегодняшнему дню разработан широкий перечень эффективных программных и аппаратных решений, применяемых в СУБД.

К настоящему времени существует более 3 сотен разнообразных СУБД (<https://db-engines.com>), способных удовлетворить самого привередливого разработчика.

Среди множества архитектурных решений доступа к БД на сегодняшний день доминирующее положение занимают клиент-серверные системы с централизованным хранением данных. Вместе с тем развитие сетевых технологий и возрастающие требования предприятий и организаций к быстрому и надежному доступу к данным привели к появлению распределенных БД, основанных на идее децентрализации.

Вопросы для самопроверки

1. Дайте определение СУБД.
2. Какие различия имеются между системой файлов и СУБД?
3. Каким образом классифицируются СУБД?
4. Какие основные функции должна выполнять СУБД?
5. Что понимается под термином метаданные?
6. Опишите назначение компонентов СУБД.
7. Какие архитектурные решения доступа к БД вам известны?
8. Почему архитектура файл-сервер не подходит для многопользовательских БД?
9. Как могут распределяться задачи между клиентом и сервером БД?
10. Приведите примеры современных клиент-серверных СУБД.

Глава 3

Персонал и пользователи БД

Совместное использование данных на предприятии, с одной стороны, предоставляет сотрудникам уникальную возможность всегда находиться в курсе текущего состояния дел, а с другой – порождает неминуемые конфликты интересов за право владения и обработки данных. Необходимость поддержки на предприятии идеологии совместной работы с данными привела к появлению специальных отделов, отвечающих за информатизацию.

Как известно, при введении новых штатных единиц в структуру предприятия руководство никогда не остается проигравшей стороной и щедро наделяет своих работников самым широким спектром прямых и косвенных обязанностей. Но в любом случае, в сферу интересов самого обобщенного отдела информатизации, эксплуатирующего информационную систему, как минимум должны входить следующие группы функций:

- 1) **сервисные функции**, направленные на поддержку сотрудников предприятия (конечных пользователей) в области доступа к данным;
- 2) **производственные функции** по удовлетворению специфичных информационных потребностей сотрудников предприятия в соответствии с решаемыми ими задачами;
- 3) **функции обеспечения информационной безопасности**, в первую очередь направленные на запрет доступа к конфиденциальной информации несанкционированных лиц (особенности обеспечения информационной безопасности при работе с БД мы обсудим в *главе 10*).

На отдел информатизации возложен весьма широкий перечень задач, за решение которых отвечают разноплановые специалисты, но раз наша книга посвящена базам данных, мы в полном праве сузить состав

должностных лиц отдела информатизации и их должностные обязанности до уровня работы с СУБД. В таком случае можно говорить о следующем контингенте сотрудников отдела информатизации [25, 38]:

- администратор данных, АД (Data Administrator, DA);
- администратор(ы) БД, АБД (Database Administrator, DBA);
- разработчики БД;
- прикладные программисты.

Безусловно, картина окажется неполной, если, распределяя обязанности по работе с БД, мы забудем упомянуть пользователей. Пусть они не относятся к отделу информатизации, но ведь именно в их интересах и создается информационная система предприятия, так что дополним список еще одним пунктом:

- конечные пользователи.

Предложенная на рис. 3.1 схема отображает основные направления деятельности персонала предприятия при работе с БД.

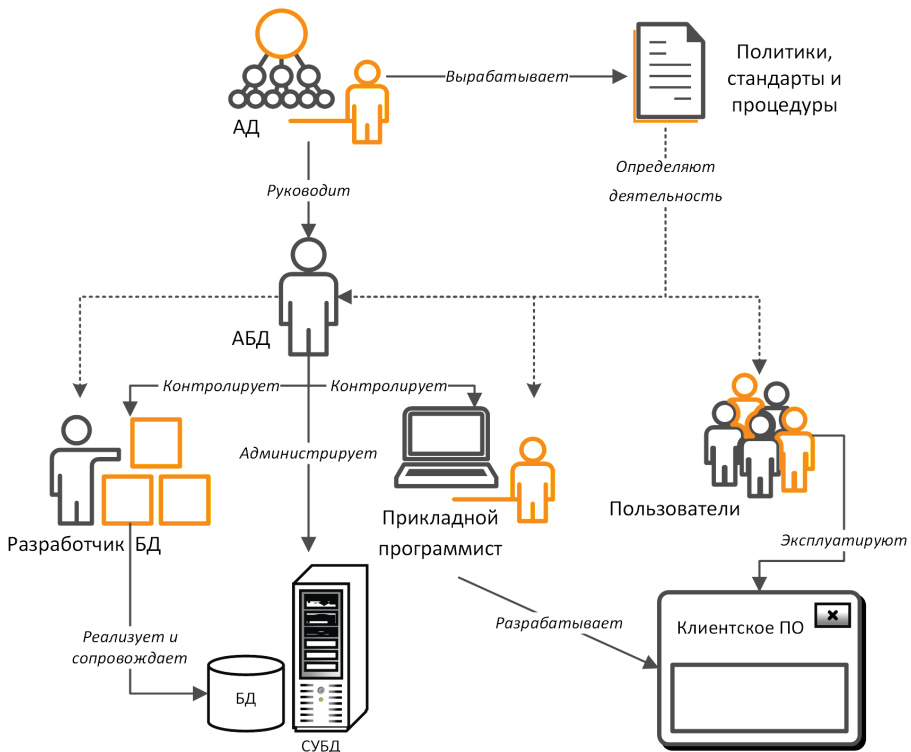


Рис. 3.1. Общее направление деятельности персонала при работе с БД

Администратор данных

С точки зрения управленческих функций в отделе информатизации на вершине пирамиды располагается администратор данных. Это наиболее опытный (и, соответственно, дорогостоящий) специалист, в первую очередь выполняющий организаторские функции, кроме того, АД отвечает за управление данными (планирование БД, разработка стандартов и бизнес-правил) и за концептуальное проектирование базы данных.

Замечание

Администрирование данных связано не только с СУБД. В ведении администратора находятся все корпоративные информационные ресурсы, в том числе и некомпьютеризированные данные.

Среди задач, решаемых администратором данных, наиболее значимы следующие [25, 38, 44]:

- оказание помощи в формировании корпоративной стратегии построения информационной системы (ИС);
- контроль за внедрением и неукоснительным исполнением на всех этапах жизненного цикла ИС политик, процедур и стандартов компании по корректному созданию, использованию и распространению данных;
- разработка концептуальной модели данных;
- планирование процесса создания БД;
- выявление требований предприятия к используемым данным;
- установка правил сбора, хранения и представления данных;
- определение политики информационной безопасности;
- разработка концептуальной и логической моделей БД;
- взаимодействие с АБД и программистами с целью обеспечения соответствия разрабатываемой БД и клиентских приложений существующим стандартам и требованиям предприятия;
- контроль за постоянной модернизацией ИС и ПО;
- обеспечение полноты требуемой документации;
- взаимодействие с конечными пользователями ИС.

Для небольших предприятий вполне допустимо, чтобы обязанности АД совмещал начальник отдела информатизации или АД выступал внештатным сотрудником, в последнем случае он приглашается на этапе подготовки проекта новой или в преддверии глубокой модер-

низации существующей информационной системы. Одним словом, администратор данных наиболее востребован на ранних стадиях жизненного цикла проекта БД.

Особой прерогативой АД выступает определение политик, стандартов и процедур при разработке, внедрении и сопровождении БД.

Под политикой (policies) понимается ключевая руководящая инструкция, подлежащая неукоснительному соблюдению персоналом. Например, в период проектирования АД вводит политику именования объектов БД (таблиц, столбцов, индексов, хранимых процедур и т. д.), которой должны придерживаться АБД и программисты. Другим примером политики, на этот раз на этапе эксплуатации БД, станет требование, чтобы все пользователи могли работать с данными исключительно на основе парольного доступа.

Стандарты выступают развитием политик, они формируют их более детализированное описание. Допустим, стандарты укажут, как следует именовать первичные ключи таблиц или минимальную длину пароля пользователя.

Наконец, процедуры представляют собой тексты инструкций, содержащие последовательность действий, которые необходимо выполнить персоналу в том или ином случае работы с БД. Если под управлением АД трудятся адекватные разработчики, возможно, и окажется достаточным устного инструктажа по соглашению об именовании, но, безусловно, должна существовать письменная инструкция по правилам предоставления доступа к БД нового пользователя.

Администратор базы данных

Администратор базы данных отвечает за реализацию базы данных (в том числе за физическое проектирование), за обеспечение безопасности и целостности данных, за сопровождение системы, а также за обеспечение максимальной производительности СУБД и приложений.

К задачам администратора баз данных относятся:

- выбор целевой СУБД;
- разработка логической модели БД;
- обеспечение требуемой защиты и целостности данных;
- тестирование БД;
- введение БД в эксплуатацию;
- подготовка пользователей к работе с БД;
- системное администрирование СУБД;

- контроль производительности;
- резервное копирование;
- восстановление;
- репликация
- и т. д.;
- ведение технической документации БД;
- сопровождение БД.

В ситуации, когда на предприятии эксплуатируется несколько типов СУБД и/или информационная система эксплуатирует две и более БД, вполне вероятно, что для поддержания системы в работоспособном состоянии потребуется несколько АБД.

Разработчики баз данных

Нуждающиеся в проектировании и последующем сопровождении больших БД крупные компании в штатный состав отделов информатизации могут вводить разработчиков БД. На более мелких предприятиях, возможно, эти функции возлагаются на АБД или делегируются компании, у которой приобретает программное обеспечение.

Разработчики БД, руководствуясь концептуальной моделью данных и принятыми на предприятии политиками и стандартами, осуществляют разработку логической, а затем и физической моделей данных.

Прикладные программисты

В современных клиент-серверных системах БД развертываются на стороне сервера. Для того чтобы услугами БД смогли воспользоваться обычные сотрудники предприятия, им, как правило, прямой доступ к серверу не предоставляется. Вместо этого специально для пользователей на высокоуровневых языках программирования создается отдельное программное обеспечение – клиентские приложения.

Разработка клиентского ПО относится к зоне ответственности прикладных программистов. В их задачи входит:

- организация доступа клиентского приложения к БД;
- проектирование интерфейса пользователя;
- наполнение приложения заданным функционалом;
- тестирование и отладка приложений;
- сопровождение приложений в период их эксплуатации.

Конечные пользователи

Главным потребителем услуг информационной системы предприятия выступают конечные пользователи. Их можно классифицировать по следующим параметрам:

- по уровню управления:
 - высшее руководство предприятия;
 - менеджмент среднего звена;
 - рядовые сотрудники;
- по уровню знаний:
 - опытный пользователь;
 - обычный пользователь;
 - начинающий пользователь;
- по частоте обращения к данным:
 - редко;
 - периодически;
 - часто.

Вполне возможно, читатель столкнется и с другими способами классификации пользователей БД. Однако вне зависимости от выбранного подхода надо понимать, что вся деятельность персонала отдела информатизации – начиная от администратора данных и заканчивая стажером – должна быть нацелена на обеспечение доступа пользователей предприятия (а возможно, и внешних пользователей, например покупателей в интернет-магазине) к информационным ресурсам организации. Поэтому персонал отдела информатизации, и в первую очередь АД и АБД, обязан на постоянной основе:

- осуществлять сбор и анализ требований пользователей к ИС предприятия с целью ее развития и совершенствования;
- поддерживать целостность данных в БД, в том числе путем выявления и предотвращения некорректных действий пользователей;
- своевременно устранять проблемы доступа пользователей к данным;
- обеспечивать безопасность данных, обрабатываемых пользователями;
- осуществлять обучение и поддержку пользователей.

В решении перечисленных задач в первую очередь нашими основными помощниками станут опытные пользователи. Однако такая катего-

рия, как опытный пользователь, если и не исключение из правил, то, по крайней мере, редкое явление, и поэтому при проектировании и реализации программного обеспечения надо ориентироваться, что ПО зачастую будет эксплуатироваться персоналом, не обладающим глубокими познаниями в области информационных технологий. Посему наша БД, с одной стороны, должна обладать максимально дружественным интерфейсом, а с другой – уметь противостоять не только действиям разнообразных хакеров, но и сопротивляться некорректным операциям, исходящим от вполне легальных (из-за этого, возможно, еще более опасных) пользователей.

Резюме

В ответ на возрастание роли информационных технологий и на изменение организации данных на современных предприятиях стали создаваться специализированные отделы, выполняющие функции администрирования данных. В штатную структуру таких отделов входят такие важные должности, как администратор данных (АД) и администратор баз данных (АБД).

Основной задачей АД является общий контроль за процессом создания и совершенствования информационной системы предприятия, таким образом, он в большей степени выполняет функцию организатора. В свою очередь, АБД ориентирован на техническую сторону – поддержку БД в работоспособном состоянии.

Практическую сторону разработки БД реализуют разработчики БД и прикладные программисты. Первые отвечают за серверную сторону проекта, вторые – за клиентские приложения.

Основным потребителем услуг, предоставляемых БД, выступают конечные пользователи. Именно в их интересах проектируется, реализуется и поддерживается в работоспособном состоянии информационная система предприятия.

Вопросы для самопроверки

1. Какие функции возлагаются на информационный отдел предприятия?
2. Объясните, какие задачи решает:
 - а) администратор данных;
 - б) администратор базы данных;
 - в) разработчик БД;
 - г) прикладной программист.

3. Сравните функции АД и АБД.
4. Какие современные языки программирования вам известны? Прокомментируйте их особенности.
5. Каким образом можно классифицировать пользователей БД?
6. Какие действия должны выполняться для поддержки пользователей предприятия?
7. Какими, на ваш взгляд, знаниями, навыками и умениями должен обладать пользователь БД?

Глава 4

Реляционная модель

Создателем реляционной модели считается математик Эдгар Фрэнк Кодд (Edgar Frank Codd, 1923–2003). Датой рождения реляционной теории можно считать июнь 1970 года. Именно тогда Кодд (на тот момент времени сотрудник одной из лабораторий корпорации IBM) опубликовал свою знаменитую статью «Реляционная модель данных для больших совместно используемых банков данных» (*Codd E. F. A Relational Model of Data for Large Shared Data Banks. CACM 13: 6*), в которой впервые прозвучал столь популярный сегодня термин «реляционная модель».

Замечание

За свой вклад в науку в 1981 году Эдгар Кодд был удостоен престижной премии имени Тьюринга в области теоретических основ вычислительной техники.

Первопричиной возникновения нового по тем временам подхода к проектированию баз данных послужили существенные ограничения предыдущих моделей. Ни сетевая, ни иерархическая модели не были способны просто и доступно описывать подлежащие учету данные. Кодд сумел объединить на первый взгляд несовместимые вещи – с одной стороны, реляционная модель опиралась на строгие математические выкладки, а с другой – была понятна рядовому пользователю, состоящему в конфронтации даже с таблицей умножения.

Замечание

Модель данных представляет собой логически взаимосвязанную совокупность описаний объектов данных и операторов, управляющих этими объектами. Применяя термин «модель», мы акцентируем внимание на то, что это абстрактная конструкция. Чтобы модель превратилась в действующую базу данных, она должна быть физически реализована средствами какой-либо СУБД. Нетрудно догадаться, что реляционная база данных может получиться только из реляционной модели.

Модель данных не представляет существенного интереса для обычного пользователя, но без нее не обойтись разработчику БД. Ведь это не что иное, как чертеж будущей БД. Как ни один строитель не начнет строить дом без чертежа, так и ни один программист не станет реализовывать физическую БД без ее подробной схемы. Фундамент реляционной модели построен на понятии сущности и связей между сущностями.

Реляционной модели данных посвящено много фундаментальных трудов, в которых подробно изложены все ключевые аспекты модели. Читателю стоит хотя бы в обзорном порядке ознакомиться с работами ведущих специалистов в этой области [15, 19, 25, 38, 44]. Задача автора несколько прозаичнее – изложить тот минимум информации, без которого понимание реляционной модели будет просто невозможно.

Любая область знаний, будь это математика, радиоэлектроника, физика, химия или информатика, по мере своего развития формирует свой набор терминов и определений. Без этого никак не обойтись, ведь в своих ученых беседах специалисты оперируют понятиями, не имеющими аналогов в окружающем мире. Зачастую размерность специфичных понятий разрастается до такой степени, что существенная часть времени, отводимого на изучение интересующего нас предмета, затрачивается только на формирование словарного запаса. С точки зрения сложности семантики, системы управления базами данных в общем и реляционная модель в частности не являются исключением из правил. Эта сравнительно новая область знаний практически мгновенно успела обзавестись специфичным набором словарных конструкций. Многие термины перекочевали в БД из лексикона программистов, часть определений позаимствована из теории множеств, в тезаурусе разработчиков моделей данных даже нашлось место для понятий, обычно встречающихся в справочниках по философии.

Сущность и атрибуты

Любой более-менее подготовленный пользователь твердо знает, что базы данных предназначены для хранения и обработки информации¹. Если в руки такому человеку попадет база городских телефонов, то, даже не запуская программу, он сможет предположить, что в ней он обнаружит телефонные номера граждан и организаций. База данных го-

¹ Строго говоря, термины «информация» и «данные» не являются синонимами. В информационных технологиях под информацией понимают как данные, так и знания о каких-либо объектах, событиях и т. п.

родской библиотеки обязана содержать информацию об авторах и их бессмертных произведениях. Столкнувшись с базой данных авиакомпаний, с высокой степенью вероятности можно утверждать, что в ней мы найдем информацию о расписании полетов, марках самолетов и городах, в которые они доставляют пассажиров.

То, что кажется элементарным для стороннего наблюдателя, для профессионального разработчика БД может стать очень непростым делом. На самом первом этапе проектирования будущей БД он ломает голову над двумя ключевыми вопросами:

1. Что подлежит хранению в БД?
2. Какие связи между данными следует поддерживать в БД?

Для получения ответа на первый вопрос разработчик должен составить первоначальный перечень типов сущностей. **Тип сущности** (entity) – это класс однотипных объектов, о которых следует хранить информацию в БД. Немного забегаая вперед, оговоримся, что на заключительной стадии проектирования типы сущности превратятся в реляционные таблицы. В 90 % случаев сущность – это какой-то объект, отвечающий на вопрос «Кто?» или «Что?». В базе данных библиотеки на роль сущностей претендуют такие одушевленные и неодушевленные объекты, как автор, книга, издательство, читатель. Совсем не обязательно, чтобы сущность соответствовала объектам реального мира, напротив, она может моделировать и отвлеченные понятия, допустим выраженные в особых отношениях между сущностями. Именно в таком случае сущность ловко маскируется под глагол. Как правило, это происходит в тех случаях, когда учету подлежат некоторые события. Например, когда читатель получает (это и есть глагол) книгу из библиотечного фонда, логика базы данных требует сохранить информацию о том, кому, когда и какая из книг была выдана. При более подробном рассмотрении выясняется, что при построении первоначального списка мы просто не выявили такой тип сущности, как читательская библиотечная карточка, в которую и делаются все перечисленные выше пометки. Библиотечная карточка поддерживает двустороннюю связь между сущностями «читатель» и «книга».

Если тип сущности – это класс однотипных объектов, то как же величать отдельный объект, входящий в этот класс? В реляционной модели такой объект называют **экземпляром сущности**. Экземпляр сущности (или просто сущность) – это конкретная реализация объекта. Для сущности «автор» – это Уильям Шекспир и Лев Толстой, для сущности «книга» это «Ромео и Джульетта» и «Война и мир».

Замечание

Экземпляр сущности – это то, о чем (или о ком) следует хранить информацию в БД. В реальной базе данных экземпляру сущности будет соответствовать одна строка таблицы.

У любой сущности есть свои характеристики. У сущности «самолет» из базы данных авиакомпании таких характеристик сотни. Это марка, максимальная скорость полета, число посадочных мест, грузоподъемность, дата выпуска, цвет фюзеляжа и многое-многое другое. У сущности «читатель» отличительных признаков не меньше, а может быть и больше, чем у летательного аппарата. В счет может пойти все, начиная с цвета глаз и заканчивая размером обуви. Однако в библиотечной базе данных вряд ли стоит учитывать рост, вес и отпечатки пальцев клиента. Гораздо полезнее знать имя, домашний адрес и номер телефона. Отличительные свойства сущности называют **атрибутами**.

Атрибуты могут быть **простыми** (например, название города, в который летит самолет) и **составными** (т. е. включать в себя несколько простых атрибутов). В качестве составного атрибута выступает адрес читателя. Здесь найдется место почтовому индексу, городу, улице, дому и номеру квартиры. Атрибут может быть **производным**, то есть полученным в результате проведения какой-то операции над другими атрибутами. Подобные атрибуты часто встречаются при построении различных бухгалтерских систем, например значение взимаемого с сотрудников предприятия налога составляет определенный процент от заработной платы этих служащих. Можно столкнуться с **многозначными** атрибутами, они описывают те свойства сущности, в которых одновременно хранится несколько значений. Например, у одного человека может быть несколько адресов проживания, несколько контактных телефонных номеров и т. п. Есть и особая разновидность атрибутов, без которых невозможно построение реляционной модели, – это атрибуты, однозначно идентифицирующие экземпляр сущности. На роль такого атрибута в состоянии претендовать индивидуальный номер налогоплательщика или уникальный заводской номер самолета. Позднее подобные атрибуты превратятся в ключи реляционных таблиц.

Построив исчерпывающий список типов сущностей и подготовив перечень их атрибутов, разработчик модели базы данных должен задуматься над особенностями реализации типов сущностей. Для этого в первую очередь следует разобраться с особенностями физического представления каждого из атрибутов типа сущности, для этого предназначены тип данных и домен.

Тип данных и домен

Отправной точкой процесса построения любой системы хранения и обработки данных по праву считается выбор **типа данных**. Вне зависимости от того, какой язык программирования вы изучали, при построении новой программы самым первым шагом становится рассмотрение типов данных, имеющихся в распоряжении системы. Типы данных определяют порядок хранения данных в памяти компьютера. Физическая концепция хранения данных зависит от особенностей конкретной платформы, на базе которой функционирует программное обеспечение. Поэтому в разных операционных системах, в разных системах управления базами данных число байт, отводимых для описания целого или вещественного числа, может отличаться. Например, в 32-разрядных приложениях размерность целого числа INTEGER равняется 4 байтам. В 64-битных системах это утверждение может оказаться ошибочным, ведь здесь проще адресовать 8-байтную переменную.

Типизация хранимых значений не просто указывает на размерность в байтах, которую должна выделить система для размещения в памяти того или иного значения. Преследуемая цель еще более значима. Типизация определяет, какие операции могут быть осуществлены с теми или иными данными. Среда разработки не позволит новичку передать результаты деления в целочисленную переменную, ведь в этом случае есть риск потерять дробную часть результата. Система станет отчаянно сопротивляться, если мы попробуем просуммировать символьный и вещественный типы данных. Пусть даже в символьной переменной хранится числовое значение. В последнем случае перед проведением математической операции необходимо повести преобразование данных. Точно так же СУБД не допустит ввода текстовых данных в поля таблиц, предназначенных для хранения числовой информации.

Подытожим все вышесказанное. Понятие тип данных интегрирует в себе три компоненты:

- ограничение множества принадлежащих типу значений;
- дефиниция применяемых к типу набора операций;
- определение способа отображения (внешнего представления) значений типа.

Как показала практика, при обработке данных в современных информационных системах возможностей обычной типизации становится недостаточно. Примеров тому великое множество. Формально при определении атрибута, хранящего дату рождения сотрудников како-

го-нибудь предприятия, следует ограничиться типом данных DATETIME. Но такой подход не способен в полной мере отразить задачи отдела кадров. Стандартизированный в SQL тип данных DATETIME допускает ввод даты в диапазоне от 1/01/0000 до 12/31/9999. Вряд ли начальник отдела кадров примет на работу сотрудника в несовершеннолетнем возрасте или, наоборот, ровесника Ивана Грозного. Поэтому атрибут «дата_рождения» должен уметь сопротивляться некорректным, с точки зрения пользователя, но абсолютно верным с точки зрения типа данных DATETIME данным. Есть и другой пример. Допустим, что в базе данных хранится почтовый индекс. Каждый поклонник эпистолярного жанра знает, что это комбинация из 6 цифр, отражающая региональную принадлежность почтового отделения. Порядок описания индекса не сложен для человека – надо заполнить все шесть значений, не четыре, не пять и не семь, а только шесть. Теперь попробуйте подыскать стандартный тип данных, способный однозначно решить эту простейшую задачу. Тип INTEGER не подходит, потому что обладает существенно большим диапазоном значений. Строка из шести элементов CHAR также не вполне адекватна, ведь она позволяет пользователю вводить любые (не только цифровые) символы. Наконец, еще один пример. Формально номер вашего телефона можно типизировать как целое число – INTEGER. Целые числа поддерживают все основные арифметические операции. Однако никому в здравом уме и в доброй памяти не придет в голову идея суммировать значения телефонных номеров или вычитать из одного номера другой, хотя формально это обычные целочисленные значения. Просто очень сомнительна практическая ценность полученного результата. В подобных и во многих других схожих ситуациях одна лишь типизация бессильна – она в недостаточной степени поддерживает человеческую логику. Если автор смог убедить читателя, что возможностей типа данных недостаточно для описания характеристик атрибутов, закончим с примерами и поговорим о том, как это ограничение можно преодолеть.

Фундаментальная для современных языков программирования идея типизации в реляционной модели получила дальнейшее развитие. На сцену вышло новое понятие – **домен** (domain). Домен ни в коем случае не подменяет понятие типизации, а выступает в качестве дополнительного ограничителя обрабатываемой в базе данных информации. Ограничение накладывается за счет введения дополнительных логических правил. Так, принимая нового сотрудника на работу, при вводе даты рождения можно контролировать его возраст путем элементарного сравнения текущей системной даты компьютера и даты рожде-

ния. Обслуживая атрибут почтового индекса, описываемого шестью символами типа CHAR, наложим дополнительное ограничение на диапазон приемлемых значений – наш домен должен пропускать только цифровые знаки. Надо сразу оговориться, что логические правила являются не столько описанием отдельного домена, сколько средством поддержания целостности данных в масштабах всей базы данных. Допустим, вы вводите почтовый индекс города Москвы, а в атрибут города передаете Монреаль. В этом случае разумная база данных должна намекнуть нам, что мы сегодня перетрудились и следует прерваться на чашечку кофе... Но для того, чтобы система смогла выявить подобную логическую ошибку, разработчик БД должен построить не только строгий набор доменных ограничений, но и реализовать развитую систему поддержания корпоративной целостности.

Замечание

Тип данных определяет особенности физического хранения данных. Домен, опираясь на концепцию типа данных, несет дополнительную нагрузку – отвечает за поддержание логических правил описания данных. Благодаря доменным ограничениям в БД реализуется одно очень важное качество – доменная целостность данных.

В профессиональных СУБД доменные ограничения могут накладываться на несколько атрибутов одновременно, таким образом одновременно контролируя непротиворечивость хранящихся в них данных.

Связь

Реляционная модель призвана не только описать перечень типов сущностей, но и отразить особенности взаимодействия между ними. Взаимодействие возникает там, где между типами сущностей проявляются отношения, выраженные в виде глагола, например продавец оформил заказ. Здесь «продавец» и «заказ» – это две сущности, а связь между ними просматривается в глаголе «оформил».

Различают три типа связи между сущностями: «один к одному», «один ко многим» и «многие ко многим». Появление в реляционной БД связи «один к одному» – крайне редкий случай. Чаще всего он свидетельствует о том, что разработчик не вполне корректно определился с атрибутами и вместо одного из атрибутов создал лишний тип сущности. На практике наиболее распространена связь «один ко многим», например автор написал много книг, завод выпустил много автомобилей, в одном отделе работает много сотрудников (рис. 4.1).



Рис. 4.1. Отношение «один ко многим»

Связь «многие ко многим» встречается реже: офисы разных компаний размещены в разных городах, множество издательств опубликовало произведения различных авторов и т. п. Реляционная модель напрямую не поддерживает отношение «многие ко многим», поэтому его приходится разбивать на два отношения «один ко многим», но об этом поговорим позднее.

Есть еще один классификационный признак связи – степень (размерность). При выявлении связей между сущностями разработчик обычно встречается с унарными (unary relationship) и двойными связями (binary relationship), но в реальной жизни имеют место связи и большей кратности. Например, выражение «авиакомпания доставляет пассажира в город» подразумевает тернарную связь (ternary relationship). Здесь мы сталкиваемся с тремя существительными-сущностями: «авиакомпания», «пассажир» и «город» – и всего одним глаголом «доставляет». Дальнейшее развитие модели такой базы данных во многом зависит от опыта и интуиции разработчика, ведь разные пассажиры вправе летать в разные города, пользуясь услугами разных авиакомпаний. Ограничившись лишь услугами двойных связей, можно утратить суть сохраняемой информации – какая из авиакомпаний в какой именно из городов доставила конкретного пассажира. К вопросу построения связей класса «многие ко многим» мы вернемся в следующей главе при обсуждении процесса ER-моделирования, а пока обсудим еще одну особенность связи.

Реляционная модель позволяет связывать сущность саму с собой – задавать рекурсивную связь. Такой, на первый взгляд несколько необычный, способ взаимодействия оказывается весьма полезным при «выращивании» древообразных конструкций, например описывающих организационную структуру предприятия, завода или, скажем, университета. Здесь четко просматривается иерархия, в которой одна и та же сущность может быть главной по отношению к подчиненным эле-

ментам и одновременно находиться в подчиненном состоянии к более высокому по рангу узлу. Так, дочерними элементами по отношению к деканату факультета выступают кафедры этого факультета, в свою очередь, деканат подчиняется ректорату учебного заведения.

Отношение

Дословный перевод термина «relation» – отношение. Что понимается в реляционной теории под отношением?

Вся информация, собранная в реляционной БД, рассматривается пользователем как совокупность взаимосвязанных двумерных таблиц, в каждой из которых хранятся данные о строго определенном типе сущностей. Реляционная таблица состоит из строк и столбцов. Каждая строка содержит информацию о конкретной сущности; так, в таблице «VENDORS» из базы данных склада мы увидим строки с данными о компаниях, производящих электронику и бытовую технику (рис. 4.2). Но в этой таблице не должно оказаться ни единой строки о клиентах нашей фирмы – покупателях, ведь это другой тип сущности. Каждый столбец реляционной таблицы предназначен для хранения определенного атрибута сущности. Для атрибута «VNAME» (полное название фирмы) разработчик таблицы применит текстовый тип данных размерностью в 50–60 символов. А на атрибут «EMAIL» будут наложены дополнительные доменные ограничения, проверяющие наличие символа «@» и других правил построения почтового адреса.

Столбец
(атрибут, поле)

Имя столбца

Кортеж
(строка, запись)

Таблица VENDORS

VENDORS_ID	VNAME	VSHORTNAME	Country	EMAIL
1	Samsung Group	Samsung	Южная Корея	support@sam...
2	Acer	Acer	Китайская Рес...	support@acer...
3	Intel Corporation	Intel	США	support@intel...
4	ASUSTeK Com...	ASUS	Китайская Рес...	support@asus...
5	HP Inc.	HP	США	support@hp.com

Рис. 4.2. Реляционная таблица

Замечание

Терминология реляционных баз данных складывалась из нескольких областей знаний, поэтому в качестве синонимов к понятию «строка» можно услышать слова

«запись» и «кортеж». У термина «столбец» также есть два тождественных понятия: «атрибут» и «поле». Ведя речь о таблице, вполне корректно применять термин «отношение».

Для того чтобы таблица (отношение) получила высокую честь называться «реляционной», необходимо, чтобы она соответствовала ряду строгих требований:

- уникальность имен столбцов внутри таблицы;
- каждая строка хранит информацию об отдельной сущности;
- строки таблицы отличаются друг от друга хотя бы единственным значением, что позволяет однозначно идентифицировать любую строку такой таблицы;
- на каждом пересечении строки и столбца хранится атомарное значение;
- строки и столбцы таблицы могут обрабатываться в любом порядке, независимо от хранящихся в них данных.

Замечание

Немного забегаая вперед, заметим, что о таблице, соответствующей представленным выше требованиям, говорят, что она приведена к первой нормальной форме (см. главу 7).

Некоторые из перечисленных требований нуждаются в комментариях. Уникальность имен столбцов требуется соблюдать только в рамках одного отношения, и ничто не запрещает использовать одинаковые имена в различных таблицах. Так, если название фирмы будет храниться в столбце «VNAME» таблицы «VENDORS», то при необходимости вполне допустимо задействовать такое же имя столбца в других отношениях. Для обращения к требуемому столбцу средствами языка SQL требуется указать название соответствующей таблицы и ее столбца. Например, запрос

```
SELECT VNAME FROM VENDORS;
```

возвратит все названия фирм, поставляющих нам товары.

Суть запрета нахождения в реляционной таблице одинаковых строк легко объяснима. Во-первых, дублирование данных противоречит теории множеств, ведь классическое множество должно состоять из различных элементов. Во-вторых, повтор идентичных данных приводит

к бессмысленному расходованию дискового пространства и оперативной памяти. В-третьих, дубликаты данных вносят существенную путаницу в работу программы. Представьте, что в студенческой группе учатся два тезки, скажем Иван Иванов и Иван Иванов. Один из них – круглый отличник, а другой – патологический двоечник и прогульщик. Как следствие первого надо перевести на следующий курс, а второго – отчислить. Если записи об этих студентах не будут отличаться хотя бы на микрон (точнее, на бит), то деканат окажется в патовой ситуации, руководству факультета будет не понятно, какую из строк таблицы следует удалить, а какую оставить. Обеспечение требования уникальности строки достигается путем ввода в таблицу дополнительных атрибутов, в частности отчества, даты рождения, номера и серии паспорта и т. п. На основе одного или нескольких уникальных столбцов формируются особые столбцы, называемые **ключевыми**.

На пересечении строки и столбца должно находиться единственное значение. Это значение должно быть атомарным, т. е. неделимым. Примером нарушения атомарности может стать попытка записи в одну ячейку таблицы группы значений, например нескольких названий фирм. Подобный подход также затрудняет процесс поиска и обработки информации.

Последнее из перечисленных требований об обработке строк и столбцов в любом порядке в меньшей степени заботит разработчика БД, так как оно реализуется автономно средствами современных СУБД. Отметим, что отсутствие требования упорядоченности не усложняет, а наоборот – значительно упрощает процесс разработки БД. Поэтому, заполняя список студентов, поступивших в вуз, не следует заботиться о соблюдении алфавитной последовательности при внесении новых строк. При необходимости получить упорядоченную по фамилии (или любому другому атрибуту/атрибутам) ведомость можно буквально в два счета, добавив в инструкцию SQL ключевое слово `ORDER BY` и перечень атрибутов сортировки. Порядок вывода столбцов точно так же назначается программистом средствами SQL.

Замечание

В следующих главах мы покажем, что понятие «отношения» несколько шире, чем просто реляционная таблица. В качестве отношения можно рассматривать объединение двух и более таблиц или выборку нескольких столбцов из одной таблицы, полученных в результате выполнения запроса `SELECT`.

Ключи

В составе любого реляционного отношения всегда должен присутствовать столбец (или группа столбцов), содержимое которого предназначается для хранения значений, способных уникальным образом идентифицировать строку таблицы. Допустим, мы проектируем таблицу для отдела кадров, предназначенную для хранения сведений о сотрудниках предприятия. Таблица станет содержать интуитивно понятный набор столбцов (ФИО, дата рождения, специальность, образование и т. п.), описывающих человека. Какие столбцы следует сделать ключевыми?

На роль ключевых вряд ли смогут претендовать столбцы, хранящие фамилию и имя человека. Если на вашем предприятии на данный момент времени не работают два полных тезки, это совсем не означает, что это исключено в будущем. Поэтому разработчику стоит сразу включить в таблицу поля с безусловно уникальной информацией для описываемой сущности, например индивидуальный номер налогоплательщика для человека или номер двигателя и кузова для автомобиля. Но и с такими данными могут произойти неприятности. Допустим, что мы выбираем в качестве ключа атрибут, содержащий информацию о серии и номере паспорта гражданина. Вряд ли кто со мной не согласится, что это уникальные данные и, на первый взгляд, вполне подходящие в качестве кандидатуры на ключ. Но паспорт может быть заменен (в результате потери, стихийного бедствия, плановой смены паспортов и т. п.). В результате у человека появляется новый паспорт с другой серией и номером, это влечет за собой сложную процедуру изменения значений первичного ключа и связанных с ним внешних ключей во всей базе данных. Если приходится исправлять две-три строки в таблице, то ничего страшного. А если паспорта заменили у всех граждан страны?

Поэтому чаще разработчики добавляют к таблице атрибуты, предназначенные для хранения искусственной информации, обычно это целочисленные поля автоинкрементного типа. Идея работы подобного столбца в том, чтобы при внесении в таблицу очередной строки значение атрибута получало приращение (обычно плюс единица, хотя во многих СУБД это значение настраиваемое). Простая и, как следствие, надежная логика работы счетчика исключает любую вероятность повторения значений в автоинкрементных полях – каждая новая строка в ключевое поле запишет новое значение. Если вы ненадолго вернетесь к рис. 4.1, на котором представлено отношение между сущностями «Отдел» и «Сотрудник», то увидите, что для сущности «Отдел» в качестве

первичного ключа введен атрибут «ключ_отдела», а для сущности «Сотрудник» – «ключ_сотрудника».

Определение

Первичным ключом (PK, Primary Key) называют ключ, отвечающий за уникальную идентификацию строки в реляционной таблице.

Помимо первичных ключей, в таблицах активно применяется еще одна разновидность ключевого поля – **внешний ключ** (FK, Foreigner Key). Благодаря внешнему ключу мы получаем возможность строить реляционные связи между таблицами. На рис. 4.1 внешний ключ имеется у сущности «Сотрудник», это атрибут «ключ_отдела». Если мы хотим указать, что сотрудник Арбузов не покладая рук трудится в бухгалтерии предприятия, то мы делаем следующее:

- узнаем значение первичного ключа у строки «Бухгалтерия» (в нашем примере это 2);
- передаем это значение во внешний ключ строки с сущностью «Арбузов».

Таким образом, поле внешнего ключа сущности «Сотрудник» может содержать любое из значений первичного ключа сущности «Отдел».

Целостность данных

Мы уже знаем, что база данных – это не просто сосуд, который постепенно заполняется произвольной информацией. БД еще призвана на установленном разработчиком уровне абстракции отражать взаимосвязанные объекты реального мира. Чем точнее данное отражение, тем совершеннее база данных. Зеркало также призвано отражать объекты реального мира, но каждый из нас знает, что оно склонно к искажению действительности. Объект, попавший в зазеркалье, уже не тот, что находится снаружи. Поэтому, кроме функций зеркала, БД должна содержать некоторый набор правил, которые позаботятся о том, чтобы данные всегда находились в согласованном состоянии.

Определение

Целостность данных (data integrity) – соответствие значений всех данных базы данных определенному непротиворечивому набору правил.

Можно выделить три базовых класса правил, призванных поддерживать целостность данных в реляционной БД:

- 1) целостность доменов;
- 2) целостность сущностей;
- 3) ссылочная целостность.

Кроме того, существует понятие корпоративной целостности, это не что иное, как реализация в БД бизнес-правил, присущих конкретному предприятию.

Целостность доменов

Целостность доменов поддерживается за счет механизма доменных ограничений. Мы о нем уже говорили в начале главы. Это как раз тот случай, когда описание домена включает некие логические правила, отбраковывающие некорректные значения, которые так настойчиво стремятся попасть в атрибуты нашей таблицы. В простейшем случае допустимые значения могут быть просто перечислены, например «понедельник», «вторник» и т. д. Может быть объявлен диапазон допустимых значений, например от 2 до 5, если речь идет об оценках учеников. Все зависит от возможностей СУБД, в которой мы создаем наши проекты.

Особая роль в поддержании доменной целостности отводится особому определителю NULL. NULL обозначает неопределенность или неизвестность. При проектировании БД разработчики выявляют обязательные атрибуты, без которых работа БД невозможна, и атрибуты, к которым применяются пониженные требования, например которые разрешено заполнить позднее или не трогать вовсе. Допустим, что вы вводите в БД анкетные данные сотрудников и у кого-то из них пропущена дата рождения. Что делать? Поставить дату на глазок? Это может привести к искажению фактов (БД призвана отражать, а не искажать объекты реального мира), и впоследствии отдел кадров отправит своего сотрудника раньше на пенсию. Если система способна понять, что ячейка таблицы хранит не значение, а неопределенность, то можно избежать подобной неприятности. Есть и другой пример. В вуз зачислено несколько сотен абитуриентов, но зачетные книжки и студенческие билеты на них пока не заполнены. Номера билетов и «зачеток» – это обязательные, безусловно подлежащие учету данные. Что же? Прекратить ввод уже известной информации (имена, адреса, номера телефонов и т. п.), пока первокурсникам не раздадут все документы? А может стать так, что документы невозможно оформить, пока в БД не появится

информация о новых студентах. Получается замкнутый круг. Посему грамотно спроектированная база данных просто обязана позволить нам ввести имеющиеся на сегодня данные, отложив на завтра заполнение неопределенных атрибутов.

Определитель NULL может оказаться полезен при организации связи между таблицами, если поле внешнего ключа допускает неопределенное значение, то это служит признаком, что связь необязательна.

Замечание

Определитель NULL – это ни в коем случае не аналог значения 0 или, например, текстовой строки, заполненной пробелами! У него иная задача, NULL указывает программисту, что значение атрибута таблицы не определено или неизвестно.

Благодаря определителю NULL базы данных работают в трехзначной логике. Поэтому в БД к известным каждому программисту значениям TRUE (истина) и FALSE (ложь) добавляется NULL (неопределенность).

Целостность сущностей

Как недвусмысленно следует из названия, целостность сущностей направлена на обеспечение внутреннего единства отдельной сущности. Вне зависимости от того, что мы намерены хранить в наших таблицах, следует соблюдать краеугольное правило – каждая строка таблицы обязана быть уникальной. А для этого нужно контролировать корректность первичного ключа отношения (если вы подзабыли, то напомним: первичный ключ – это минимальный уникальный идентификатор кортежа в отношении). Суть требования к первичному ключу проста – во входящих в его состав атрибутах не должно содержаться ни одного определителя NULL. Такое пожелание логично. Ведь если мы допускаем, что во входящем в состав первичного ключа столбце может находиться неопределенное значение, то это означает, что поле перестает поддерживать механизм однозначной идентификации строки таблицы.

Замечание

Ни один из столбцов, входящих в состав первичного ключа, не должен допускать ввода определителя NULL!

Помимо поддержания корректности первичного ключа, целостность сущностей в состоянии обеспечить широкий спектр дополнительных сервисных возможностей. Их перечень определяется профессионализ-

мом разработчика базы данных. Например, было бы хорошо научить БД контролировать непротиворечивость значения, вводимого в атрибут, на основе значений, имеющих в других атрибутах таблицы или даже в атрибутах других таблиц. База данных железнодорожной кассы обязана уведомить оператора, что пассажир покупает на свое имя два билета в абсолютно противоположных направлениях на одно и то же время. База данных отдела кадров металлургического завода не должна пропустить на вредную работу к мартеновской печи несовершеннолетнего. База данных в деканате не переведет на следующий курс студента, не сдавшего летнюю экзаменационную сессию.

Ссылочная целостность

Реляционная база данных – это не только таблицы с информацией, но и логические связи между ними, а нарушение связи между таблицами может разрушить всю БД. Именно поэтому на стражу реляционных связей поставлено третье правило целостности данных, получившее название ссылочной целостности. Смысл правила в следующем: внешний ключ не может осиротеть, ему всегда должен соответствовать первичный ключ в главной таблице.

Поводов для нарушения связи предостаточно. Во-первых, удаление строки в главной таблице. Допустим (рис. 4.1), мы удаляем строку «Отдел кадров» в таблице «Отделы предприятия». В результате в подчиненной таблице «Сотрудники» оказываются брошенными сотрудники Костенко и Елецкова, ведь их внешний ключ по-прежнему будет хранить значение уже несуществующего первичного ключа. Во-вторых, при добавлении новой записи в таблицу «Сотрудники» во внешнее поле «ключ_отдела» может попасть значение, которому нет соответствия в главной таблице. В-третьих, ссылочная целостность может подвергнуться опасности при редактировании ключевых полей в главной или подчиненной таблице. В современных СУБД все перечисленные проблемы предотвращаются автоматически, но это не означает, что рядовому разработчику баз данных не стоит задумываться над поддержанием ссылочной целостности в своих проектах.

Корпоративная целостность

В реальной жизни для поддержания реляционной БД в непротиворечивом состоянии перечисленных выше правил обеспечения целостности обычно не хватает. У любого предприятия, магазина, учреждения существуют свои собственные, зачастую уникальные бизнес-правила.

Например, в БД продуктового склада следует контролировать, чтобы скоропортящиеся молочные товары попадали в холодильные камеры, а картофель и капуста – в овощехранилище. Проектируя БД агентства по продаже недвижимости, следует задуматься над тем, чтобы за агентами по продаже закреплялись дома и квартиры, находящиеся в одном районе города, иначе последним придется вместо осуществления сделок часами простаивать в автомобильных пробках. В БД библиотеки следует предусмотреть ограничения на выдачу очередной книги, если читатель просрочил возврат предыдущей. Для реализации специфичных для каждого из заказчиков бизнес-правил разработчики БД создают дополнительные правила поддержания целостности, называемые правилами корпоративной целостности. Основным средством поддержки корпоративной целостности выступают хранимые процедуры и триггеры.

Реляционная алгебра

Являясь отличным математиком, Э. Кодд сразу понял, что наилучшим способом описания реляционной модели данных станет теория множеств. Может быть, нам и не обязательно в совершенстве владеть всеми тонкостями теории множеств, но с предложениями Кодда нам познаться следует. Поэтому рассмотрим восемь простейших операций реляционной алгебры, определяющих особенности функционирования реляционной модели данных. Итак, нам предстоит разобраться с операциями, осуществляющими:

- 1) σ выборку (selection);
- 2) $\downarrow$ проекцию (projection);
- 3) $\times$ декартово произведение (cartesian product);
- 4) $\cup$ объединение, сложение (union);
- 5) $-$ вычитание, разность (set difference);
- 6) $\cap$ пересечение (intersection);
- 7) $/$ деление (division);
- 8) $\parallel$ соединение (join).

Определение

Реляционная алгебра представляет собой теоретический язык операций, которые на основе одного или нескольких отношений позволяют создавать другое отношение.

Операции выборки и проекции относятся к классу унарных, так как работают с одним отношением. При осуществлении выборки из

исходного множества формируется результирующее подмножество, содержащее только удовлетворяющие определенному условию элементы.

На языке множеств выборку можно сформулировать следующим образом. Пусть существует предикат выборки F , аргументами которого выступают атрибуты реляционной таблицы, состоящей из множества кортежей R . Результатом выборки окажется подмножество кортежей R' , для которых предикат выборки F истинен.

$$R' = \sigma_F(R).$$

Например, с помощью операции выборки мы сможем получить список студентов, родившихся в 2000 году. Взгляните на рис. 4.3. Выборка из таблицы «Студенты» составила три строки.

R – таблица «Студенты»

ID	SURNAME	FNAME	BIRTHDAY	SPECIALITY
1	Орехов	Владимир	11/04/1999	Математика
2	Петровский	Александр	21/11/2000	Математика
3	Кульгина	Оксана	5/01/2000	Ин. язык
4	Самойлов	Евгений	15/07/2001	Информатика
5	Кузьмина	Ирина	2/03/2000	Физика
6	Яковенко	Константин	12/09/2001	Химия
7	Ищенко	Владимир	09/19/2002	Астрономия

R' – выборка из таблицы «Студенты»

2	Петровский	Александр	21/11/2000	Математика
3	Кульгина	Оксана	5/01/2000	Ин. язык
5	Кузьмина	Ирина	2/03/2000	Физика

Рис. 4.3. Демонстрация операции выборки

В отличие от операции выборки (выделяющей горизонтальные элементы исходного множества), операция проекции направлена на получение вертикальной составляющей множества. Допустим, таблица R обладает n атрибутами, а нас интересует подмножество атрибутов с именами $i_1, \dots, i_k$, причем $k \leq n$. Тогда результатом проекции станет

$$\downarrow \\ i_1, \dots, i_k \text{ } R.$$

На этот раз возвращаются все строки из исходной таблицы R , но в новое подмножество войдут не все, а только определенные пользователем столбцы таблицы.

Иллюстрацией операции может стать рис. 4.4, на котором представлена проекция, полученная из таблицы студентов. В итоговое подмножество включены атрибуты с фамилией и именем студентов.

***R* – таблица «Студенты»**

ID	SURNAME	FNAME	BIRTHDAY	SPECIALITY
1	Орехов	Владимир	11/04/1999	Математика
2	Петровский	Александр	21/11/2000	Математика
3	Кульгина	Оксана	5/01/2000	Ин. язык
4	Самойлов	Евгений	15/07/2001	Информатика
5	Кузьмина	Ирина	2/03/2000	Физика
6	Яковенко	Константин	12/09/2001	Химия
7	Ищенко	Владимир	09/19/2002	Астрономия

Проекция

Орехов	Владимир
Петровский	Александр
Кульгина	Оксана
Самойлов	Евгений
Кузьмина	Ирина
Яковенко	Константин
Ищенко	Владимир

**Рис. 4.4.** Демонстрация операции проекции

Декартово произведение открывает набор операций, осуществляемых с двумя множествами кортежей R и S :

$$R \times S.$$

Отношение, получаемое в результате декартова произведения, представляет собой сцепление строк из одного отношения со строками из другого отношения. В результате произведения каждой строке из одной таблицы R присоединяется одна строка из другой таблицы S . На практике подобная операция редко когда оказывается востребованной, ведь в результирующем множестве оказываются все возможные сочетания строк из двух таблиц. Однако из любого правила есть и исключения. Допустим, что студенты хотят получить данные о том, какие учебные дисциплины им предстоит изучать, тогда без декартового произведения не обойтись (рис. 4.5).

***R* – «Студенты»**

SURNAME	FNAME
Орехов	Владимир
Петровский	Александр
Кульгина	Оксана

X

***S* – «Дисциплины»**

DISCIPLINE
Операционные системы
Языки программирования
Методы программирования
СУБД

***R* × *S***

SURNAME	FNAME	DISCIPLINE
Орехов	Владимир	Операционные системы
Орехов	Владимир	Языки программирования
Орехов	Владимир	Методы программирования
Орехов	Владимир	СУБД
Петровский	Александр	Операционные системы
Петровский	Александр	Языки программирования
Петровский	Александр	Методы программирования
Петровский	Александр	СУБД
Кульгина	Оксана	Операционные системы
Кульгина	Оксана	Языки программирования
Кульгина	Оксана	Методы программирования
Кульгина	Оксана	СУБД

Рис. 4.5. Демонстрация операции произведения

Операция объединения проводится только с таблицами R и S , совместимыми по объединению, например с идентичной структурой. В результате мы получим суммарную таблицу, содержащую строки из двух исходных отношений:

$$R \cup S.$$

Представленная на рис. 4.6 операция объединения суммирует записи из таблиц R и S в результирующее отношение. Стоит заметить, что если бы исходные таблицы содержали одинаковые по содержанию строки, то в результате операции объединения из итогового множества должны быть удалены записи-дубликаты.



Рис. 4.6. Демонстрация операции объединения

Операция вычитания позволяет выяснить, какие из строк первой таблицы отсутствуют во второй таблице. Как и в случае сложения, разность работает только с отношениями, совместимыми по объединению (с одинаковой структурой). Предположим, что у нас есть две таблицы R и S , в первой находятся данные о студентах учебной группы, а во второй – данные о студентах, успешно сдавших экзаменационную сессию. Операция вычитания $R - S$ позволит вычислить, кому следует явиться на пересдачу (рис. 4.7).

Операция пересечения может проводиться с таблицами, совместимыми по объединению, действие позволяет выявить строки, общие для двух таблиц. Допустим, что нам следует узнать, какие из студентов успешно сдали экзамен по высшей математике (таблица R) и экзамен по СУБД (таблица S). Пересечение двух отношений $R \cap S$ предоставит нам запрашиваемый результат (рис. 4.8).

Для понимания сути операции деления воспользуемся следующим примером. Допустим, в таблице R содержатся данные о том, какие виды занятий определенных дисциплин ведут педагоги университета. Нам хочется уточнить, кто ведет лабораторные работы по СУБД и практичес-

кие занятия по языкам программирования. Для этого в таблицу S за-
носятся две строки с указанными данными (рис. 4.9) и осуществляется
деление.

$$P = R/S.$$

Что окажется в итоговой таблице P ?



Рис. 4.7. Демонстрация операции вычитания

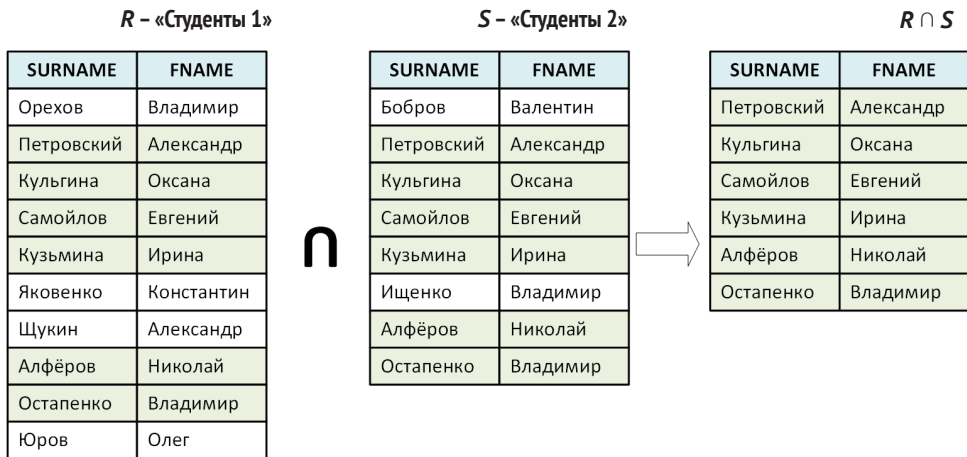


Рис. 4.8. Демонстрация операции пересечения

Для того чтобы усвоить, как работает операция деления, надо запо-
мнить две вещи:

- во-первых, результат деления исходной таблицы R на таб-
лицу-делитель S будет содержать столбцы, отсутствующие в делите-
ле (в нашем случае это атрибут фамилии «SURNAME»);

- во-вторых, в качестве строк в результат P войдут только те записи из делителя S , что при декартовом произведении результата на делитель $P \times S$ содержатся в делимом R .

Таким образом, после деления мы получаем две строки с фамилиями Орлов и Володина. Преподаватель Шуверов не попал в результирующее отношение, так как он не ведет лабораторные по дисциплине СУБД.

R – делимое			S – делитель		P
SURNAME	DISCIPLINE	KIND	DISCIPLINE	KIND	SURNAME
Орлов	СУБД	Лекция	СУБД	Лаб. работа	Орлов
Орлов	СУБД	Лаб. работа	Языки программирования	Практ. занятие	Володина
Орлов	Языки программирования	Практ. занятие			
Володина	СУБД	Лаб. работа			
Володина	Языки программирования	Практ. занятие			
Шуверов	Языки программирования	Практ. занятие			

Рис. 4.9. Демонстрация операции деления

Наконец, мы добрались до наиболее важной реляционной операции – операции соединения. Существует несколько способов соединения отношений (естественное соединение, внешнее соединение, тета-соединение и т. д.), и все они основаны на декартовом произведении, только теперь после перемножения строк двух таблиц на результат накладываются различные дополнительные ограничения. Допустим, в нашем распоряжении имеются две таблицы, в таблице S хранится информация о кафедрах вуза, а в таблице R – информация о преподавателях. Продемонстрируем наиболее распространенную операцию естественного соединения:

$R \bowtie S$.

Таблицы соединяются по общему столбцу, в таблице кафедр R это первичный ключ CHAIR_ID, а в таблице педагогов S это одноименный столбец внешнего ключа (рис. 4.10).

R – «Преподаватели»			S – «Кафедры»			P – Естественное соединение		
SURNAME	CHAIR_ID		CHAIR_ID	CHAIR		SURNAME	CHAIR_ID	CHAIR
Орлов	1	$\bowtie$	1	Высшая математика	$\Rightarrow$	Орлов	1	Высшая математика
Володина	1		2	Физика		Володина	1	Высшая математика
Шуверов	3		3	Информатика		Шуверов	3	Информатика
Калужный	2		4	Химия		Калужный	2	Физика
Аскеров	2					Аскеров	2	Физика

Рис. 4.10. Операция естественного соединения

В результирующей таблице не нашлось места для кафедры химии. Это особенность естественного слияния, в таблице R нет ни одной записи с внешним ключом, ссылающимся на кафедру химии ($\text{CHAR_ID}=4$), поэтому кафедра химии не попала в итоговую таблицу. Если же нам требуется увидеть и «брошенные» corteжи, то вместо услуг естественного слияния необходимо воспользоваться помощью внешнего соединения. Различают правое и левое внешние соединения. На рис. 4.11 представлено правое внешнее соединение. Правым внешним соединением называется соединение, при котором corteжи отношения кафедр S (таблица расположена справа), не имеющие совпадающих значений в общих столбцах отношения преподавателей R , также включаются в результирующее отношение.



Рис. 4.11. Демонстрация правого внешнего соединения

В одной из строк результирующей таблицы в столбце «SURNAME» мы обнаружим определитель NULL. Это произошло из-за того, что в таблице S педагогов не хранится информация о сотрудниках кафедры химии. Тета-соединение определяет отношение, содержащее строки из декартового произведения отношений R и S , удовлетворяющие предикату F :

$$R \bowtie_F S = \sigma_F(R \times S).$$

При этом предикат F может использовать не только оператор равенства (= (в этом случае тета-соединение превращается в уже знакомое нам естественное соединение), но и любой другой оператор сравнения (<, ≤, ≥, >, ≠).

Резюме

Своим появлением на свет реляционная модель данных обязана талантливому специалисту Эдгару Фрэнку Кодду. Именно он предложил использовать для хранения данных четкий и выверенный математичес-

кий аппарат, позволяющий описывать отношения между множествами.

Физическое представление отношения имеет вид двумерной таблицы, состоящей из строк и столбцов. Доктор Кодд сформулировал требования к таблице, благодаря соблюдению которых таблица получает право называться реляционной.

Вопросы для самопроверки

1. Что в БД понимается под типом сущности и сущностью?
2. Какие связи между сущностями могут быть отражены реляционной моделью?
3. Что понимается под термином «тип данных»?
4. Для чего предназначены доменные ограничения?
5. Для чего предназначен первичный ключ?
6. Какие требования предъявляются к реляционной таблице?
7. Каким образом организуется связь между двумя отношениями?
8. Как классифицируются связи между отношениями?
9. Что в реляционной модели понимается под целостностью данных?
10. Какие виды целостности данных вам известны?
11. Почему говорят, что реляционные БД работают в трехзначной логике?
12. Какие операции реляционной алгебры вам известны?

Глава 5

Технология разработки БД

Еще в старозаветные времена появления первых ремесел люди, сами того не замечая, стали создавать технологии. Нашими предками разработана технология выделки кожи, технология выпечки хлеба, технология кройки и шитья одежды и миллионы других технологий. Что уж говорить о нынешнем постиндустриальном мире, все производство в котором немислимо без технологий.

Технологии не обошли стороной и современную информатику. Уже более чем полвека существует такое понятие, как технология программирования – свод правил, применяемых для разработки всего спектра программного обеспечения (ПО), в том числе и баз данных.

Так ли критически важны технологии при разработке БД? В этом с лихвой убедились первые разработчики, пытавшиеся создавать ПО на заре информатизации в 1950–1960-х годах. В отсутствие четких моделей БД и методов их проектирования создание каждой БД превращалось в весьма непростую задачу. В те времена каждый программист создавал технологию для себя, как говорится, с нуля методом проб и ошибок, тратил на процесс годы и далеко не всегда приходил к намеченной цели.

Сегодня, к счастью, можно говорить о том, что технологии разработки БД уже существуют. Именно поэтому предложим читателю, находящемуся еще в самом начале пути, вместо того чтобы изобретать велосипед, получить представление о наработках его предшественников. Это поможет нам избежать их ошибок и существенно ускорить свое профессиональное становление.

Определение

Технология программирования – это совокупность методов и средств, используемых в процессе разработки программного обеспечения.

Технология программирования представляет собой набор технологических инструкций, включающих [21]:

- указание последовательности выполнения технологических операций;
- определение условий, при соблюдении которых выполняется та или иная операция;
- описание самих операций с определением исходных данных, ожидаемых результатов, а также инструкции, нормативы, стандарты, показатели и критерии, методы оценки и т. п.

Таким образом, следуя инструкциям, ранее сформулированным профессионалами своего дела, разработчик программного обеспечения получит больше шансов создать по-настоящему успешный продукт, чем его коллеги, руководствующиеся исключительно собственным пониманием о ступенях процесса проектирования БД.

Роль БД на предприятии

По своей сути любая БД представляет собой скрупулезно построенное хранилище структурированных данных. Само по себе хранилище ни для кого особого интереса не представляет до тех пор, пока оно не интегрируется в состав более сложной надсистемы, называемой информационной.

Определение

Информационная система (Information System, IS) – это система, предназначенная для сбора, корректировки и распространения информации внутри организации и объединяющая в своем составе персонал, оборудование, базы данных и программное обеспечение.

Все элементы информационной системы объединены общей целью – обеспечить поддержку принятия решений по эффективному управлению предприятием, в котором развернута эта система. Почему мы говорим лишь о поддержке принятия решения, а не о принятии решения в целом? Дело в том, что на сегодняшний момент времени еще рано говорить о системах с развитым искусственным интеллектом, поэтому решение принимает не электронная машина, а человек (называемый экспертом) или группа лиц (экспертная группа). Для принятия того или иного решения эксперт нуждается не просто в связанных друг с другом реляционных таблицах, а в специально подготовленных структурированных данных (графиках, аналитических записках, статистических отчетах, прогнозах и т. п.), именно эти данные информационная система запрашивает у БД и затем предоставляет принимающему ре-

шение лицу в удобном для него виде. Именно для этих целей в любой организации создается (зачастую интуитивно) **цикл преобразования информации** (рис. 5.1).

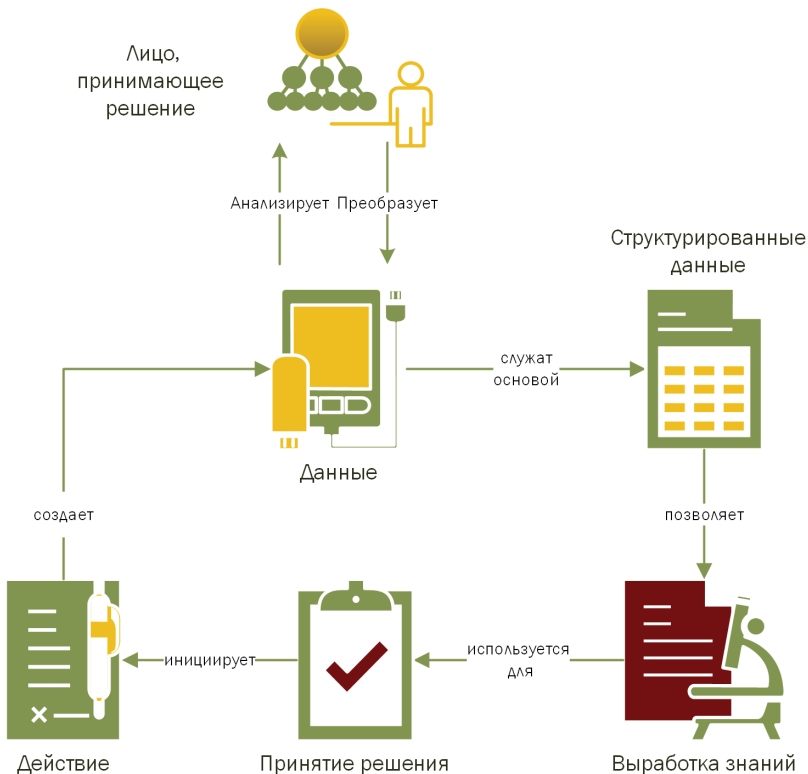


Рис. 5.1. Цикл преобразования информации

Цикл преобразования информации действует непрерывно, обеспечивая функционирование предприятия. Лицо, принимающее решение (руководитель, директор, совет директоров), анализирует и преобразует имеющиеся в его распоряжении данные о состоянии дел в его предприятии. В свою очередь, данные выступают основой для построения специально обработанных (структурированных) данных, в нашем случае это и есть БД. Благодаря БД данные представляются и интерпретируются в удобном для анализа виде. Структурированные данные позволяют вырабатывать новые знания, которые будут использованы для принятия очередного управляющего решения. Появление решения приводит к выполнению определенного действия, а оно естественно порождает новые данные. Круг замкнулся...

Создание информационной системы – это многогранный кропотливый процесс, затрагивающий практически все стороны жизнедеятельности организации. В самом общем случае он включает этапы:

- 1) планирования системы;
- 2) анализа задач системы и требований к системе;
- 3) проектирования системы;
- 4) реализации и ввода в эксплуатацию;
- 5) сопровождения.

Каждый из этапов разделяется на десятки составляющих, над которыми трудятся соответствующие специалисты. Мы не станем отнимать их хлеб, но вновь позволим себе сделать весьма важное замечание – фундаментом любой информационной системы выступает база данных. Опыт подсказывает, что еще никому не удавалось построить отличный дом на плохом фундаменте, поэтому процедура проектирования БД должна быть неразрывно связана с процессом проектирования информационной системы.

Замечание

База данных – хотя и ключевая, но далеко не единственная составная часть информационной системы. Помимо БД, в типичную информационную систему входят: программное обеспечение поддержки БД, аппаратное обеспечение, прикладное ПО и эксплуатирующий систему персонал.

Прежде чем мы приступим к рассмотрению этапов технологии проектирования БД, зададим себе вопрос: какие показатели служат безусловным признаком того, что предприятию или организации необходимо задуматься о разработке и внедрении своей собственной информационной системы?

Признаком того, что пора приступить к разработке БД, обычно выступает тот факт, что существующая на данный момент система (если она вообще есть) перестала удовлетворять требованиям компании. Инициатива обращения за помощью к разработчику БД, безусловно, должна исходить от руководства предприятия, но это не означает, что сотрудники нижних звеньев управления и производства не могут быть заинтересованы в совершенствовании системы. Одним словом, на предприятии должна сложиться революционная ситуация, когда начинает действовать классическая формула «верхи не могут, а низы не хотят».

Сравнение начала разработки новой БД с революцией не столь сильно преувеличено. Как правило, у любой, проработавшей хотя бы пару лет компании уже существует своя более и менее налаженная система работы. Эта система может быть полностью основана на бумажных документах, а может быть и частично автоматизирована. Поэтому руководителю предприятия (в большинстве случаев консерватору) необходимы очень веские причины, для того чтобы в буквальном смысле слова выкинуть на помойку пусть плохенькую, но пока еще работающую систему. Большинство этих причин он обнаружит в любом учебнике по организации бизнеса, самая главная из них меркантильная – увеличение прибыли и снижение расходов. Но для большинства людей, привыкших работать по старинке, это не столь наглядно. Для них прибыль и расходы четко ассоциированы с производительностью труда, экономией ресурсов и еще с сотней причин, но не каждый руководитель видит четкую связь между ростом доходов и эфемерной информационной системой. С таким бизнесменом можно долго дискутировать о повышении эффективности контроля и управления над компанией, получаемом предприятием после разработки и внедрения БД. Но пока не произойдет революция в мышлении руководства предприятия, разработчику БД даже не стоит предлагать свои услуги.

Желание руководства компании разработать и внедрить на предприятии информационную систему – необходимое, но не достаточное условие для начала работы. Для создателя БД, особенно на этапе проектирования, очень важна поддержка простых сотрудников предприятия. Дело в том, что дирекция обычно владеет верхней частью информационного айсберга, а это только внешняя сторона Луны. То, что происходит на местах, руководству по объективным причинам неизвестно, ведь вникнуть во все детали не хватит и жизни. Все мелочи известны рядовым сотрудникам, естественно, в рамках своих обязанностей. Технолог знает, какие ингредиенты входят в состав выпускаемой продукции, кладовщик умеет принимать готовую продукцию на хранение, водитель знает маршруты доставки заказчикам и т. п. Именно из деталей складывается целостная картина всех протекающих в стенах компании бизнес-процессов, поэтому игнорировать мелочи не стоит.

Не стоит забывать самый главный вопрос, который просто необходимо задать потенциальному заказчику, готовому заплатить за разработку и внедрение в своей компании новой информационной системы. Дословно он звучит так: «Чем вас не устраивают имеющиеся на рынке программные продукты?» Было бы наивно полагать, что именно мы станем первопроходцами в вопросах разработки БД для отдела кадров,

склада или библиотеки. Немного порывшись в интернете, наверняка можно обнаружить десяток программ по заданной тематике. Ответ на вопрос о программном обеспечении стороннего производителя очень важен. Ведь может случиться так, что руководство компании просто не знает о положении вещей на рынке ПО. Для разработчика БД это скорее минус, чем плюс. Почему? Хотя бы потому, что спустя полгода или год с начала выполнения программистами заказа руководство компании «неожиданно» выяснит о существовании уже готовой БД, написанной одним из конкурентов. В итоге программисту намекнут, что в его услугах больше нет нужды. Поэтому всегда следует быть в курсе продукции потенциальных конкурентов, это, по крайней мере, сэкономит драгоценное время...

Замечание

Есть существенная разница между универсальным программным продуктом стороннего производителя и программным обеспечением, разработанным специально для компании. Различие ровно такое, как между костюмом, пошитым в массовом порядке на фабрике, и платьем, скроенным портным специально для вас. Первый продукт необходимо долго и упорно адаптировать к условиям конкретной компании, второй продукт сразу придется впору. Кроме того, наличие в распоряжении компании своего собственного разработчика (да еще и «скованного» контрактом на сопровождение БД) позволит ей оперативно парировать все сбои в работе БД и быстро вносить изменения и доработки.

Жизненный цикл базы данных

База данных не только функционирует внутри информационной системы, но и развивается вместе с этой системой, поэтому в начале 80-х годов был введен термин **жизненный цикл базы данных** (Database Live Cycle, DBLC).

Определение

Жизненным циклом БД называют период от момента появления идеи создания БД до момента завершения его поддержки разработчиком или организацией, выполнявшей сопровождение.

Состав процессов жизненного цикла БД, как, впрочем, и всего остального программного обеспечения, регламентируется международным стандартом ISO/IEC 12207:2008 Systems and software engineering – Soft-

ware life cycle processes (Системная и программная инженерия – Процессы жизненного цикла). У нас с вами нет возможности подробно изложить концепцию этого стандарта, однако постараемся выделить важнейшие его элементы, относящиеся к жизненному циклу БД.

На рис. 5.2 представлены наиболее существенные этапы жизненного цикла БД, эта схема отчасти станет путеводителем по излагаемому в этой книге материалу. Вне всякого сомнения, специалист по проектированию БД способен представленную схему расширить, добавив в нее дополнительные элементы и связи, но вряд ли у него поднимется рука удалить хотя бы один из предложенных на рисунке блоков, ведь здесь собран тот золотой минимум, обойтись без которого нельзя.

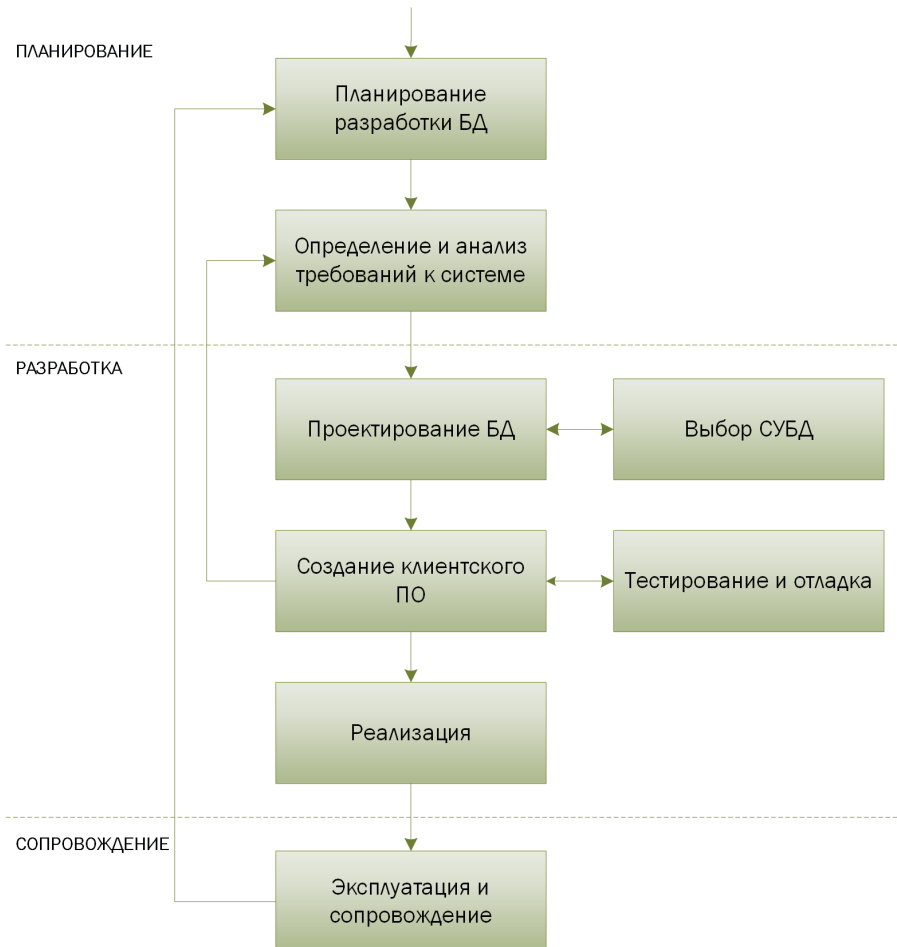


Рис. 5.2. Жизненный цикл проекта БД

Нашу беседу начнем с самого главного секрета – процесс проектирования БД начинается не с включения компьютера, а с остро заточенного карандаша и пачки листов бумаги. Скажем больше: чем больше вы изведете бумаги на первых этапах проектирования, тем потом меньше придется корпеть за клавиатурой.

Этап планирования разработки БД

На этапе планирования разработки БД проектировщик должен получить ответы на ряд предварительных вопросов:

- Какой объем работы придется выполнить?
- Какой состав сил и средств потребуется для выполнения намеченного объема работ?
- Сколько времени потребуется на разработку и реализацию проекта?
- Сколько будет стоить весь проект?

Безусловно, ответы на перечисленные вопросы окажутся весьма приблизительными, но уже по ним вы получите очень важные ориентиры как для себя, так и для заказчика проекта. Все показатели (объем работ, необходимые силы и средства, время и стоимость) должны в равной степени устраивать как проектировщика, так и заказчика, причем каждая из сторон должна понимать, что это всего лишь предварительные расчеты.

Замечание

В идеале этап планирования должен закончиться оформлением юридического документа, представляющего собой договор о намерениях проектировщика и заказчика разработать, реализовать и ввести в эксплуатацию программный продукт.

Этап определения и анализа требований к системе

Этап определения и анализа требований к системе – едва ли не самый значимый во всем жизненном цикле БД. На этом этапе должна быть собрана вся информация, необходимая для проектирования БД. Это очень важно, поэтому повторяю: в результате выполнения этапа в распоряжении проектировщика БД должна оказаться вся информация о компании и протекающих в ней бизнес-процессах. Если у вас это не получится, то расплачиваться за некорректно отработанный этап придется до конца жизненного цикла проекта. Форма оплаты самая

разнообразная – от бесконечных мелких переделок до полного свертывания работ.

В качестве иллюстрации последствий для проекта, разработка которого была осуществлена без учета всех требований заказчика, позволим себе перефразировать пример, приведенный в одной из книг общепризнанного гуру в области объектно-ориентированного программирования Гради Буча [16]. Вряд ли заказчику уже возведенного 100-этажного здания придет в голову мысль попросить строителя перестроить фундамент. Но почему-то заказчику программного продукта, вдруг неожиданно вспомнившего о какой-то дополнительной неучтенной детали проекта, кажется вполне логичным потребовать от программиста перестроить фундамент уже готовой программы. Если читатель уже имеет опыт разработки программного обеспечения, то он наверняка на собственном опыте знает, что зачастую легче написать программу заново, чем пытаться перевернуть все с ног на голову в уже запущенном в работу проекте.

С запугиванием покончено. Надеемся, что у читателя возросло чувство ответственности за этап определения и анализ требований к системе. Переходим к формированию списка вопросов, ответы на которые надо получить на втором этапе жизненного цикла. Вне зависимости от масштабов нашей БД (быть может, вы автоматизируете билетную кассу провинциального театра, а может, вам в руки попала трансконтинентальная торговая сеть) было бы здорово сразу получить следующую информацию:

- 1) цели и задачи компании;
- 2) организационно-штатная структура компании;
- 3) модель бизнес-процесса компании.

Это еще далеко не весь перечень, но пока переведем дух и обсудим первые три пункта списка. Цели и задачи компании необходимы для формирования нашего понимания направления ее основной деятельности. Зная целевую функцию, можно, пусть даже пока на упрощенном уровне, предсказать, что именно захочет получить от будущей БД заказчик.

Обсудим второй пункт нашего списка. Знания организационной структуры предприятия просто необходимы при определении информационных потоков внутри компании. Кому-то нужны отчеты, кому-то аналитика, кто-то хочет владеть всеми данными, кому-то достаточно знать лишь детали. Более того, если компания не может похвастаться грамотно спроектированной компьютерной сетью, то рассматривайте

построение (модернизацию) сети как одну из частных задач проекта БД. Тогда вам придется доказать руководству, что размещение хостов, коммутаторов, маршрутизаторов и прочего оборудования увязывается с географическим расположением объектов компании и с ее логическими информационными потоками. Иначе вы рискуете тем, что идеально спроектированная БД окажется развернутой не в узловой точке схождения потоков с информацией, а где-нибудь на задворках системы... После этого вы столкнетесь с необъяснимым снижением производительности сервера и с разрывами транзакций в самых неожиданных местах.

В идеале к организационной структуре компании неплохо добавить и штатную. Она не окажется помехой, даже если вы всего-навсего автоматизируете небольшой участок предприятия, например склад фабрики. Опыт подсказывает, что к концу проекта руководство неожиданно захочет знать, какой из кладовщиков получал товар по накладной, кто из начальников цехов не вернул на склад остатки строительных материалов и кого, в связи со всем вышесказанным, следует в ближайшее время лишить премии...

Третий пункт списка самый сложный во всем этапе определения и анализа требований. Все, о чем мы говорили до сего момента, вполне укладывается в модель знаний специалиста по информационным технологиям. Здесь задействованы или родные для программиста специальные знания, или понятные каждому общедоступные сведения. Но моделирование бизнес-процесса предприятия даже для суперспециалиста по IT – это действительно проблема. Причина тому – слабые знания в той предметной области, для обслуживания которой предназначена будущая БД. Недостаток знаний – это объективная реальность, и с ней ничего не поделаешь. Вы можете быть превосходным программистом и высекать искры из клавиш компьютера ночи напролет, но как только вы столкнетесь с задачей автоматизации реального производства, торгового предприятия, медицинского учреждения или пожарного депо, не стоит завышать свою самооценку. Чем быстрее придет понимание, что ваши знания в этой области стремятся к нулю, тем будет лучше для вас! Если вам так не кажется, то, пожалуйста, перечитайте этот абзац заново. Ведь излишняя самонадеянность погубила уже немало проектов.

Проблема некомпетентности разработчика БД в предметной области компании может быть решена несколькими способами. Самый лучший, но практически не реализуемый подход заключается в срочном поступлении в вуз на специальность экономиста (инженера-строителя, провизора, пожарного и т. п.). Правда, к тому моменту времени, когда вы приобретете достаточный уровень знаний в области бизнес-процессов

компании, ваши услуги вряд ли окажутся востребованными. Поэтому рекомендуется воспользоваться другим путем – вовлечь в процесс проектирования БД максимальное число специалистов компании.

Вовлечение сотрудников предприятия в проект совсем не означает, что теперь вместо выполнения своих прямых должностных обязанностей они в рабочее время станут рисовать модель будущей БД. Потребуется несколько иной подход. Вы получите всю требуемую информацию путем:

- опроса основных специалистов компании;
- анализа обязанностей сотрудников;
- изучения документов на рабочих местах, в отделах и службах компании, в особенности документов, претендующих на роль отчетных (ведомости, накладные, заказы, заявки и т. д.);
- наблюдения за процессом функционирования компании, особенно в области документооборота;
- анкетирования сотрудников компании.

Конечно же, собранные подобным образом сведения окажутся абсолютно не структурированными, но это все равно лучше, чем отсутствие информации вообще. Кстати, будьте внимательны, в своей совокупности полученная информация может считаться конфиденциальной, что налагает на разработчика БД дополнительную ответственность.

Опытный, разбирающийся в специфике работы компании разработчик БД может сразу заметить изъяны в требованиях руководства и пользователей к проекту и попытаться усовершенствовать полученную информацию. Однако для начинающего проектировщика более логичным решением станет внесение правок (некорректные на его взгляд) требования в присутствии заказчика.

Пока мы обсудили только первые три пункта этапа определения и анализа требований к системе, а теперь переходим к оставшимся:

- 4) выделение границ и возможностей проекта;
- 5) анализ смежных бизнес-процессов с целью выявления перспектив дальнейшего совершенствования БД.

Качество выполнения четвертого и пятого пунктов этапа определения и анализа требований к системе находится в прямой зависимости от опыта и интуиции разработчика БД. Вы только что построили модель бизнес-процессов компании, а теперь необходимо найти компромисс между излишней детализацией, с одной стороны, и недопустимым

упрощенчеством – с другой. Умению балансировать ровно посередине научить невозможно, в этом случае наши дела обстоят ровно как у канатоходца – мастерство приходит со временем (вместе с шишками и синяками).

Замечание

Очерчивая границы проекта, обязательно задумайтесь о перспективах его дальнейшего развития. Всегда рассматривайте текущий проект как элемент пазла, к которому завтра наверняка захочется присоединить очередной элемент.

Представленное выше описание этапа определения и анализа требований к системе дает всего лишь упрощенное представление об этом этапе жизненного цикла БД. На сегодняшний день разработана достаточно эффективная методология по разработке требований к программному обеспечению и управлению этими требованиями. К сожалению, в рамках данной книги мы не сможем даже поверхностно рассмотреть этот важный вопрос, однако вам помогут книги специалистов в этой области [12].

Внимание!

Логическим завершением этапа определения и анализа требований к системе должен стать проект технического задания на разработку БД (см. главу 26). Указанный проект согласовывается с участвующими в проекте руководителями основных отделов и служб компании и утверждается руководством компании (заказчиком БД). В окончательное техническое задание проект превратится в середине этапа проектирования БД.

Этап проектирования БД

Представленный на рис. 5.2 одним прямоугольником этап проектирования БД на самом деле не столь прост. Специалисты разделяют этот этап на три фазы (рис. 5.3):

- 1) концептуальное проектирование;
- 2) логическое проектирование;
- 3) физическое проектирование.

Во время концептуального проектирования окончательно формируется замысел будущей базы данных, но без учета любых физических аспектов ее реализации. На этой ступени проектирования разработчика пока не интересует ни конкретная СУБД, на которой позднее развер-

нется БД, ни используемый для создания приложений язык программирования, ни особенности аппаратной платформы. Пока наш основной интерес направлен на создание общей модели, отражающей представления будущих пользователей БД об автоматизируемом участке компании (складе, бухгалтерии, отделе кадров, производственных цехах и т. п.). Вся необходимая для этого информация уже должна быть собрана на предыдущем этапе жизненного цикла БД.

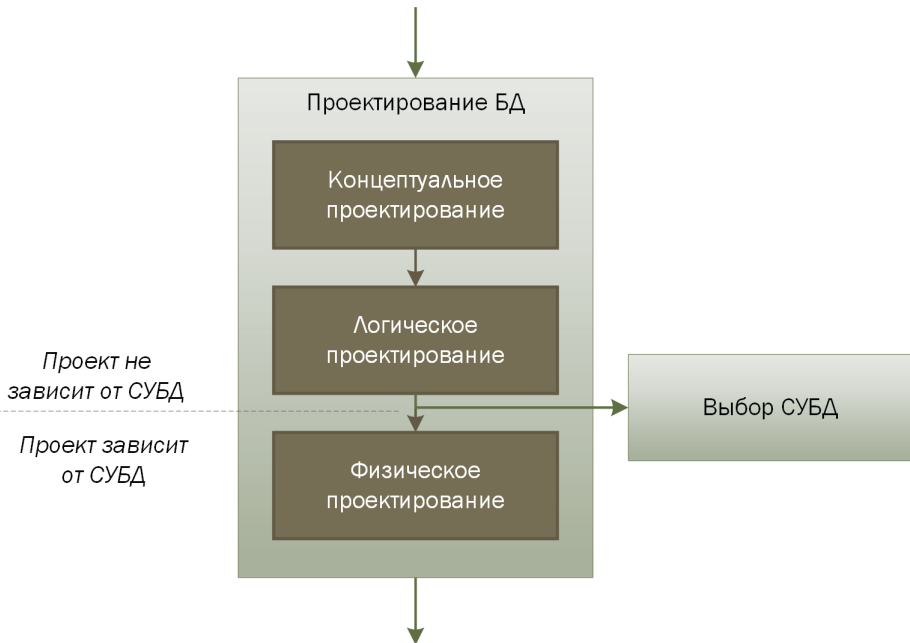


Рис. 5.3. Фазы проектирования БД

Основным средством построения концептуальной модели БД выступает модель «сущность-связь» (см. главу 6) или родственные ей модели. В основе таких моделей лежит простая и одновременно эффективная методика, позволяющая наглядно представлять смысл подлежащих хранению в БД данных.

На завершающей стадии концептуального проектирования разработчик БД обязательно должен проверить адекватность полученной модели, для этого он обсуждает полученные результаты (элементы ER-диаграмм) с сотрудниками компании. При обнаружении несоответствий в модель немедленно вносятся исправления. Процесс сверки прекращается только после того, когда все пользователи подтвердят корректность концептуальной модели.

Таким образом, результатом фазы концептуального проектирования станет ER-модель будущей БД, включающая в себя описание:

- типов сущностей;
- связей между типами сущностей;
- атрибутов (желательно с предварительным описанием доменов и ограничений);
- первичных ключей.

Фаза логического проектирования предназначена для преобразования обобщенной концептуальной модели в завершенную логическую. Девизом логической фазы может стать фраза: «Анализ и рационализация». Разработчик уточняет все требования, выявленные на концептуальной стадии проектирования, и стремится несколько упростить решение (при этом не снижая его функциональные возможности). Для этого предварительная ER-модель проверяется с помощью правил нормализации (см. главу 7). В результате мы получаем не избыточные реляционные таблицы, свободные от присущих ненормализованным данным аномалий вставки, редактирования и удаления.

Помимо нормализации, на логическом этапе осуществляются следующие действия:

- уточняются ограничения на данные;
- определяются домены данных;
- вводятся бизнес-правила и корпоративные ограничения целостности;
- определяется местоположение будущих таблиц (в случае если речь идет о распределенной БД).

Представленная логическая модель вновь сверяется с будущими пользователями БД и заказчиком проекта. Вносятся последние уточнения и исправления, и все члены проектной группы и заказчик проекта приходят к единому мнению относительно создаваемой БД.

На этапах концептуального и логического проектирования разработчики обычно пользуются одной из двух стратегий проектирования БД: стратегией **восходящего проектирования** (bottom-up design) или стратегией **нисходящего проектирования** (top-down design).

Восходящий подход обычно применяется для сравнительно небольших проектов. Суть метода заключается в том, что проектировщик совместно с заказчиком БД строит полный список атрибутов (столбцов таблиц), подлежащих хранению. Позднее атрибуты группируются в

типы сущностей, которые попадают в модель. Когда мы дойдем до главы 7, посвященной нормализации таблиц, то вы увидите, что процесс нормализации основан на восходящем подходе.

Обратная восходящей, нисходящая стратегия лучше подходит для средних и больших проектов. Здесь проектирование начинается с выявления основных типов сущностей, и только затем сущности «обрастают» атрибутами. Классический пример нисходящего метода – модель сущность-связь (ER-модель).

Так какой из стратегий следует отдавать предпочтение? Прав окажется тот разработчик, который станет комплексно применять и восходящую, и нисходящую методологии (в нашем случае нормализацию таблиц и ER-модель).

Внимание!

По завершении фазы логического проектирования совместно с заказчиком создается окончательная версия технического задания на БД (см. главу 26). В техническом задании закрепляются требования к проекту, утверждается логическая модель БД, описываются необходимые отчеты и т.п. После разработки технического задания между разработчиком БД и заказчиком заключается юридический договор на разработку БД, в котором, помимо всего прочего, окончательно утверждаются стоимость и сроки выполнения проекта.

Переход к следующей (физической) фазе проектирования БД производится после выбора целевой СУБД, именно она определяет особенности будущего программного продукта. О правилах отдания предпочтений той или иной технологии при выборе СУБД мы поговорим страницей позднее, а пока отметим тот факт, что с этого момента все остальные фазы проектирования БД и этапы жизненного цикла БД приобретают зависимость от СУБД.

Замечание

Ряд специалистов полагает, что выбор целевой СУБД корректнее проводить не после логического проектирования, а несколько ранее – после этапа концептуального проектирования. Мы не будем вставать ни на ту, ни на другую сторону. Главное, чтобы разработчик определился с СУБД до начала этапа физической разработки БД.

Физическое проектирование – это уточнение решения с учетом имеющихся в наличии разработчика технологий, возможности реализации и требуемой производительности. Только на заключительной

фазе проектирования БД на смену так нелюбимой программистами бумажной работе приходит реальная работа на компьютере. Во время физического проектирования задачей проектировщика становится перенос логической модели на платформу целевой СУБД. С этой целью разработчик делает следующее:

- создает таблицы и связи между ними;
- назначает вторичные индексы таблиц;
- разрабатывает представления;
- реализует бизнес-логику БД (в первую очередь с помощью триггеров и хранимых процедур);
- определяет функциональные характеристики транзакций;
- внедряет механизмы защиты (как минимум предусматривает авторизацию пользователей и назначает правила доступа к данным).

Полученная БД тщательным образом документируется, в особенности в части, касающейся:

- определения пользовательских типов данных;
- описания таблиц и связей между ними, порядка поддержки бизнес-логики;
- назначения и порядка вызова хранимых процедур и триггеров.

Несмотря на то что на рис. 5.2 фазы проектирования нарисованы в виде последовательных операций, вполне допускается их взаимное перекрытие (параллельное развитие) и итеративность (каждая фаза из-за многократных проверок уточнений осуществляется в несколько циклов).

Этап выбора СУБД

Наша будущая БД должна функционировать под управлением соответствующего сервера, поэтому задача выбора наиболее подходящей СУБД должна быть решена еще на самых ранних этапах проектирования. В любом случае, ответ на этот вопрос надо получить не позднее завершения этапа логического проектирования (рис. 5.2 и 5.3). В идеале сведения о выбранном сервере следует вписать одним из пунктов в техническое задание на разработку.

Зачастую решающее влияние на процесс выбора целевой СУБД оказывает наличие на предприятии уже развернутого программного обеспечения (иногда устаревшего и, соответственно, не поддерживаемо-

го компанией-разработчиком). В подобной ситуации заказчик БД, в первую очередь исходя из соображений экономии денежных средств, настоятельно рекомендует проектировщику сосредоточить свое внимание на имеющейся в наличии системе. Отрицательные стороны такого подхода к процессу совершенствования корпоративной информационной системы в комментариях не нуждаются. От того, сможете ли вы переломить ситуацию, зависит многое, в том числе безопасность и надежность работы будущей БД.

Другой стороной медали выступают предпочтения программистов, которые уже достаточно давно работают с определенной СУБД и обладают необходимым опытом и профессионализмом. Если эксплуатируемое вами программное обеспечение современно, надежно, безопасно, экономически выгодно и т. д., то в таком случае докажите заказчику, что именно за этим ПО будущее его компании.

Если же все происходит впервые и оплачивающее разработку БД правление компании дает заказчику полный карт-бланш (но при этом не забудет потребовать от исполнителя доказательств оптимальности принятого решения), то при выборе СУБД следует руководствоваться перечнем из 9 критериев, в свою очередь, каждый из критериев включает в себя несколько вложенных показателей (рис. 5.4).

Подчеркнем, что перечень представленных на рис. 5.4 критериев можно (и нужно) развивать и дополнять, учитывая текущее состояние информационных технологий и характер решаемых разработчиком БД задач. Кроме того, в литературе по разработке БД [19, 25, 38] читатель найдет дополнительные параметры и показатели, рекомендуемые специалистами при выборе СУБД.

Результаты анализа программных продуктов различных производителей выносятся на обсуждение с заказчиком базы данных, и только затем принимается решение о выборе целевой СУБД.

Этап создания клиентского программного обеспечения

Завершив работу над формированием физической структуры БД, проектировщик переходит к этапу разработки приложений, предназначенных для работы с БД. Пользователь не обязан вникать в особенности реляционных БД, ему не стоит учить язык запросов SQL, пользователь просто хочет обладать необходимым инструментом, благодаря которому он без особых затруднений сможет воспользоваться услугами БД. Поэтому, с одной стороны, задачей разрабатываемого нами прикладного ПО является предоставление интуитивно понятного интерфейса

конечному пользователю, а с другой – обеспечение прозрачного для пользователя взаимодействия с БД.

Возможности по определению данных	• Возможность определения доменов • Простота реструктуризации БД • Стандартные и расширенные типы данных • Механизм представлений • Производительные индексы
Требования к развертыванию	• Аппаратные требования • Требования к операционной системе и другому ПО • Требования к сетевой инфраструктуре
Защита данных	• Авторизация • Аутентификация • Шифрование
Доступность	• Степень поддержки стандарта SQL • Многопользовательский доступ • Интеграция с ПО других производителей
Особенности управления транзакциями	• Число одновременных подключений • Модель обработки транзакций • Поддержка точек восстановления • Особенности ведения системного журнала
Современные средства разработки	• Удобный интерфейс разработчика • CASE-инструменты • Хранимые процедуры, триггеры и т.п. • Наличие средств отладки
Инструментарий администратора	• Резервное копирование и восстановление • Контроль производительности • Ведение протокола работы СУБД • Импорт/экспорт данных
Поддержка производителем	• Возможность модернизации • Регулярный выпуск исправлений • Полнота документации
Экономические параметры	• Стоимость приобретения • Стоимость эксплуатации

Рис. 5.4. Критерии для оценки СУБД

В самом общем случае приложение баз данных должно уметь делать следующее:

- получать доступ к БД;
- читать, добавлять, редактировать и удалять данные;
- представлять полученные данные в требуемом пользователем виде (формы, отчеты, многомерное представление и т. д.);
- поддерживать целостность данных, определять дополнительную бизнес-логику и ограничения на данные;
- обеспечивать требуемую безопасность данных.

В качестве инструментальных средств, применяемых для написания прикладного ПО, сегодня используются языки программирования 4-го поколения. Программам попроще обычно хватает возможностей генераторов форм, имеющихся в распоряжении настольных СУБД класса Microsoft Access. Профессиональные клиент-серверные проекты преимущественно создаются на платформах профессиональных сред программирования, например в Visual Studio компании Microsoft или Embarcadero RAD Studio. Кроме того, сегодня очень популярны инструменты, создающие веб-клиенты, не требующие развертывания какого-то специального ПО и работающие под управлением практически любого интернет-браузера.

Создание пользовательских приложений – сложный и кропотливый процесс, требующий от программиста глубоких знаний не только СУБД и языка программирования, но и операционной системы и технологий и методов программирования.

Как правило, создание коммерческого ПО предваряется выпуском прототипов будущих приложений. Прототип представляет собой работоспособную модель приложения, но с несколько ограниченным функционалом. Основное предназначение прототипа – предоставить будущим пользователям возможность опробовать приложение в действии и выяснить, в каких улучшениях они заинтересованы.

Этап тестирования и отладки

Без тестирования мы с вами не сможем гарантировать заказчику правильную работу созданного продукта. Поэтому, вне зависимости от степени сложности БД, для каждого проекта необходимо разработать исчерпывающий план тестирования, распространяющийся на все основные функции БД и прикладного ПО [11, 13, 21, 43].

Определение

Тестирование – это процесс выполнения программы с целью обнаружения ошибок.

Тестирование обеспечивает:

- обнаружение ошибок;
- демонстрацию соответствия функций БД и клиентского ПО их назначению;
- демонстрацию реализации требований к характеристикам БД и клиентского ПО.

Косвенно тестирование осуществляет проверку ряда показателей надежности (один из важнейших критериев, характеризующих качества ПО).

Различают два принципа тестирования ПО:

- 1) структурное тестирование;
- 2) функциональное тестирование.

Структурное тестирование (еще его называют тестированием «белого ящика», т. к. нам полностью доступен исходный код программы) основано на анализе управляющей структуры программы. Лицо, ответственное за проведение тестирования, строит граф управления программой и формирует тестовые варианты для всех возможных маршрутов.

Во время структурного тестирования проводится:

- 1) модульное тестирование. Тестированию подлежат минимальные компоненты ПО, например запрос, процедура, триггер;
- 2) интеграционное тестирование. При интеграционном тестировании модули ПО объединяются и осуществляется проверка на корректность взаимодействия между интегрируемыми компонентами.

Функциональному тестированию подвергается уже полностью откомпилированное программное изделие. Как вы понимаете, на этом этапе исходный код вторичен, поэтому процесс называют тестированием «черного ящика». По своей сути функциональное тестирование ПО представляет собой эксплуатацию приложения в контролируемых условиях с последующим анализом полученных результатов. При этом проверяется работа приложения не только с нормальными, но и с ошибочными данными. Также во время тестирования следует изучить поведение БД и ПО в нештатных ситуациях.

Функциональное тестирование обычно включает в себя два этапа: альфа-тестирование и бета-тестирование. Во время альфа-тестирования имитируется реальная работа БД и прикладного ПО, но в качестве пользователей выступают штатные разработчики проекта. На этапе бета-тестирования программное обеспечение передается заказчику (или третьему лицу) с целью апробации работы и выявления не замеченных ранее ошибок в его работе.

Все выявленные во время тестирования ошибки должны быть локализованы и исправлены, для этого осуществляется отладка ПО.

Определение

Отладка – это процесс локализации и исправления ошибок, обнаруженных при тестировании программного обеспечения.

Различают три вида ошибок.

1. Синтаксические ошибки – ошибки, выявляемые при выполнении синтаксического и частично семантического анализа программы. Как правило, с такими ошибками не сложно бороться, так как они автоматически фиксируются компилятором (транслятором, интерпретатором).
2. Ошибки компоновки – ошибки, обнаруженные компоновщиком при объединении модулей программы. Подобная категория ошибок также относительно легко выявляется и устраняется.
3. Ошибки выполнения – самая сложная для устранения категория ошибок. Ошибки выполнения обнаруживаются операционной системой, аппаратными средствами, лицами, осуществляющими тестирование, или просто конечным пользователем при работе с программой.

Отладка – это едва ли не самый сложный компонент работы программиста. Помимо того что отладка требует глубоких знаний и умений, она еще и психологически дискомфортна, ведь приходится сражаться с собственными ошибками, зачастую находясь в условиях цейтнота.

Этап реализации

На этапе реализации осуществляется выпуск окончательной версии программного продукта (см. главу 24). Прежде чем проектировщик переходит к выпуску БД, необходимо убедиться в выполнении следующих условий:

- база данных и пользовательские приложения обладают всеми указанными в техническом задании функциональными возможностями;
- программное обеспечение успешно прошло тестирование, и все выявленные критические ошибки устранены;
- заказчик и проектировщик приняли совместное решение о том, что реализация всех элементов БД в целом завершена.

На завершающей стадии этапа реализации разработчик ПО готовит материалы, необходимые для обучения и технической поддержки пользователей и для сопровождения приложения.

Этап эксплуатации и сопровождения

Эксплуатация и сопровождение – это заключительный этап жизненного цикла БД. В его начальной стадии осуществляется развертывание базы данных и прикладного ПО на сервере и клиентских станциях заказчика ПО. При необходимости администратор СУБД создает учетные записи пользователей базы данных. После этого проект полностью переходит под контроль заказчика.

Внимание!

Развертывание БД и пользовательского ПО возможно только после выполнения всех тестов, доказывающих их стабильность.

Основная часть обязанностей по сопровождению БД возлагается на администратора БД. Администратор осуществляет профилактическое обслуживание БД, восстанавливает систему после сбоев, управляет учетными записями пользователей, следит за производительностью системы, поддерживает безопасность БД, ведет системный аудит. Одним словом, администратор поддерживает БД в работоспособном состоянии (см. главу 3).

Хотя эксплуатация и сопровождение отнесены к заключительному этапу жизненного цикла, это не означает, что уже введенный в эксплуатацию проект не подлежит совершенствованию. У заказчика могут возникнуть дополнительные пожелания по развитию проекта – введению новых форм ввода и просмотра данных, запросов, отчетностей. Если концептуальная и логическая модели достаточно глубоко проработаны, то БД даже будет допускать изменения в структуре и в бизнес-логике.

Но однажды наступит тот день, когда приспособить БД к новым реалиям окажется невозможно. Что в этом случае следует предпринять?

Направление дальнейшего движения подскажет стрелка, исходящая из этапа эксплуатации и сопровождения (см. рис. 5.2), мы вновь оказываемся на первой ступеньке лестницы жизненного цикла БД и начинаем планировать разработку новой БД.

Резюме

Процесс проектирования и реализации жизнеспособной БД содержит серию этапов, включающих планирование разработки, определение и анализ требований к системе, проектирование БД, выбор целевой СУБД, создание прикладного ПО, тестирование, реализацию, эксплуатацию и сопровождение.

Перечисленные этапы обладают итеративными чертами и предусматривают возможность возврата к предыдущему или даже к начальному этапу проектирования в случае невозможности реализовать пожелания заказчика к проекту в заданном объеме и с требуемым качеством.

Вопросы для самопроверки

1. Что такое технология программирования?
2. Что такое информационная система предприятия?
3. Укажите место БД в цикле преобразования информации.
4. По каким признакам можно судить о необходимости создания (модернизации) БД на предприятии?
5. Какие этапы входят в жизненный цикл БД?
6. На какие вопросы следует получить ответы на этапе планирования разработки БД?
7. Что нужно знать о предприятии, приступая к этапу определения и анализа требований к системе?
8. Каким образом следует осуществлять сбор требований пользователей к проектируемой БД?
9. Какие фазы проектирования БД вам известны?
10. В чем отличие между стратегиями восходящего и нисходящего проектирования?
11. Какие критерии следует учитывать на этапе выбора СУБД?
12. Какие виды тестирования ПО необходимо провести перед вводом БД в эксплуатацию?
13. С какими категориями ошибок приходится сталкиваться во время отладки?
14. Почему эксплуатацию и сопровождение нельзя считать заключительным этапом жизненного цикла БД?

Глава 6

Концептуальное проектирование и ER-модель

Не секрет, что процесс разработки программного обеспечения сопряжен со многими сложностями, это и поиск эффективных алгоритмов, подбор подходящих структур данных, отладка и тестирование некорректного кода, дизайн удобного интерфейса приложения. Но каждая из перечисленных трудностей – ничто в сравнении с проблемой поиска общего языка между заказчиком и разработчиком программного продукта. Это подтвердит любой программист, написавший хотя бы одну программу по просьбе третьих лиц. Проблема многогранна, но в ее базе лежит всего-навсего одно противоречие – программист и заказчик разговаривают на разных языках. Бухгалтер, экономист, аптечный работник, строитель и любой другой узкий специалист прекрасно понимает, чего он хочет от будущей БД, но объяснить это программисту не в состоянии. Их, как пропасть, разделяет языковой барьер. Радиоинженер, мечтающий заполучить программу, макетирующую сверхновый процессор, сталкивается с разработчиком программного обеспечения, в совершенстве владеющим десятком языков программирования, но не имеющим ни малейшего представления не только о законах Кирхгофа, но даже не способным по внешнему виду отличить транзистор от диода. У медали имеется и обратная сторона: разработчик базы данных, стремясь проверить адекватность построенной им модели, обращается за помощью к заказчику программного продукта. Программист рассчитывает, что последний сможет оценить степень соответствия проектируемой им информационной системы представлению об этой системе заказчика. Заказчик и рад это сделать, но он ничего не понимает в БД и в современных языках программирования...

Именно поэтому процесс моделирования будущей базы данных начинается с этапа концептуального проектирования, во время которого заказчику и разработчику БД проще всего найти общий язык.

Концептуальная модель БД

Концептуальное проектирование базы данных представляет собой процесс создания модели, используемой на предприятии информации без учета каких-либо физических аспектов ее реализации.

На этапе концептуального проектирования разработчик будущей БД должен обладать полной картиной об автоматизируемом предприятии, однако он пока не ограничен в вопросах выбора архитектурных решений, целевой СУБД, языков программирования и т. п. Одним словом, создатель БД пока может не задумываться о физическом проектировании.

И это очень серьезное подспорье для программистов, ведь, к сожалению, реляционная модель неудобна в период проектирования будущей БД не только с точки зрения заказчика, но и даже с позиций профессионального разработчика ПО, отлично знающего детище Эдгара Кодда. Модель не в состоянии наглядно представлять смысл хранимых данных. Попробуйте на досуге смоделировать и изобразить на листе бумаги производственный процесс химического предприятия в виде плоских двумерных таблиц... Также хотя физически реляционная модель и превосходно поддерживает взаимосвязи между таблицами, но с точки зрения наглядной для разработчика формализации этих зависимостей средств у нее практически нет.

ER-модель

Проблема упрощения процесса проектирования базы данных, так чтобы, с одной стороны, модель была понятна и неспециалисту, а с другой – вполне устраивала и подготовленного программиста, не нова. Работа над этой проблемой вызвала к жизни целое семейство семантических моделей данных. На сегодняшний момент времени существует несколько эффективных решений, мы познакомимся с ключевым из них – высокоуровневой концептуальной моделью, разработанной Питером Ченом (Peter Pin-Shan Chen). Указанная модель известна под названием «сущность-связь» (Entity-Relationship), или просто ER-модель. Впервые она была анонсирована в марте 1976-го, тогда П. Чен опубликовал работу «The Entity-Relationship Model. Toward a Unified View of Data» в научном журнале «Transactions on Database Systems (TODS)».

Замечание

Модель «сущность-связь» относится к разряду концептуальных. Другими словами, она представляет общий взгляд на данные и позволяет нам на понятийном уровне разобраться с тем, что будет представлено в будущей базе данных. Ответ на вопрос, как данные должны быть реализованы на практике, надо искать у логической и физической моделей.

Достоинство предложенного П. Ченом решения – в том, что ER-модель представляется в виде наглядных графических диаграмм. Для того чтобы научиться читать и составлять эти диаграммы, не требуется глубокой специальной подготовки. Терминология ER-модели опирается на хорошо знакомые нам (см. главу 4) понятия «тип сущности», «атрибут» и «связь».

Типы сущностей и атрибуты

Построение ER-модели начинается с выявления всех типов сущностей, подлежащих хранению в будущей базе данных. Различают две разновидности типов сущностей: слабую и сильную. Слабый тип находится в зависимом состоянии от сильной сущности, напротив, сильный тип вполне самостоятелен и ни от кого (или чего) не зависит. На диаграммах сильный тип изображается в виде прямоугольника с названием типа сущности внутри него, для обозначения слабого типа сущности контур прямоугольника рисуется двойной линией (рис. 6.1).

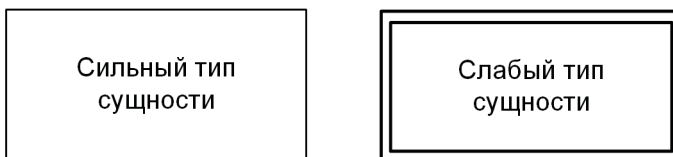


Рис. 6.1. Обозначение слабого и сильного типов сущности

Каждый тип сущности обладает некоторым набором атрибутов, в которых хранятся значения, описывающие конкретную сущность. Различают простые, составные, однозначные, многозначные и производные атрибуты. Главное отличие между простым и составным атрибутами в том, что простой состоит из одного компонента, а составной из нескольких. Примером составного атрибута может стать почтовый адрес, он включает группу простых атрибутов (индекс, страна, город, улица, дом). Однозначный атрибут содержит одно-единственное значение для сущности, например фамилию для типа сущности «Сотрудник». Мно-

гозначный атрибут допускает одновременное хранение нескольких значений, например у сотрудника может быть несколько телефонных номеров. Производный атрибут содержит значение, полученное на основе данных, хранящихся в других атрибутах. Например, возраст сотрудника можно легко вычислить, зная день его рождения и сегодняшнее число. Графическое представление атрибутов на ER-диаграмме вы найдете на рис. 6.2.

Замечание

В реляционных таблицах производные атрибуты редко когда превращаются в реальные столбцы таблицы, физически хранящие значения. Ведь нет смысла занимать драгоценную память под данные, которые можно получить путем элементарных вычислений.



Рис. 6.2. Представление атрибутов на диаграммах ER-модели

Для изображения на схеме атрибута применяется эллипс, к своему типу сущности он присоединяется линией. Название атрибута записывается внутри эллипса. Если атрибут содержит идентификатор сущности (позднее он превратится в первичный ключ отношения), то название атрибута подчеркивается. Производный атрибут обводится пунктирной линией, многозначный – двойной.

Замечание

При рисовании многозначного атрибута некоторые разработчики вместо черчения эллипса с двойным контуром рисуют обычный эллипс, но соединяют его с сущно-

стью двойной линией. То же самое можно сказать и о производном атрибуте, только на этот раз эллипс соединяют с прямоугольником пунктирной линией.

Рассмотрим представленный на рис. 6.3 фрагмент ER-модели. Это диаграмма «Employee», описывающая тип сущности «сотрудник предприятия». Как видите, на диаграмме мы сумели представить все имеющиеся в нашем распоряжении разновидности атрибутов. Атрибут «Employee_id» выполняет функции первичного ключа сущности. Имя «FName» и фамилия «LName» являются ингредиентами составного атрибута «EmployeeName». Кроме этого, у нас имеется еще один составной атрибут «Address», он хранит почтовый адрес служащего. Обведенный двойной линией многозначный атрибут «PhoneNum» говорит нам о том, что у реального человека может быть несколько контактных телефонных номеров. Производный атрибут «Age» отражает возраст сотрудника, на схеме этот атрибут соединен с датой рождения «Birthday». Такая подробность подскажет разработчику, что расчет возраста должен вестись от даты появления на свет работника предприятия. Согласитесь, что для понимания сути предложенной диаграммы не требуется специальной подготовки, поэтому, нарисовав подобную модель будущей таблицы, можно без сомнений идти к заказчику базы данных.

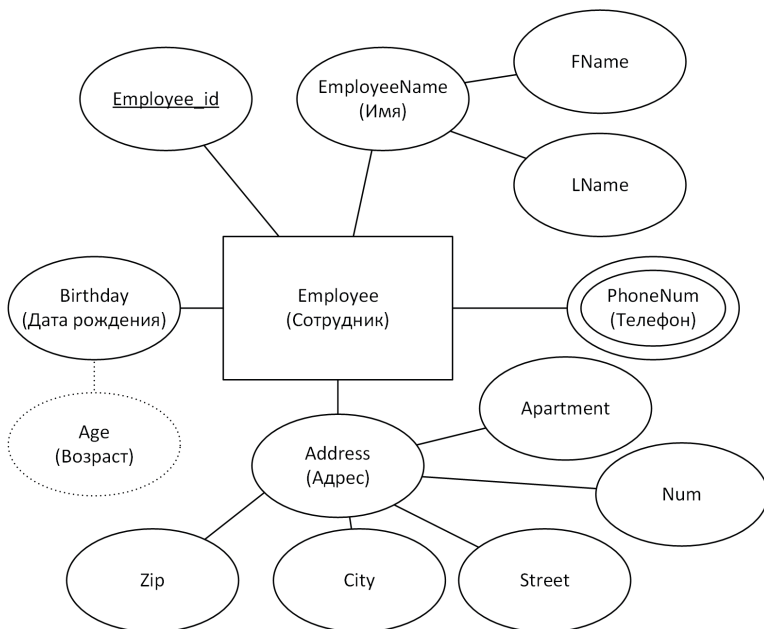


Рис. 6.3. Представление типа сущности на диаграммах ER-модели

Подтипы сущностей

Чем больше в организации числится персонала, тем сложнее организовать учет данных о нем. Люди обладают разными специальностями и разными профессиональными навыками. Если фирма располагает научно-исследовательскими лабораториями, то нам потребуется знать ученые степени и звания сотрудников. Если в этой же фирме ведется производственная деятельность, то стоит учитывать квалификационные разряды металлургов, фрезеровщиков и токарей. Если у нас имеется служба безопасности, то, возможно, следует обладать информацией о спортивных достижениях сотрудников охраны. Список можно продолжать до бесконечности; чем шире спектр интересов предприятия, тем больше атрибутов потребуется для описания его сотрудников. Как следствие у всех лиц, задействованных в нашей фирме, кроме общих данных (фамилия и имя, дата рождения и т. п.), появляются уникальные черты, характерные только для узких профессиональных групп. Каким образом организовать хранение данных в таких случаях?

Лобовым решением подобных задач может стать попытка определить один-единственный универсальный тип сущности (например, как на рис. 6.3) и, пытаясь учесть все возможные случаи жизни, снабдить его всем возможным спектром атрибутов. Указанное решение вполне работоспособно, но абсолютно нерационально. В результате, принимая на работу обычную уборщицу, сотрудник отдела кадров будет вынужден указывать, что она не является доктором наук, не обладает черным поясом по карате и не имеет опыта пилотирования на вертолете в ночное время... Как следствие наш программный продукт окажется весьма неудобным в эксплуатации и займет призовое место во все расширяющейся плеяде плохо спроектированных БД.

Вполне жизнеспособен еще один способ выхода из сложившегося положения. Вместо вооружения типа сущности огромным набором атрибутов, специализирующихся на хранении узконаправленных сведений, мы предпринимаем попытку создать сравнительно небольшой набор разнотипных столбцов универсального назначения и вводим дополнительный атрибут флагов. Вся остальная бизнес-логика БД определяется состоянием флагов: если установлено значение 0 – то в универсальном столбце номер N хранится номер водительского удостоверения, если 1 – в столбце N мы найдем номер диплома кандидата наук, если 2 – тайный счет сотрудника в банке в Швейцарии, и т. д. С точки зрения экономии памяти рассматриваемый подход вполне приемлем, ведь мы избавились от целой вереницы полей, большей частью времени хранящих не-

определенность NULL. Но зато головную боль приобретают прикладные программисты (они пишут клиентские приложения) и в наилучшем положении оказывается администратор базы данных, несущий ответственность за физическую реализацию БД. У администратора начинаются проблемы с поддержанием непротиворечивости данных, у программистов – со способами построения форм ввода, запросов, отчетов. Ко всему прочему подлить масла в огонь смогут непредвиденные обстоятельства. Например, в один прекрасный день к вам обратится начальник отдела кадров и спросит, как ему поступить в ситуации, когда бывший пилот самолета генерального директора переходит на почетную должность директора сауны? Ведь в столбце *N* пилота хранилось время полета в часах, а у директора сауны – наработка котла в котельной...

Наконец, мы пришли к третьему варианту действий разработчика будущей БД. Для снижения остроты проблемы огромного количества атрибутов стоит ввести в ER-модель подтипы сущности. Это тот случай, когда общая для всех сотрудников информация хранится в головном типе сущности (супертипе), а нюансы и тонкости выносятся в несколько специализированных подтипов.

В нашем примере количество подтипов определяется числом профессиональных групп сотрудников предприятия (рис. 6.4). Все атрибуты супертипа в равной степени принадлежат всем подтипам, ведь у любого сотрудника есть имя и дата рождения. А информация, хранимая в подтипах сущности, специфична для каждой из групп. Обратите внимание на то, что подтипы «Manager» и «Scientific» помечены кружочком с символами «Gs», а подтип «Worker» отмечен символом «G». Символы «Gs» указывают на то, что подтипы относятся к разряду пересекающихся, т. е. ученый одновременно может быть и управленцем нашего предприятия. В таком случае мы допускаем, что данные об одном и том же человеке могут одновременно находиться в таблицах, учитывающих менеджеров и ученых. Символ «G» отмечает непересекающийся тип – рабочий не может одновременно являться ученым или менеджером.

Замечание

Супертип и подтипы состоят друг с другом в отношениях «один к одному». Это едва ли не единственный случай в реляционных БД, когда появление связи 1:1 значительно упрощает процесс проектирования и сопровождения БД.

Вне всякого сомнения, появление нескольких подтипов сущности усложняет работу над проектом. В процессе проектирования выявляется великое множество проблем и проблемок:

- не существует единого правила поддержания целостности данных при организации взаимодействия между головной и подчиненными таблицами;
- возникают определенные сложности в определении первичных ключей в таблицах подтипов;
- средствами современного SQL весьма сложно создать запрос, объединяющий всю информацию из таблицы-супертипа и всех таблиц подтипов;
- достаточно тяжело обеспечить безопасный доступ пользователя к столбцам таблиц подтипов.

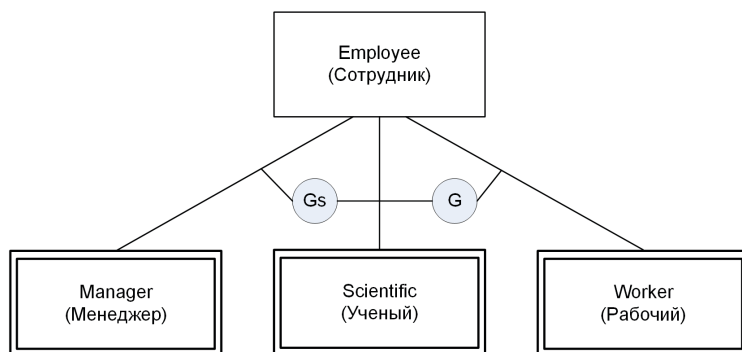


Рис. 6.4. Изображение супертипа и подтипов на диаграмме

Вместе с тем построенные на основе идеи подтипов сущностей базы данных приобретают неоспоримые преимущества:

- устраняется избыточность данных, так как каждый подтип хранит только необходимую информацию;
- упрощается процесс определения доменных ограничений для столбцов таблиц подтипов;
- появление новой (не учтенной в момент проектирования БД) классификационной группы записей приводит к созданию очередной таблицы подтипов. Поэтому не возникает нужды реструктурировать таблицу супертипа (добавление новых или изменение старых столбцов), с вытекающими отсюда проблемами кардинальной переработки всех клиентских приложений.

Связи в ER-модели

Сформировав полный перечень всех подлежащих учету типов сущностей и их атрибутов, создатель ER-модели переходит к очередному

ответственному этапу. Теперь ему предстоит выявить все ассоциации между типами сущностей и на этой основе построить связи между ними. В предыдущей главе уже упоминалось, что явным признаком связи выступает глагол, который можно применить, характеризуя взаимоотношения между типами сущностей. Например, сотрудник **работает** в отделе, или самолет **выполняет** рейс.

Для того чтобы разработчик мог на схеме отразить смысловую нагрузку связи между сущностями, ее имя, а точнее глагол, показывающий характер взаимодействия между сущностями, записывается внутри ромба. С целью упрощения диаграммы допускается опускать атрибуты типов сущностей, ограничиваясь только первичными ключами (рис. 6.5).

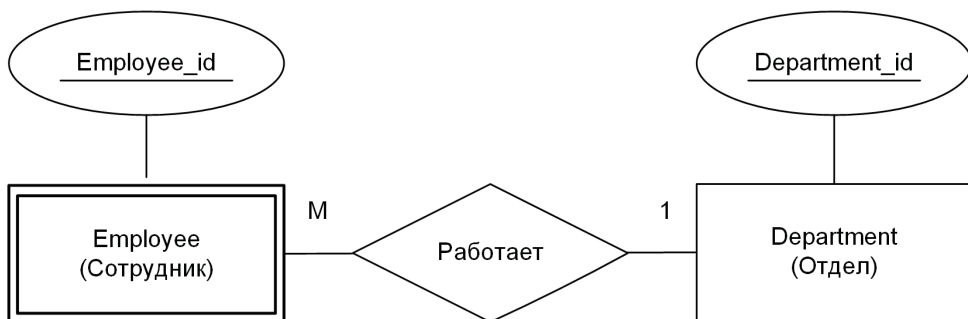


Рис. 6.5. Представление связи 1:M на диаграммах ER-модели

В ER-моделировании различают три типа связи между типами сущностей: «один к одному» (1:1), «один ко многим» (1:M) и «многие ко многим» (M:N). При появлении на диаграмме связи типа 1:1 следует задуматься, не является ли предполагаемый тип сущности всего лишь атрибутом. Исключение составляет обсужденный страницей ранее механизм супертипов и подтипов.

На рис. 6.5 представлена наиболее часто встречающаяся связь «один ко многим» — в одном отделе работает много сотрудников. На диаграмме отдел «Department» представлен как сильная сущность, сотрудник «Employee» — как слабая. Такое решение объясняется тем, что сотрудник находится в подчиненном отношении к отделу, в котором он трудится. Часто программисты для повышения информативности диаграмм при обозначении связи вместо использования символа «M» явным образом указывают мощность связи. Например, обозначение (1, 10) говорит о том, что минимальное значение мощности соответствует 1, а максимальное — 10 (другими словами, в отделах может работать от 1 до 10

сотрудников). Такие пояснения могут оказаться весьма полезными при практической разработке программного обеспечения, так как более подробно отражают бизнес-правила автоматизируемого учреждения или предприятия.

Проектируя ER-модель, нам следует особое внимание уделять связи «многие ко многим». Вполне реальна ситуация, когда несколько сотрудников одновременно выполняют несколько производственных поручений в заказе «Order» (рис. 6.6). Например, один и тот же автомеханик в рабочий день может получить заявки на обслуживание нескольких автомобилей, при этом отдельно взятый автомобиль может попасть в руки нескольких специалистов (допустим, на ремонт двигателя и на замену электропроводки).

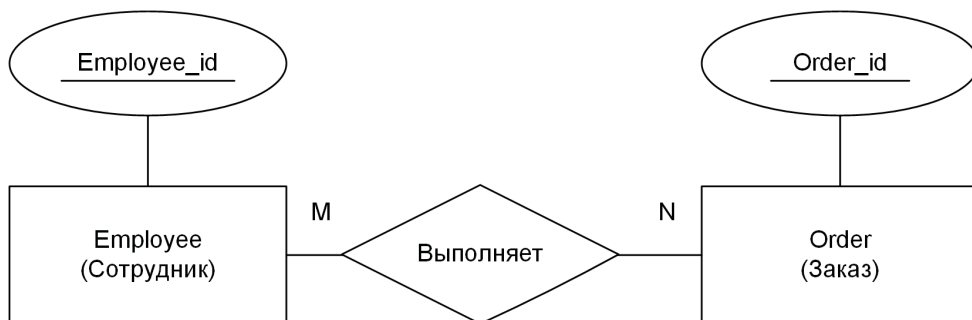


Рис. 6.6. Представление связи M:N на диаграммах ER-модели

Хотя связь M:N легко изобразить на диаграмме, ее далеко не просто реализовать физически, ведь реляционные базы поддерживают только отношение «один ко многим». Безусловно, рассматриваемая ситуация не безвыходная. Позднее, обсуждая процесс нормализации базы данных, мы очень подробно рассмотрим способ решения подобных задач на физическом уровне. А сейчас, пока мы находимся на концептуальном уровне моделирования, лишь напомним, что в тех случаях, когда между двумя типами сущностей возникает связь «многие ко многим», разработчик БД создает искусственный тип сущности, выполняющий функции коммутатора между основными сущностями (рис. 6.7).

Дополнительный тип сущности разрывает связь M:N пополам, что позволяет нам трансформировать неподъемное для реляционных баз данных отношение «многие ко многим» в пару обычных связей «один ко многим». Вне зависимости от всех хитросплетений нашей ER-модели искусственно созданная сущность-коммутатор в любом случае будет содержать два атрибута для внешних ключей. Они предназначены для

поддержания ассоциации между объединяемыми отношением «многие ко многим» типами сущностей, один атрибут коммутатора станет держать связь с расположенным на диаграмме слева типом «Employee», а другой – с правым типом сущности «Order». На диаграммах совсем необязательно опускаться до степени детализации рис. 6.7. Это может оказаться излишним с точки зрения наглядности схемы, поэтому большинство разработчиков ER-модели предпочитает компромиссное решение, оно представлено на рис. 6.8. Суть компромисса в том, что на диаграммах искусственный тип сущности изображается в виде ромба, вписанного в прямоугольник. Такое графическое решение, с одной стороны, указывает на то, что это будущая таблица, а с другой – напоминает, что своим появлением на свет таблица обязана необходимости организовать коммутацию между другими таблицами.

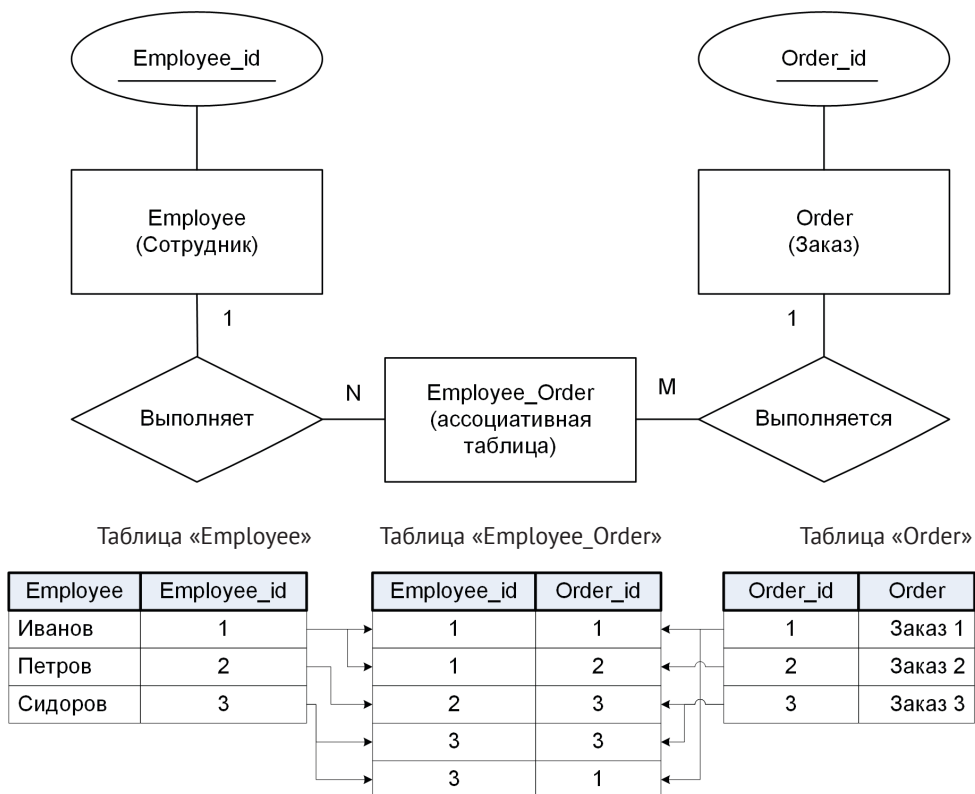


Рис. 6.7. Преобразование отношения M:N к двум отношениям 1:M

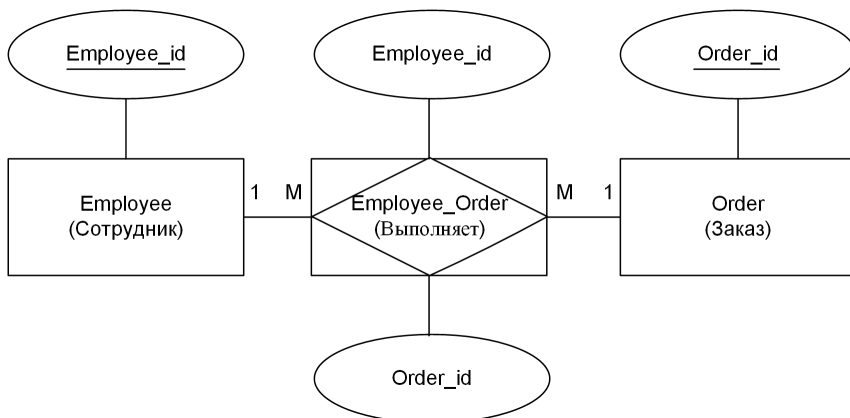


Рис. 6.8. Коммүтирующий тип сущности на ER-диаграмме

Сильные и слабые связи

Рассмотрим еще одну особенность взаимодействия между типами сущностей в базе данных – наличие сильных и слабых связей. Сильная связь обычно возникает между сильным и слабым типами сущностей в тех условиях, когда существование слабого типа невозможно без поддержки сильного. Например, слабая сущность «самолет» принадлежит сильной сущности «авиакомпания», или слабая сущность сотрудник не может трудиться на предприятии, не входя в штат какого-либо из отделов. Напротив, в той ситуации, когда присутствие связи между двумя типами сущности не обязательно, мы говорим о слабой связи. Слабые связи надо искать между типами сущностей, не находящимися в прямой зависимости друг от друга и, как следствие, способными жить автономно. Например, работа над некоторыми заказами может осуществляться не только в стенах нашего предприятия, но и совместно с каким-то соисполнителем (смежной фирмой). Несмотря на то что тип сущности «Ассомплисе» (соисполнитель) относится к разряду сильных, так как может существовать самостоятельно, связь между заказом и смежником оказывается слабой. Ведь нам не всегда нужны помощники, и большинство заказов мы выполняем только своими силами. Об этом нас информирует маленький кружок со стороны необязательной сущности (рис. 6.9).

В последующем, на этапе создания реляционной таблицы «Order», кружок подскажет нам о том, что информация о соисполнителе заказа необязательна и поэтому поле внешнего ключа допускает вставку определителя NULL.

Замечание

В некоторых нотациях ER-модели для обозначения сильной связи на диаграммах очерчивают ромб двойной линией.

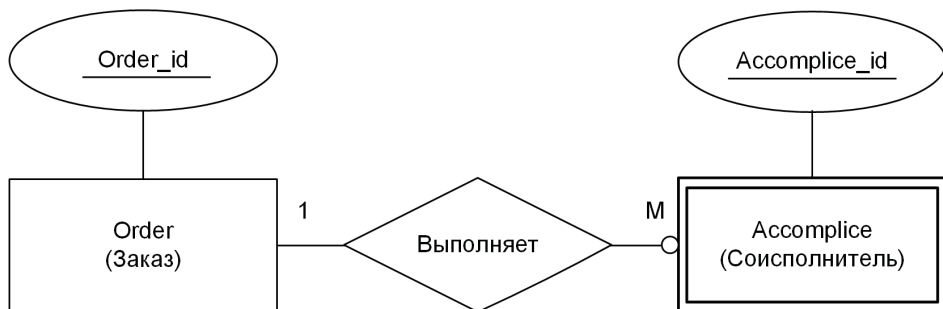


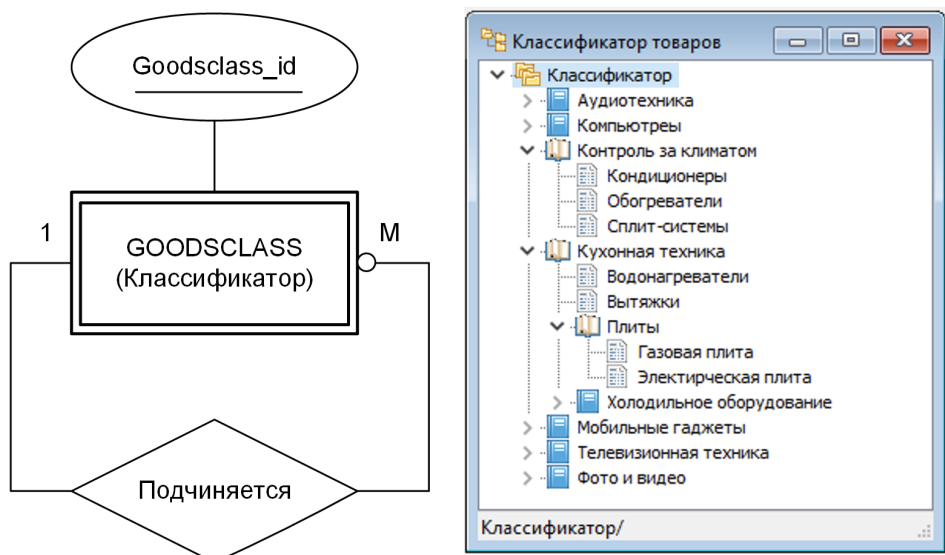
Рис. 6.9. Пример необязательной связи

Рекурсивная связь

До сих пор на наших диаграммах мы сталкивались с наиболее часто возникающими в реальной жизни бинарными связями. Теперь поговорим об унарных связях, которые весьма эффективны при организации рекурсивных отношений внутри одной таблицы. Замыкание типа сущности на самого себя окажется весьма полезным при описании многоуровневых иерархических структур, подобных классификатору товаров, хранящихся на складе. Взгляните на рис. 6.10, на котором представлено решение такой задачи. Здесь тип сущности «Goodsclass» содержит названия категорий товаров, которыми торгует автоматизируемая фирма. Между категориями явно просматривается правило взаимодействия главный–подчиненный.

На практике для построения иерархии реальной реляционной таблице потребуется как минимум три столбца: поле первичного ключа «Goodsclass_id», поле внешнего ключа «Parent_id» и содержащее названия категорий текстовое поле «Goodsclass». Рекурсивная связь между записями таблицы обеспечивается за счет взаимодействия внешнего и первичного ключей, с этой целью поле внешнего ключа дочернего элемента хранит значение первичного ключа родительского узла. Если же идет речь о самом старшем элементе иерархии, который никому не подчинен (в нашем примере во главе дерева расположена папка «Классификатор»), то в его внешнем ключе окажется определитель NULL. Подтверждение моих слов вы обнаружите во фрагменте таблицы «Goodsclass», представленной в нижней части рис. 6.10. Например, узлу

«Контроль за климатом» принадлежат дочерние узлы «Кондиционеры», «Обогреватели» и «Сплит-системы». В полях внешнего ключа всех трех подчиненных узлов вы найдете число 4, а это и есть значение первичного ключа родительского узла «Контроль за климатом».



Фрагмент данных таблицы «GOODSCCLASS»

Goodsclass_id	Parent_id	Goodsclass
1	NULL	Классификатор
2	1	Аудиотехника
3	1	Компьютеры
4	1	Контроль за климатом
5	4	Кондиционеры
6	4	Обогреватели
7	4	Сплит-системы

Рис. 6.10. Представление унарной рекурсивной связи на диаграмме ER-модели

Связи высокого порядка

Модель Питера Чена отличается особой доброжелательностью и допускает возникновение связей более высокого порядка, чем бинарные:

тернарных, кватернарных и т. д. В этом есть определенная логика, ведь в реальном мире объединение одной связью, скажем, трех сущностей далеко не редкость. Допустим, что в наше хранилище нефтепродуктов поступает бензин одной и той же марки от разных поставщиков, отсюда бензин расходуется по нашим бензозаправочным станциям, и из этого же хранилища мы отпускаем бензин ряду оптовых покупателей. Предложенная ситуация смоделирована на рис. 6.11.

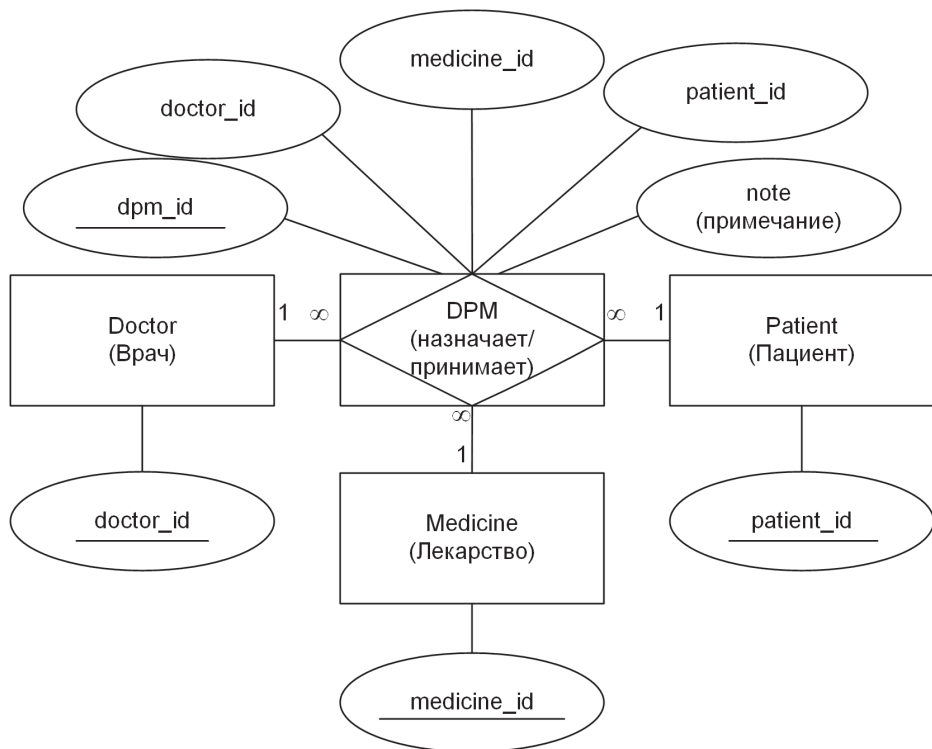


Рис. 6.11. Тернарная связь между типами сущностей

То, что разрешено ER-модели, которая работает на концептуальном уровне проектирования БД, иногда затруднительно реализовать на физическом уровне. Реляционная модель данных без проблем поддерживает унарные и бинарные связи, но при превышении размерности связи разработчик вынужден идти на определенные хитрости. В данном случае рецепт почти такой же, как и в подробно рассмотренной на рис. 6.6 и 6.7 ситуации с поддержанием связи «многие ко многим» между двумя типами сущностей. И тогда, и сейчас мы выйдем из положения за счет введения искусственной сущности-коммутатора. Комму-

татор «S_C_PS» (название создано по первым буквам коммутируемых объектов) объединяет поставщиков, заказчиков и АЗС. Помимо трех внешних ключей, предназначенных для организации связи с интересующими нас сущностями, наша искусственная таблица снабжена первичным ключом и атрибутом «Amount», в котором мы станем хранить объем поставленного/полученного топлива.

Вариации ER-моделей

Модель Питера Чена вполне заслуживает лавры первого простого и одновременно эффективного инструмента моделирования БД. Для ее понимания вовсе не требуется глубоких знаний в области программирования и СУБД. Достаточно способности логически мыслить и немного терпения с крупицей интуиции. Благодаря своей успешности ER-модель Чена не только сразу снискала признательность у обычных программистов, но и вдохновила ряд специалистов на создание схожих по идеологии инструментов проектирования БД.

Один из наиболее популярных нотаций ER-модели принадлежит уже знакомому нам по сетевой модели данных Чарльзу Бэчмэну¹. Бэчман несколько видоизменил графические обозначения связей на диаграммах «сущность-связь». Связи в нотации Бэчмана также чертятся в виде линии, соединяющей типы сущностей, но на ней отсутствует ромб с названием связи. Вертикальная черта на конце линии связи означает, что эта сторона представляет отношение «один». На стороне «многие» линия связи расщепляется на три луча. Некоторая схожесть связи на стороне «М» с трехпалой птичьей лапкой привела к тому, что за ER-моделью Бэчмана прочно закрепилось название «Crow's Foot model» (модель «воронья лапка») (рис. 6.12).



Рис. 6.12. Представление связи на модели «Crow's foot»

¹ Чарльз Бэчман является одним из создателей сетевой модели данных, за выдающийся вклад в развитие баз данных он был удостоен в 1973 г. премии Тьюринга.

На диаграммах «воронья лапка» атрибуты сущности рисуются не в виде отдельных эллипсов, а записываются в виде списка, под названием сущности. Благодаря этому модель приобрела более компактную форму представления.

Еще одна, заслуживающая внимания версия ER-моделей получилась в результате работы американской фирмы Hughes Aircraft. Создатели модели воспользовались наработками в области систем автоматизированного производства (integrated computer-aided manufacturing, ICAM), широко проводимых в США в 1970-х годах, и реализовали свою модель ICAM Definition, сокращенно IDEF. Несколько позднее модель IDEF была доработана и сегодня более широко известна под именем IDEF1X. IDEF1X впитала лучшие стороны моделей Чена и Бэчмэна.

Различными разработчиками создан внушительный перечень программного обеспечения, позволяющего автоматизировать этап концептуального моделирования БД. Эти продукты объединяют под понятием CASE-системы. Термин CASE (Computer-Aided Software Engineering) можно перевести как автоматизированное проектирование и создание программ. В качестве примеров CASE продуктов стоит упомянуть: ER/Studio (корпорации Embarcadero), ERwin Data Modeler (фирмы Platinum – CA), PowerDesigner (фирмы Sybase), Microsoft Visio. На рис. 6.13 представлен экранный снимок процесса построения ER-модели в редакторе MySQL Workbench. Здесь вы обнаружите диаграмму взаимодействия между сущностями БД «Склад», с которой мы еще не раз встретимся на страницах книги. Как видите, корни современных CASE-систем, специализирующихся на проектировании БД, также уходят в модель Питера Чена.

Основная цель CASE-проектирования заключается не столько в автоматизации хода разработки БД, сколько в превращении трудоемкого и малопонятного для непосвященных «ритуала» кодирования в относительно более простой процесс логического проектирования. Разработчику зачастую даже не надо знать языки программирования, достаточно владеть методологией концептуального проектирования БД и уметь пользоваться мышкой... Как следствие CASE-средством сможет воспользоваться даже начинающий пользователь, что не только ускоряет, но и значительно удешевляет стоимость получаемого программного обеспечения. Ведь зачастую достаточно щелкнуть кнопкой, и из нарисованной концептуальной модели создается физическая БД! Но, с другой стороны, качество результирующего продукта, выходящего из-под «пера» CASE-инструмента, и качество «ручной работы» профессионального программиста сегодня сравнивать не стоит.



В результате труда разработчика баз данных должна появиться БД, представляющая собой некоторую абстракцию весьма сложной части реального мира. Вновь созданная БД должна с заданной степенью точности соответствовать настоящему миру и четко решать поставленные ей задачи по хранению и обработке данных. Именно поэтому процесс разработки БД включает в себя этап концептуального проектирования, на котором программист создает упрощенную модель будущих данных.

Наиболее показательной концептуальной моделью считается модель «сущность-связь» Питера Чена. ER-моделирование основано на принципах нисходящего подхода (от общего к частному), разработчик сначала выявляет типы сущностей и постепенно опускается до уровня атрибутов и связей.

Диаграммы ER-модели отличаются простотой и хорошей информативностью, что позволяет нам не только проектировать будущую БД, но и оценить адекватность этой модели в беседе с заказчиком программного обеспечения.

Вопросы для самопроверки

1. Что понимается под концептуальным проектированием БД?
2. Какую роль в процессе проектирования базы данных играют высокоуровневые модели данных?
3. Назовите основные концепции ER-модели и укажите способ их представления на диаграммах Питера Чена.
4. Дайте классификацию атрибутов ER-модели.
5. Разъясните предназначение подтипов сущностей.
6. Какие типы связей поддерживает ER-модель?
7. Что такое «мощность» связи?
8. Какие нотации ER-моделей вам известны?
9. Какие CASE-системы позволяют создавать ER-модели БД?

Глава 7

Логическое проектирование и нормализация

В этой главе мы обсудим процесс разработки качественных реляционных таблиц, но на этот раз перейдем с концептуального уровня проектирования на логический. Если вы ненадолго вернетесь к 3-й главе этой книги, то вспомните минимальный набор требований, предъявляемых к таблице, претендующей на высокое звание реляционной. Уже тогда мы упомянули термин «нормальная форма». **Нормализация** преследует несколько целей. Наиболее важная из них – борьба с избыточностью данных. Устранение избыточности достигается не просто за счет рационального назначения размерности атрибутам таблиц, а за счет формирования эффективных взаимоувязанных табличных структур, объединяющих несколько отношений. Так что процесс нормализации выступает в роли основного элемента процесса проектирования БД.

Замечание

Процесс нормализации и ER-моделирование выступают разными сторонами одной медали. Оба подхода приводят к созданию модели БД, с той лишь разницей, что ER-моделирование осуществляется на концептуальном уровне и основано на принципе от общего к частному (так называемый нисходящий подход). Нормализация производится на этапе логического проектирования и, в отличие от ER-моделирования, исповедует идеи восходящего подхода (от частного к общему).

Замечание

В результате нормализации БД таблицы приобретают дополнительную гибкость, что позволит нам без особых сложностей изменять структуру таблиц в будущем, например в процессе модернизации действующей БД.

Победа в борьбе за лишние байты не только приводит к минимизации расходов на хранение данных, но и устраняет ряд косвенных проб-

лем, точнее аномалий, присущих неграмотно построенным таблицам. К таковым относятся аномалии вставки, редактирования и удаления данных. Для демонстрации указанных аномалий рассмотрим фрагмент складской ведомости, содержащей информацию о поступлениях товаров на склад предприятия. Данные представлены в формате очень далекой от реляционной модели электронной таблицы Microsoft Excel и содержат колонки с именем поставщика, номером приходной накладной с датой поступления, классификацией продукции, наименованием продукции и т. д. (рис. 7.1).

Поставщик	Накладная	Классификатор	Наименование	Производитель	Количество	Цена за ед.	Склад	Итого
Арт-дизайн	№ 1 от 2018-01-11	Ноутбуки	Ноутбук-трансформер DELL Inspiron	Dell	4	33 000,00 Р	Основной склад	132 000,0
		Ноутбуки	Ноутбук-трансформер DELL Inspiron	Dell	4	25 300,00 Р	Основной склад	101 200,0
Компьютер-сервис	№ 2 от 2018-01-12	Моноблоки	Моноблок APPLE iMac MME02RU/A	Apple	2	70 000,00 Р	Основной склад	140 000,0
		Моноблоки	Моноблок APPLE iMac MNE02RU/A	Apple	2	95 600,00 Р	Основной склад	191 200,0
		Моноблоки	Моноблок APPLE iMac Z0R5000P7	Apple	2	126 000,00 Р	Основной склад	252 000,0
		Моноблоки	Моноблок APPLE iMac MNE02RU/A	Apple	1	144 000,00 Р	Основной склад	144 000,0
Микс	№ 3 от 2018-01-15	Жесткие диски и SSD	Жесткий диск WD Blue WD3200LPCK	Western Digital	4	2 200,00 Р	Основной склад	8 800,0
		Жесткие диски и SSD	Жесткий диск TOSHIBA DT01ACA050	Toshiba	4	2 240,00 Р	Основной склад	8 960,0
		Жесткие диски и SSD	Жесткий диск SEAGATE Barracuda ST10000DM007	Seagate	10	2 400,00 Р	Основной склад	24 000,0
		Жесткие диски и SSD	Жесткий диск TOSHIBA P300 HDWD10	Toshiba	10	2 550,00 Р	Основной склад	25 500,0
		Жесткие диски и SSD	Жесткий диск SEAGATE Barracuda ST10000DM007	Seagate	6	2 650,00 Р	Основной склад	15 900,0
Северный ветер	№ 4 от 2018-01-15	Процессоры	Процессор AMD A4 4000	AMD	4	1 200,00 Р	Основной склад	4 800,0
		Процессоры	Процессор AMD Athlon X2 340	AMD	4	1 380,00 Р	Основной склад	5 520,0
		Процессоры	Процессор AMD A4 4000	AMD	4	1 490,00 Р	Основной склад	5 960,0
		Процессоры	Процессор AMD A4 6320	AMD	8	1 650,00 Р	Основной склад	13 200,0
		Процессоры	Процессор AMD Sempron 2650	AMD	6	1 890,00 Р	Основной склад	11 340,0
		Процессоры	Процессор INTEL Celeron G3930	Intel	2	2 210,00 Р	Основной склад	4 420,0
		Процессоры	Процессор INTEL Celeron G3900	Intel	2	2 300,00 Р	Основной склад	4 600,0
Стелла	№ 5 от 2018-01-16	Материнские платы	Материнская плата ASUS A68HM-K	Asus	2	2 200,00 Р	Основной склад	4 400,0
		Материнские платы	Материнская плата GIGABYTE GA-78L	Gigabyte Technology	2	2 300,00 Р	Основной склад	4 600,0
		Материнские платы	Материнская плата GIGABYTE GA-H8	Gigabyte Technology	2	2 450,00 Р	Основной склад	4 900,0
		Материнские платы	Материнская плата MSI H81M-E33	Micro-Star International	3	2 480,00 Р	Основной склад	7 440,0
		Материнские платы	Материнская плата ASUS M5A78L-M	Asus	3	2 550,00 Р	Основной склад	7 650,0
		Память	Модуль памяти AMD R532G1601S1S	AMD	10	900,00 Р	Основной склад	9 000,0
		Память	Модуль памяти AMD R332G1339U1S	AMD	10	980,00 Р	Основной склад	9 800,0
Электрон-люкс	№ 6 от 2018-01-16	Телевизоры	LED телевизор LG 28LH451U "R", 28"	LG electronics	4	11 000,00 Р	Основной склад	44 000,0
		Телевизоры	LED телевизор LG 28MT48S-P2	LG electronics	4	13 800,00 Р	Основной склад	55 200,0
		Телевизоры	LED телевизор LG 32LW300C	LG electronics	6	15 600,00 Р	Основной склад	93 600,0

Рис. 7.1. Информация о поступлениях товаров на склад предприятия

Для обычного покупателя или продавца вся совокупность или любая часть собранной в таблице информации понятна и находит однозначное толкование. В результате самой первой поставки от компании «Арт-дизайн» стеллажи нашего склада пополнились ноутбуками компании Dell. Работник магазина сделал об этом соответствующую запись в книге учета. Все ноутбуки оприходованы по накладной № 1 от 11.01.2018. В поступлении имеется два типа ноутбуков: 4 шт. по цене 33 000 руб. и 4 шт. по цене 25 300 руб. При внесении второй строки работник посчитал, что ему не требуется повторно указывать название

фирмы-поставщика, номер и дату приходного ордера. Ведь и так понятно, что продукция из одной партии. Этим же правилом оператор компьютера руководствовался каждый раз, сталкиваясь с подобными повторами.

Предложенный на рис. 7.1 подход к обработке информации вполне приемлем для небольшого магазина, владелец которого сам с собой заключил соглашение о способе учета товара. Но, в отличие от нас, компьютер более строг, и ему в нашей таблице может много чего не понравиться. Вполне вероятно, наш электронный помощник не сможет правильно сопоставить строку товара с накладной из-за незаполненных ячеек таблицы. При построении фильтра по столбцу компьютер однозначно ошибется, если при введении классификатора мы сначала напишем «Материнские платы», затем «Motherboard», а третий раз вообще остановимся на сокращении «МП». Наш мозг понимает, что речь ведется об одном и том же, а для компьютера это не так. Вполне внятная с нашей точки зрения информация даже для самого совершенного процессора может оказаться некорректной. Судите сами. Наш электронный помощник мыслит категориями единичек и ноликов, поэтому с его точки зрения (и, кстати, не лишенной основания) «AMD» и «Amd» – абсолютно разные названия производителя процессоров, а строки без названия поставщика и накладной вообще могут показаться некорректными. Так что при включении в таблицу очередной строки нам требуется играть по правилам компьютера, иначе мы столкнемся с **аномалией вставки**.

Аномалию удаления продемонстрируем на примере имени поставщика и номера приходной накладной. Вновь взглянем на рис. 7.1. Запомним, что большинство поставок не ограничивалось одной строкой и включало две и более позиций. Так, по самому первому в списке приходному ордеру № 1 на склад поступило два наименования товара, по № 2 – 4 и т. д. Допустим, что в ордер № 3 мы ошибочно внесли жесткий диск «WD Blue WD3200LPCX» и удаляем связанную с ним строку из таблицы. Вместе с диском мы рискуем удалить и все данные о номере и дате приходного ордера. В результате оставшиеся строки накладной просто потеряют связь с поставщиком и накладной или, что может оказаться еще хуже, перейдут под «опеку» предыдущей накладной. Качество товаров от этого не пострадает, но нарушится весь складской учет и отчетность бухгалтерии. Поэтому при удалении строки с данными о поставщике и ордере нам потребуется перенести эти сведения в следующую за удаляемой строку. Эти дополнительные действия оказались бы лишними, если бы данные о поставках были правильно структурированы.

Продолжим критику ненормализованных таблиц. Как вы заметили, в нашем примере встречаются повторяющиеся данные, чтобы убедиться в этом, просто взгляните на столбцы «Поставщик», «Классификатор» и «Производитель». Многократные повторы существенно усложняют их обновление. Допустим, мы решим вместо классификатора «Память» использовать более точное название «Модуль памяти». В этом случае, дабы внести необходимые правки, нам придется кропотливо просмотреть все строки исходной таблицы. Это пример **аномалии редактирования** ненормализованных данных.

Для устранения трех перечисленных типов аномалий придется изрядно потрудиться – провести нормализацию данных. Нормализация – это процесс последовательного преобразования таблиц базы данных к виду, принятому в реляционной модели данных.

Замечание

Специалисты различают пять последовательных этапов (форм) приведения данных к реляционному виду от первой нормальной формы (First Normal Form, 1NF) до пятой (5NF). На каждой из ступеней нормализации таблица приобретает новые черты, которые потребуются для перехода к очередному этапу. Поэтому, чтобы прийти к заключительной нормальной форме, надо на каждом из этапов шаг за шагом методично совершенствовать структуру таблиц. Перепрыгнуть хотя бы через одну из ступеней нормализации нельзя!

Первая нормальная форма

В 4-й главе этой книги уже вскользь поговорили о минимальном наборе требований к реляционной таблице. Среди требований был упомянут один из важнейших принципов построения реляционных отношений – атомарность (одна ячейка таблицы – одно значение). Указанный принцип и станет героем первой части рассказа о нормализации данных.

Практически любой человек, делающий первые шаги на поприще проектирования БД, повторяет одну и ту же ошибку – собирает всю подлежащую хранению информацию в одну таблицу. Более того, видимо, находясь в поиске острых впечатлений, начинающий разработчик применяет максимум стараний для того, чтобы данные в таблице расположились наихудшим способом.

Самое худшее, что можно сделать с реляционной таблицей, – это разместить в ней **повторяющиеся группы данных**. Повторяющаяся группа появляется каждый раз, когда в одну ячейку таблицы мы пыта-

емся вставить больше одного значения. В примере склада автор не стал совершать такой ошибки, но в повседневной практике встречал регулярно. Например, достаточно велико искушение в текстовом столбце классификатора товаров (допустим, для строки с какой-нибудь микроволновой печью) написать следующее: «Бытовая техника; Кухонная техника; Мелкая бытовая техника; Печи микроволновые».

И все это делается из самих благих побуждений – якобы такое решение позволит пользователю более быстро найти нужный товар... Далеко не факт.

Приведем еще один отрицательный пример повторяющихся групп (как говорится, для закрепления). Допустим, мы с вами разрабатываем БД для кинотеатра. В этой БД, помимо всего прочего, необходимо каждый фильм характеризовать жанром. Сказано – сделано. Добавим в таблицу столбец «Жанр». Теперь можно почивать на лаврах? Да как бы не так! Все было бы здорово, если бы фильм можно было оценить одним жанром, допустим «боевик» или «фантастика». Но на практике все гораздо сложнее. Как мы поступим с фильмом в прокате, помеченным как «фантастический комедийный боевик» или «триллер, детектив, драма»? Занесем все жанры в одну текстовую ячейку через запятую? С точки зрения реляционной теории это категорически запрещено! Ведь, приходя в театр, кино или на стадион, мы рассчитываем на отдельное место и сразу требуем администратора, обнаружив в своем кресле другого зрителя. С реляционными данными действует тот же самый закон: одна ячейка – одно значение.

Безусловно, из правил бывают и исключения. Иногда по тем или иным причинам разработчикам БД приходится идти на «преступление» и закрыть глаза на запрет повторения групп. Такое вполне вероятно при хранении в БД массивов, включающих сотни значений. Но в подобной ситуации надо готовиться к целому букету неприятностей. Как минимум разработчик столкнется с проблемами выборки данных при построении запросов. Допустим, что нам захочется найти все исторические фильмы. Если мы строго следовали правилу одна ячейка – одно значение, то задача решается элементарно. Но если нет, то задачка для первого класса превращается в существенную проблему. Справедливости ради отметим, что язык запросов SQL обладает средством для решения подобных задач. Для этого предназначен предикат LIKE (см. главу 12), кроме того, ряд СУБД дополнительно поддерживает регулярные выражения, позволяющие препарировать текстовые данные на более глубоком уровне (см. главу 15). Но, как всегда, положительная сторона LIKE и регулярных выражений с лихвой перерывается большим недо-

статком – запросы, основанные на перечисленных механизмах, весьма ресурсоемки и медлительны.

Упомянем еще один характерный промах начинающего разработчика, связанный с нарушением атомарности значения. Вновь посмотрим на рис. 7.1, на этот раз нас интересует столбец «Накладная», в нем совместно хранится два разнотипных значения: номер накладной и дата. Это неверно, мы нарушаем правило атомарности.

Замечание

Под атомарностью значения понимается, с одной стороны, его целостность, а с другой – неделимость. Разделение данных на «атомы» впоследствии позволит нам вооружить БД дополнительным сервисом, в частности разнообразными способами сортировки данных и расширенными методами фильтрации и поиска данных.

В стремлении соблюдать правило атомарности не стоит перегибать палку. Допустим, в базе кинофильмов нам придется учитывать режиссеров, продюсеров, актеров и т. д. Не секрет, что имя состоит из фамилии, имени и отчества. Если строго следовать правилу атомарности, то для этой цели потребуется три столбца. Но в данном случае есть место для споров. Многие программисты вправе возразить, что все три ингредиента имени человека логично хранить вместе в одном столбце таблицы – ведь по своей сути имя целостно. Автор не позиционирует себя как противника этой идеи. Тем более если заказчик БД даст гарантию, что конечные пользователи всегда будут корректно заполнять это поле. Ведь не исключено, что один сотрудник введет строку «Иванов Иван Иванович», другой – «Иванов И. И.», третий – «И. И. Иванов». С точки зрения электронно-вычислительной машины, это три абсолютно разных человека, и сумятица с именами впоследствии затруднит редактирование, поиск и сортировку данных.

В контексте темы атомарности стоит рассмотреть и другой пример. Дата также состоит из трех значений – года, месяца и дня. Как лучше организовать ее хранение? В виде отдельных элементов или как одно целое? Зачастую ответ на этот вопрос можно получить, только проанализировав всю логику работы БД. Впрочем, большинство разработчиков обычно считает дату (и время) единым и неделимым значением. Основная причина такого убеждения в том, что все современные СУБД для представления даты/времени ввели соответствующие типы данных, более того, эти типы данных стандартизированы в SQL.

Определение

Таблица приведена к первой нормальной форме, если в ней отсутствуют повторяющиеся группы и все значения, хранимые в ней, неделимы (атомарны).

Итак, в результате приведения нашей таблицы учета поступления товаров к 1NF мы получим перечень столбцов, представленный на рис. 7.2.

Товары на складе	
	Поставщик
	Номер накладной
	Дата накладной
	Производитель
	Наименование
	Количество
	Цена за ед.
	Склад
	...

Рис. 7.2. Таблица склада в состоянии 1NF

На первый взгляд, в результате приведения к 1NF таблица БД становится неповоротливой – в ней могут образоваться десятки или даже сотни столбцов. Но это лишь первый шаг в цепочке преобразований, и на первой ступени нормализации это не столь важно. Еще один недостаток 1NF – чрезмерная избыточность данных, выраженная в многочисленных дублированиях одних и тех же значений. Это также не повод для волнений, т. к. с переходом к более высоким формам нормализации и этот недостаток нивелируется.

В любом случае, 1NF – первый и очень важный шаг в построении будущей реляционной БД. Собрав нормализованную таблицу, разработчик переходит к приведению отношения к более высоким нормальным формам, а для этого входящие в отношение атрибуты проверяются на функциональную зависимость между собой.

Функциональная зависимость атрибутов

Функциональные зависимости – это те связи, которые могут возникнуть между атрибутами отношений. Говорят, что один атрибут отношения (пусть он называется А) функционально зависит от другого (назовем его В), если каждому значению А однозначно соответствует значение В. Другими словами, если в столбце А таблицы будут вносить-

ся одинаковые значения, то в соответствующем столбце В также окажутся одинаковые данные. В символической форме функциональная зависимость записывается следующим образом: $A \rightarrow B$. Левую часть такой записи называют детерминантом, а правую – зависимой частью. Степень функциональной зависимости может оказаться разной:

- частичная зависимость;
- полная зависимость;
- транзитивная зависимость;
- многозначная зависимость.

Частичная зависимость возникает при наличии взаимосвязи между отдельным атрибутом и группой атрибутов отношения, выполняющих роль ключа отношения. Это тот случай, когда какой-то из атрибутов отношения окажется в функциональной зависимости только от определенной части составного ключа, а не от всего ключа в целом. На второй ступени нормализации мы обязаны устранить все частичные зависимости и добиться полной функциональной зависимости (т. е. каждый столбец, не входящий в состав ключа таблицы, должен полностью от него зависеть).

Полная зависимость представляет собой прямую противоположность частичной. Примером полной функциональной зависимости может стать, с одной стороны, индивидуальный номер налогоплательщика (ИНН), а с другой – фамилия, имя и отчество счастливого обладателя ИНН. Нетрудно догадаться, что при наличии функциональной зависимости между двумя атрибутами справедливо как прямое $A \rightarrow B$, так и обратное $B \rightarrow A$ утверждение. То есть по ИНН можно узнать ФИО человека, а по ФИО – выяснить ИНН. Полную зависимость между атрибутами обозначают следующим образом: $A \leftrightarrow B$.

Транзитивная зависимость возникает между двумя атрибутами, когда они связаны друг с другом через посредника. Например, когда атрибут А зависит от В, а В, в свою очередь, зависит от С: $A \rightarrow B \rightarrow C$, но обратная зависимость $A \leftarrow B \leftarrow C$ отсутствует.

Наконец, **многозначная зависимость** имеет место в ситуации, когда одному значению атрибута А соответствует несколько значений атрибута В.

Замечание

Наличие транзитивной или многозначной зависимости между атрибутами одного отношения можно считать достаточным условием, подтверждающим необходимость разнесения этих атрибутов (читай: столбцов таблицы) по разным отношениям.

Наряду с зависимыми существуют и независимые атрибуты. Это тот случай, когда между двумя атрибутами нет прямой связи. Например, в наших данных к независимым атрибутам можно отнести поставщика и склад. Для обозначения независимости A от B применяют следующую запись: $A \nrightarrow B$. Соответственно, если B также не зависит от A : $B \nrightarrow A$. Взаимную независимость двух атрибутов принято обозначать $A \nleftrightarrow B$.

На рис. 6.3 представлена схема зависимостей между атрибутами проекта базы данных склада. На ней вы обнаружите все перечисленные выше разновидности зависимостей. Построение подобных схем на этапе нормализации может оказаться весьма полезным для начинающих программистов, так как они позволяют разобраться в хитросплетениях данных и упрощают процесс выявления зависимостей между атрибутами отношений.

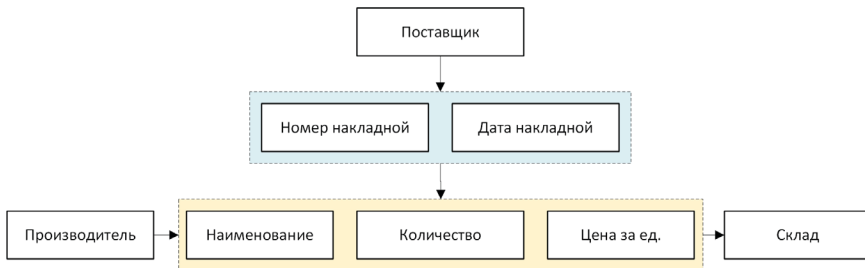


Рис. 7.3. Схема зависимостей между атрибутами

Процесс установления функциональной зависимости тесно связан с порядком назначения ключевых полей таблицы. У нашего (приведенного к 1NF) отношения пока нет атрибутов, переведенных в разряд ключевых. Но как только мы выберем ключевые столбцы, то все оставшиеся неключевые атрибуты станут в той или иной степени зависеть от ключа таблицы.

Порядок определения первичного ключа

Перед рассмотрением второго этапа нормализации ненадолго переведем нашу беседу в плоскость обсуждения порядка назначения ключа для отношения. Нам уже известно, что каждый кортеж таблицы нуждается в уникальной идентификации (см. главу 4). В качестве идентификатора может использоваться одно или несколько полей таблицы. На высокое звание ключа отношения вправе претендовать набор атрибутов, который полностью и однозначно определяет значения всех остальных неключевых атрибутов. Встречаются случаи, когда одна таблица содер-

жит несколько вариантов сочетаний атрибутов, претендующих на роль первичного ключа (например, автомобиль можно идентифицировать номером кузова и номером двигателя), тогда говорят, что отношение содержит несколько **потенциальных ключей**.

Замечание

Между атрибутами, входящими в состав первичного ключа, и всеми остальными неключевыми атрибутами должна существовать полная функциональная зависимость.

Какие из столбцов нашей таблицы (см. рис. 7.2) вправе стать потенциальным ключом? Сразу отмечаем версию о возможности построения ключа на основе всего одного атрибута. На первый взгляд, может показаться, что на звание потенциального ключа претендует поле с номером накладной, но ведь этот же номер может повториться в будущем году. Не окажется прав и тот, кто предположит, что для построения ключа достаточно столбцов с номером и датой приходного ордера. Если вы внимательно изучите рис. 7.1, то заметите, что по накладной № 1 было получено два наименования товара (что соответствует двум строкам таблицы). Таким образом, в состав ключа придется включить целых три столбца: номер накладной, дата накладной, наименование. На данный момент согласимся с этим решением, хотя буквально на следующей странице на роль первичного ключа будет предложен другой кандидат.

Построение ключей из нескольких атрибутов – крайне неблагодарное занятие. Бич таких таблиц – частичная зависимость некоторых атрибутов таблицы от части первичного ключа. Частичная зависимость может возникнуть только в той таблице, у которой ключ составной. Поэтому разработчики баз данных в качестве первичного ключа стараются использовать только один атрибут. Чаще всего для этих целей в таблицу внедряется дополнительный столбец, не несущий никакой полезной нагрузки с точки зрения хранящейся в таблице информации, но содержащий заведомо уникальные значения. В простейшем случае это целочисленное поле автоинкрементного типа. По сути, это не что иное, как обычный счетчик. Первой появившейся в таблице строке присваивается 1, второй – 2 и т. д. Автоинкрементное поле гарантирует, что даже при удалении кортежа из таблицы его уникальный номер не будет назначен какой-либо из вновь вводимых строк.

Замечание

Частичная зависимость между атрибутами может возникнуть только в той таблице, у которой ключ составной (построен из нескольких столбцов).

Современные СУБД позволяют настраивать поведение поля-счетчика. Ниже представлен фрагмент кода создания таблицы на диалекте Transact-SQL системы управления базами данных SQL Server.

```
CREATE TABLE SUPPLIERS
  (SUPPLIERS_ID INTEGER IDENTITY(1,1) PRIMARY KEY,
   SNAME CHAR(20) NOT NULL,
  ...,...)
```

Во второй строке кода мы указываем на то, что в таблице SUPPLIERS следует создать поле первичного ключа SUPPLIERS_ID на основе счетчика. Особенности работы счетчика определяются двумя параметрами, указанными после слова IDENTITY. Первый из параметров назначает значение, с которого стартует отсчет, а второй – шаг счетчика. Как видите, и исходное значение, и шаг приращения могут быть любыми. Для чего это нужно? Хотя бы для ситуации, когда два (или более) экземпляра, идентичных БД, работают на разных складах и компьютеры не соединены сетью. Однако нам требуется предусмотреть возможность объединять информацию из двух БД в единое целое. Как это сделать без конфликта первичных ключей? Очень просто! В первом экземпляре БД счетчики таблиц настраиваем только на нечетные IDENTITY(1,2), а во втором – на четные значения IDENTITY(2,2). В результате при слиянии БД значения первичных ключей двух таблиц никогда не пересекутся.

Замечание

Автоинкрементный атрибут таблицы – хороший, но далеко не единственный способ определения искусственных первичных ключей. Уникальность в состоянии гарантировать поля, построенные на основе меток дата/время (стандартизированный в SQL тип данных TIMESTAMP), и 128-битные поля глобальных уникальных идентификаторов GUID (Globally Unique Identifiers).

Вторая нормальная форма

Говорят, что таблица приведена ко второй нормальной форме, если она приведена к 1NF и в ней отсутствуют частичные зависимости. Другими словами, у такой таблицы не должно быть атрибутов, зависящих только от части первичного ключа.

Вернемся к нашему проекту. Предположим, что мы определили составной первичный ключ таблицы на основе атрибутов «Номер накладной», «Дата накладной» и «Наименование». Обсудим это решение с точки зрения частичных функциональных зависимостей. Наличие со-

ставного ключа привело к тому, что такой атрибут, как «Поставщик», зависит от первой половины ключа («Номер накладной» и «Дата накладной»), а атрибут «Производитель» зависит от другой части ключа – «Наименование». Следовательно, наша исходная таблица не соответствует требованиям 2NF.

Отказ от составного первичного ключа имеет еще один неоспоримый плюс – упрощается процесс приведения таблицы к 2NF. Доказательство на поверхности: если первичный ключ не является составным, то нет и частичных зависимостей. Нам осталось лишь провести очередную доработку таблицы склада. Теперь в ней появился еще один столбец «ПКлюч», представляющий собой поле-счетчик (рис. 7.4).

Товары на складе	
РК	ПКлюч
	Поставщик
	Номер накладной
	Дата накладной
	Производитель
	Наименование
	Количество
	Цена за ед.
	Склад
	...

Рис. 7.4. Таблица «Склад» в состоянии 2NF

После приведения к 2NF между атрибутом ключа «ПКлюч» и любым другим столбцом таблицы установилось однозначное соответствие, частичные зависимости устранены.

Определение

Таблица приведена ко второй нормальной форме, если она соответствует 1NF и в ней нет частичных зависимостей, т. е. нет неключевых столбцов, зависящих от части первичного ключа.

Третья нормальная форма

Нашему проекту еще так далеко до совершенства! На втором этапе нормализации за счет отказа от составного ключа мы в один миг избавились от частичных зависимостей. Но даже такого грандиозного успеха по-прежнему недостаточно, для того чтобы наша таблица стала идеальной, ведь в ней притаился очередной враг реляционных БД – транзитивные зависимости. Дабы вы не утруждали себя перелистыванием страниц к тому месту главы, где мы уже говорили об этой разновиднос-

ти функциональных зависимостей, напомним, что под транзитивной зависимостью понимается такая зависимость между атрибутами отношения, когда один из атрибутов связан с другим через атрибут-посредник $A \rightarrow B \rightarrow C$.

В таблице склада посредников больше чем достаточно – она прячет в себе несколько транзитивных зависимостей. Очевидный пример транзитивности – «Поставщик». Указанный атрибут непосредственно связан лишь с атрибутами, описывающими накладную («Номер накладной» и «Дата накладной»), а все остальные атрибуты связаны с «Поставщик» транзитивно (рис. 7.3).

Еще один фактор, свидетельствующий о транзитивной зависимости, – возникновение связи «один ко многим» между атрибутами таблицы. В нашем случае поставщик может осуществить много поставок, соответственно, между поставщиком и накладными имеется связь 1:M.

Определение

Таблица приведена к третьей нормальной форме, если она соответствует 2NF и в ней отсутствуют транзитивные зависимости.

Транзитивная зависимость $A \rightarrow B \rightarrow C$ устраняется одним-единственным способом – за счет выделения атрибутов $B \rightarrow C$ (или $A \rightarrow B$) в другую таблицу. Связи между такими таблицами строятся по ключевым столбцам: первичный ключ главной таблицы соединяется с внешним ключом подчиненной таблицы. В результате атрибут «Поставщик» выделяется в отдельную таблицу «Поставщики», то же самое происходит и с атрибутами накладной – они переносятся в таблицу «Приходные накладные» (рис. 7.5).

Замечание

Для понимания сути третьей ступени нормализации предложим неформальное определение 3NF: внутри приведенной к 3NF таблицы должны отсутствовать связи «один ко многим».

В нашем случае преобразование таблицы из 2NF в 3NF приводит к декомпозиции исходного отношения в набор из 5 отношений. Здесь можно выделить 3 главные таблицы: «Поставщики», «Производители» и «Классификатор». Перечисленные таблицы называются главными потому, что они не зависят ни от каких других таблиц (специалисты по

ER-моделированию сказали бы, что они представляют сильный тип сущности). Зачастую подобные таблицы называют «справочными», так как они являются поставщиками справочных данных для подчиненных по отношению к ним таблицам.

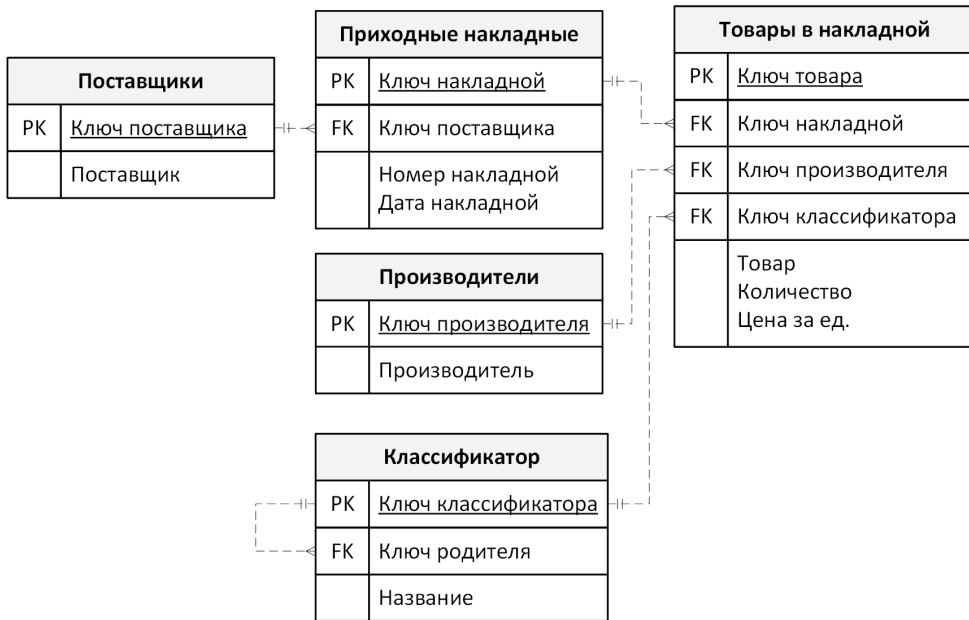


Рис. 7.5. Фрагмент БД с таблицами, приведенными к 3NF

Справочные таблицы весьма полезны, благодаря им:

- существенно сокращается физический размер дочерней таблицы. Потому что теперь вместо реальных данных (названий поставщиков, названий производителей, классификации товара), занимающих сотни байт, в подчиненной таблице требуется хранить значение внешнего ключа, которое обычно не превышает 4 байт;
- при редактировании значения (например, переименование поставщика) нет необходимости просматривать сотни записей, а достаточно изменить одно значение в справочной таблице;
- при вводе данных значительно снижается вероятность появления ошибки пользователя, так как вместо ввода с клавиатуры ему предлагаются на выбор заранее подготовленные значения.

Все остальные таблицы получают из главных таблиц справочную информацию.

Нормальная форма Бойса-Кодда

Помимо 3NF, специалисты выделяют усиленную разновидность 3NF – нормальную форму Бойса-Кодда 3NFBC (Boyce-Codd). Смысл усиления в том, что во время формулирования первоначальных требований к 3NF Кодд не предусмотрел вероятность того, что в нормализуемом отношении может существовать более одного потенциального ключа, указанные ключи окажутся составными и эти ключи станут обладателями хотя бы одного общего атрибута.

Вероятность совместного возникновения перечисленных событий крайне невысока, но все-таки не исключена. Поэтому и предусмотрено более строгое определение 3NFBC. Прежде чем мы с ним познакомимся, вспомним, что левая часть символической записи $A \rightarrow B$ называется детерминантом функциональной зависимости. А теперь дошла очередь и до определения.

Определение

Таблица приведена к третьей нормальной форме Бойса-Кодда, когда детерминанты всех ее функциональных зависимостей являются потенциальными ключами.

Вспоминать о нормальной форме Бойса-Кодда стоит только в ситуации, когда в таблицах мы активно применяем составные ключи. Если же при проектировании БД разработчик опирается на идею искусственных (например, автоинкрементных), состоящих из одного атрибута ключей, о существовании 3NFBC можно забыть.

Четвертая нормальная форма

Если третья нормальная форма призвана для борьбы с транзитивными зависимостями, то героиня этой части главы 4NF состоит в конфронтации с другим злом реляционной модели – многозначными зависимостями. Многозначная зависимость – это не что иное, как связь «многие ко многим».

Итак, база данных, соответствующая четвертой ступени нормализации, обязана избавиться от многозначных зависимостей между атрибутами отношений. Но каким образом? Ведь реляционная модель специализируется только на связях «один к одному» (1:1) и «один ко многим» (1:M).

Для реализации связи «многие ко многим» (M:N) приходится идти на хитрость. Она заключается во введении дополнительной таблицы, ко-

торая разделит связь $M:N$ на две $1:M$. Мы так уже поступали при изучении ER-модели. Возвратитесь к рис. 6.7, на нем приведен классический пример связи $M:N$. Один сотрудник может выполнять несколько заказов, и наоборот – один заказ может выполняться несколькими сотрудниками. Решение проблемы также отражено на этом рисунке, мы создали промежуточную таблицу (ассоциативное отношение) и забыли о многозначной зависимости.

Могут ли многозначные зависимости появиться в примере БД для склада? Безусловно. Предложенный на рис. 7.5 фрагмент БД пока описывает только фрагмент БД, если точнее – накладную, по которой товар поступает на наше предприятие, но в логической модели пока не решена задача перераспределения товара внутри предприятия. Поэтому нам необходимо доработать БД.

Во-первых, вся поступающая по приходным накладным продукция должна быть закреплена на складе по умолчанию (основной склад предприятия).

Во-вторых, должен существовать механизм передачи товара с основного склада в различные подразделения предприятия (другие склады, магазины, цеха и т. п.).

Теперь рассмотрим бизнес-логику работы БД в момент приема товаров. Допустим, по приходной накладной на основной склад предприятия поступает 10 холодильников, об этом создается соответствующая запись в таблице «Товары в накладной». Вполне вероятно, что пара холодильников из этой партии будет немедленно отправлена в магазин, а остальные 8 переместятся на другой склад (склады) нашей фирмы. Таким образом, между таблицей «Товары в накладной» и таблицей «Склады» (ее пока еще нет на схеме с рис. 7.5) формируется связь $M:N$.

Как отобразить решение этой задачи? Например, так, как предложено на рис. 7.6.

Обратите внимание на появление между таблицами «Товары в накладной» и «Склады» коммуникационной таблицы «Перемещаемый товар», благодаря ей нам удалось разделить связь $M:N$ на две связи $1:M$.

Определение

Таблица приведена к четвертой нормальной форме, если она соответствует 3NF и в ней отсутствуют многозначные зависимости (многие ко многим).

Замечание

В приложении 1 вы найдете окончательную версию логической модели для БД «Склад». Именно она будет использоваться в дальнейших главах в качестве опорной.



Рис. 7.6. Улучшенная БД с таблицами, приведенными к 4NF

Пятая нормальная форма

Для обычного разработчика БД пятая нормальная форма представляет скорее теоретический, нежели практический интерес. 5NF требует обеспечения беспрепятственной возможности перестройки данных в нормализованных таблицах. Приведение таблицы к высшей степени нормализации – крайне редкий случай. Это действие имеет смысл, только если таблица содержит так называемые зависимые сочетания.

Определение

Зависимые сочетания – это свойство декомпозиции, которое вызывает генерацию ложных строк при обратном соединении декомпозированных отношений с помощью операции естественного соединения.

Поясним на примере. Предположим, что в базе данных существует некое отношение, хранящее данные о поставщике, товаре (полученном от этого поставщика) и производителе, изготовившем товар (рис 7.7). Вполне вероятно, что в процессе работы с базой данных на основе исходной таблицы нам придется построить два запроса:

- 1) первый запрос вернул данные о поставщиках и товарах (мы его сохранили в виде таблицы TABLE2);
- 2) на базе второго запроса мы получили информацию о товарах и производителях (таблица TABLE3).

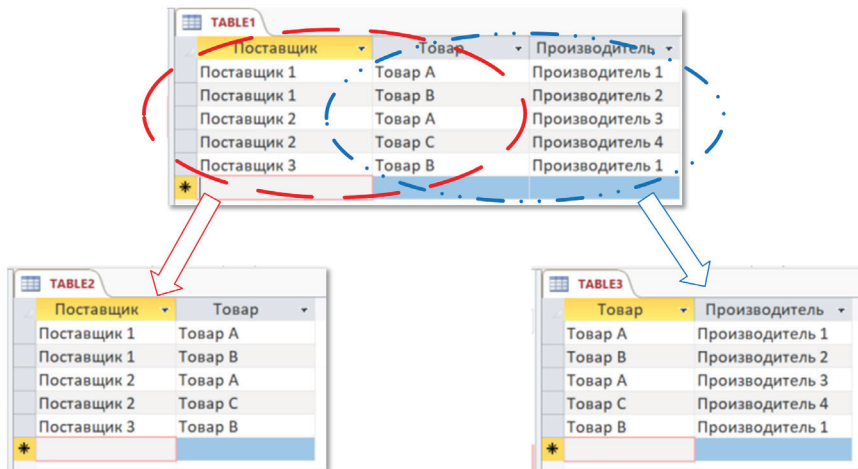


Рис. 7.7. Декомпозиция таблицы

Обратите внимание, что данные в новых таблицах вполне корректны и соответствуют значениям, содержащимся в исходной таблице.

А теперь попробуем решить обратную задачу – восстановим исходную таблицу TABLE1 на основе данных из TABLE2 и TABLE3. Для этого создаем обычный запрос на языке SQL, объединяющий обе таблицы по общему для них аргументу «Товар».

```
SELECT TABLE2.Поставщик, TABLE2.Товар, TABLE3.Производитель
FROM TABLE2 INNER JOIN TABLE3 ON TABLE3.Товар=TABLE2.Товар;
```

Не станем комментировать текст запроса. Тем, кто пока незнаком с языком SQL, придется немного потерпеть до главы 11, с которой мы начнем разговор о базовом языке реляционных БД. Главное – результат: после соединения двух таблиц мы получили четыре ложные (не существующие в реальной таблице) строки (рис. 7.8).

Исходное отношение

Поставщик	Товар	Производитель
Поставщик 1	Товар А	Производитель 1
Поставщик 1	Товар В	Производитель 2
Поставщик 2	Товар А	Производитель 3
Поставщик 2	Товар С	Производитель 4
Поставщик 3	Товар В	Производитель 1

Отношение после попытки объединения

Поставщик	Товар	Производитель
Поставщик 2	Товар А	Производитель 1
Поставщик 1	Товар А	Производитель 1
Поставщик 3	Товар В	Производитель 2
Поставщик 1	Товар В	Производитель 2
Поставщик 2	Товар А	Производитель 3
Поставщик 1	Товар А	Производитель 3
Поставщик 2	Товар С	Производитель 4
Поставщик 3	Товар В	Производитель 1
Поставщик 1	Товар В	Производитель 1

Рис. 7.8. Генерация ложных строк после объединения

Финал «гонки» нормальных форм

Нормальные формы вводились исследователями на протяжении 1970-х годов с целью устранить выявленные ими аномалии данных. И ввод каждой очередной НФ устранял определенную группу аномалий, но буквально сразу находилась новая подлежащая устранению аномалия. Даже казалось, что «гонка» нормальных форм будет продолжаться вечно, однако этот процесс пришел к логическому финалу в 1981 году.

В 1981 г. Р. Фагин (R. Fagin) в своей статье ввел понятие **доменно-ключевой нормальной формы** (domain/key normal form, DK/NF) и доказал, что любое отношение, приведенное к DK/NF, окажется свободным от всех возможных аномалий модификации¹. Таким образом Фагин указал на то, что введение нормальных форм более высокого порядка (с точки зрения борьбы с аномалиями модификации) не имеет смысла.

Определение

Отношение находится в доменно-ключевой нормальной форме, если каждое ограничение, накладываемое на это отношение, является логическим следствием определения доменов и ключей.

В работе [44] предлагается неформальное определение DK/NF: «отношение находится в DK/NF, если выполнение ограничений на домены и ключи приводит к выполнению всех ограничений». Таким образом, в состоянии DK/NF отношение может перейти на любой ступени нормализации, например после 2NF. И в дальнейшей нормализации больше не нуждаться.

В упомянутой работе [44] подмечено, что пока не известен ни один алгоритм преобразования отношения к доменно-ключевой нормальной форме, так что создание отношений в DK/NF является более искусством, чем наукой.

Резюме

Нормализация осуществляется на этапе логического проектирования БД и представляет собой вариант восходящего подхода, который начинается с установления связей между атрибутами. На практике для построения приемлемой логической модели БД следует пройти 4 ступени

¹ Fagin R. A normal form for relational databases that is based on domains and keys. ACM Transactions On Database Systems, September 1981, p. 387–415.

нормальных форм:

- 1NF. Все поля в таблицах неделимы и не содержат повторяющихся групп;
- 2NF. Все неключевые поля в таблицах зависят от первичного ключа;
- 3NF. В таблицах отсутствуют избыточные неключевые поля;
- 4NF. В таблицах устранены многозначные зависимости.

Пятая нормальная форма предназначена для устранения зависимых сочетаний и учитывается только при декомпозиции отношений.

Теория реляционных баз данных и вместе с ней правила нормализации не стоят на месте. В работе одного из ведущих специалистов по теории баз данных Криса Дейта уже упоминается вероятность появления шестой нормальной формы, правда с оговоркой, что если речь ведется «о классических операциях проекции и соединения, то пятая нормальная форма является последней нормальной формой» [19].

Вопросы для самопроверки

1. С какими аномалиями мы можем столкнуться при модификации данных в ненормализованной таблице?
2. На каком этапе проектирования БД (см. рис. 4.2) осуществляется нормализация?
3. Какие разновидности функциональных зависимостей вам известны?
4. Дайте определение первой, второй, третьей нормальной форм и нормальной формы Бойса-Кодда.
5. Что такое первичный ключ, и какое он имеет отношение ко второй нормальной форме?
6. Поясните назначение четвертой нормальной формы.
7. Что представляет собой пятая нормальная форма, и надо ли учитывать ее на этапе логического проектирования БД?
8. Почему процесс нормализации считается восходящим подходом?

Глава 8

Физическое представление данных

Буквально в первой главе книги мы с вами уже затрагивали вопрос организации хранения данных в БД. Правда, в тот раз мы обсудили неудачный подход, реализованный в системах, основанных на файлах. Картина дополняется тем, что в начале 60-х годов основным способом долговременного хранения данных выступала крайне ненадежная магнитная пленка, с которой осуществлялось последовательное считывание записей. Сегодня в качестве основного вторичного устройства хранения выступают магнитные жесткие диски, объемы которых на момент написания этих строк достигают 15 терабайт. В скором будущем на смену магнитным дискам придут твердотельные накопители (пока их размеры, стоимость и, самое главное, надежность далеки от приемлемых).

Двухуровневая модель хранения данных

В современных реляционных СУБД в качестве опорной применяется двухуровневая модель хранения данных (рис. 8.1).

1. Прикладная модель хранения размещается в оперативной (а при необходимости и в виртуальной) памяти. Доступ к данным, расположенным в оперативной памяти, отличается высокой скоростью, поэтому на данном уровне осуществляются операции, связанные с обслуживанием текущих запросов.
2. Модель хранения БД отвечает за долговременное хранение, данные заносятся во внешнюю память – на магнитные или твердо-

тельные диски сервера, здесь находи(я)тся файл(ы) базы данных. По мере необходимости СУБД осуществляет чтение блоков данных с жесткого диска сервера и загружает их в кеш СУБД, располагающийся в основной памяти. После модификации данных в кеше СУБД осуществляется обратное действие – запись данных на жесткий диск.

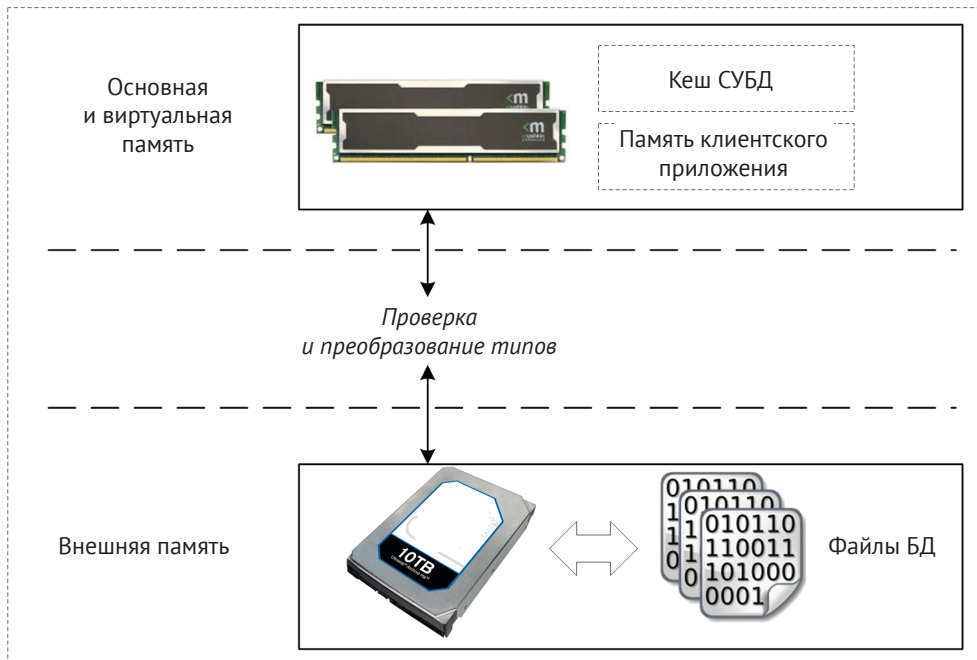


Рис. 8.1. Двухуровневая модель хранения данных

При переносе данных из основной памяти во внешнюю и при осуществлении обратной операции в двухуровневой модели зачастую приходится осуществлять операцию проверки, а при необходимости и преобразования типов данных (например, переводить значение даты/времени из формата вещественного числа, используемого в ряде языков программирования, на которых написаны клиентские приложения, в текстовый формат, стандартизированный в SQL).

Каким образом на внешних устройствах хранения размещаются отношения (если речь идет о реляционных БД) или описания объектов (для объектно-ориентированной модели данных)?

Замечание

Помимо широко распространенной в реляционных СУБД двухуровневой модели, существуют и другие подходы хранения данных, в частности в объектно-ориентированных СУБД из-за отсутствия необходимости осуществлять преобразование типов обычно используется одноуровневая модель.

Представление реляционных данных

Казалось бы, что может быть проще, чем физическое размещение на жестком диске отношения? Тем более что на логическом уровне реляционное отношение представляет собой плоскую двумерную таблицу, в большинстве случаев с фиксированным размером столбцов. Наверняка любой программист сможет решить подобную задачу без особых затруднений! Однако это лишь поверхностное суждение. На практике, чтобы оказаться на магнитном накопителе, реляционная таблица проходит ряд этапов преобразования (рис. 8.1).

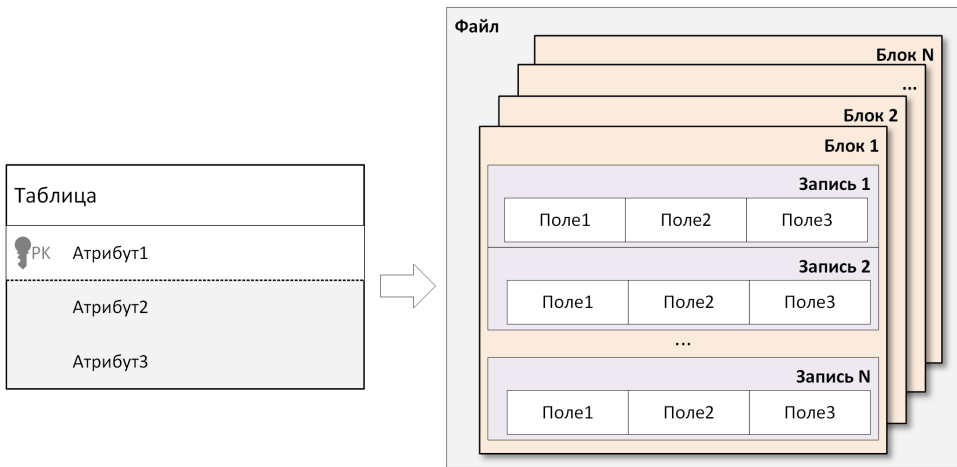


Рис. 8.2. Представление реляционной таблицы на устройстве хранения

Атрибуты (опорные элементы реляционной таблицы) на вторичном устройстве хранения отображаются в виде байтовых последовательностей, называемых полями (fields). В ситуации, когда столбец таблицы имеет фиксированный размер (допустим, он специализируется на хранении стандартного целочисленного типа данных INTEGER), в соответствие атрибуту ставится поле постоянной длины,

а если столбец не имеет четко заданного размера, мы получим поле переменной длины.

Поля объединяются в структуру, называемую записью (record). Обычно запись соответствует отдельной строке исходной таблицы (кортежное хранение). Из-за того, что в составе записи могут оказаться поля переменной длины, запись также не всегда оказывается фиксированного размера.

В свою очередь, записи сохраняются в контейнерах более высокого уровня – блоках (blocks). Физическая структура блока в первую очередь определяется особенностями модификации данных, принятыми в конкретной СУБД, ведь нам необходимо быстро и надежно осуществлять операции вставки, редактирования и удаления строк из таблицы.

Наконец, блоки объединяются в коллекции, которые называют файлами (files). Однако в данном случае не трактуйте термин файл как отдельный файл реляционной таблицы, который вы найдете на жестком диске вашего компьютера. Вполне вероятно, что это будет некий структурированный поток байтов, попадающий в очередной контейнер более высокой иерархии. В качестве примера можно привести распределенную однопользовательскую СУБД Microsoft Access, в которой все объекты БД (в том числе и таблицы) на конечном этапе попадают в один физический файл (обычно с расширением имени .accdb). С другой стороны, некогда очень популярные форматы БД Paradox и dBase создавали для отдельной реляционной таблицы отдельный файл.

Поля

Поля специализируются на обслуживании отдельного элемента данных – атрибута таблицы. Размер поля определяется типом данных соответствующего столбца, так что если вы определили столбец INTEGER, то для хранения целочисленного значения поле затребует от 4 до 8 байт (в зависимости от специфики СУБД). Создавая символьный столбец CHAR(*n*) во вторичной памяти, мы получим поле размерностью, позволяющей хранить *n* символов в заданной кодировке. Точно так же произойдет со всеми другими стандартными типами данных, размерность которых строго определена.

Значения даты и времени обычно хранятся в обычном текстовом формате – задействуется текстовая строка постоянной длины CHAR. Так, столбец типа DATE использует формат записи даты «yyyy-mm-dd». Таким образом, полное число позиций, требуемых для отображения даты (без разделителей), равно 8. Соответственно, для значения даты, вероятнее

всего, будет зарезервировано поле из 8 байт. Если столбец TIME представляет время в формате «hh-mm-ss», то без учета разделителей нам окажется достаточным поля из 6 байтов. Некоторые коррективы может внести необходимость хранения долей секунд, в последнем случае возможно применение типов данных переменной длины (таких как VARCHAR).

Битовая последовательность BIT(n) делится на группы по 8 бит и упаковывается в байты. Если размерность последовательности n не кратна 8, то в последнем байте остаются неиспользуемые биты – их система игнорирует. Примерно то же самое происходит и с булевым типом данных – соответствующему столбцу отводится 1 байт поля, но так как для хранения значения true/false достаточно 1 бита, оставшиеся 7 бит не используются.

Замечание

Некоторые СУБД допускают упаковку нескольких булевых величин (и других малых данных, занимающих меньше 8 бит) в один байт. Но такой подход целесообразен лишь при высокой стоимости хранения единицы информации или в условиях существенных ограничений по памяти (например, в мобильных устройствах).

Ситуация несколько усложняется, когда при определении столбца разработчик БД применит тип данных переменной длины. В качестве примера воспользуемся типом данных VARCHAR(n), специализирующимся на обслуживании текстовых строк, размер которых может изменяться в широких пределах. Для хранения подобных данных СУБД обычно резервирует поле размерностью $n+1$ байт (если речь идет о символах в 8-битной кодировке). Дополнительный байт используется одним из двух способов:

- 1) дополнительный байт размещается в самом начале поля и хранит число символов в строке, которое не может превышать значения n , первый значащий символ строки располагается во втором байте поля;
- 2) дополнительный байт хранит символ завершения строки – обычно NULL. В этом случае первый значащий символ текстовой строки заносится в первый байт поля, а NULL окажется в следующем байте за последним символом строки.

Обратите внимание на то, что первый вариант хранения ограничивает размер текстовой строки 255 символами, так как большее значение просто не может быть размещено в одном байте.

Записи

Строки таблицы на вторичных устройствах хранятся в форме записей. Самый идеальный случай – это когда все входящие в запись поля имеют фиксированный размер, тогда запись представляет собой обычное последовательное соединение полей. На предложенном на рис. 8.2 примере представлено подобное решение.

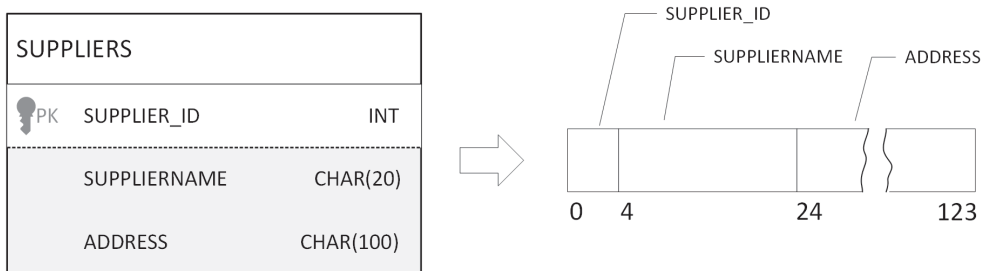


Рис. 8.3. Запись с полями постоянного размера

В кортеж таблицы SUPPLIERS входят три столбца (целочисленный 4 байта и 2 текстовых по 20 и 100 байт соответственно). На устройстве хранения строке таблицы будет задано соответствие в виде записи, содержащей 3 поля суммарной длиной 124 байта.

Зачастую, помимо собственно полей, в состав записей включаются заголовки – отдельные поля, содержащие дополнительную служебную информацию:

- 1) размер записи (что особенно актуально в случае, если в запись входят поля переменной длины);
- 2) указатель на дополнительные данные, хранящиеся в другом месте, например предназначенные для хранения больших объектов атрибуты типа BLOB;
- 3) сведения о времени создания и модификации записи;
- 4) сведения о блокировках для обеспечения многопользовательского доступа.

В системном каталоге БД каждому типу записи сопоставляется схема этого типа. В ней описываются имена полей, типы данных, а также места расположения полей внутри записи (отступы от начала записи).

Блоки

Следующим уровнем абстракции данных выступает блок¹ – контейнер для записей. Обычно в структуру блока входят заголовок и какое-то количество записей, принадлежащих одному отношению. В заголовке блока содержится служебная информация:

- 1) идентификатор блока;
- 2) сведения о том, какое отношение обслуживается блоком;
- 3) таблица смещений (offset table), содержащая значения отступов от начала блока для каждой входящей в блок записи;
- 4) ссылки на один или несколько других блоков (в случае если блок входит в состав индекса);
- 5) информация о последнем обращении к блоку с целью модификации данных;
- 6) сведения о блокировках блока для обеспечения многопользовательского доступа.

Число вмещаемых записей определяется размерностью блока (обычно по умолчанию размер блока установлен в 4 Кб). Не исключена ситуация, когда размер записи превысит размер блока. Если превышение размера записи над размером блока незначительно, можно увеличить размер последнего, в противном случае СУБД придется расщеплять запись по нескольким блокам.

Процедура расщепления осуществляется автоматически. Для этих целей СУБД добавляет в заголовки записей дополнительные данные (номер фрагмента, адрес следующего фрагмента и т. п.).

Файл

Для группировки блоков создается очередной уровень абстракции – файл. В простейшем случае блоки одного отношения объединяются в отдельный файл, который вы сможете найти на диске компьютера. Однако у более продвинутых клиент-серверных систем гораздо чаще встречаются более сложные решения, в которых все объекты БД (домены, таблицы, индексы, представления, процедуры, триггеры и т. д.) объединяются в один составной файл. Как вы понимаете, структура такого файла целиком и полностью зависит от фантазии разработчика СУБД.

¹ В документации к ряду СУБД вместо термина «блок» часто применяют термин «страница» (page).

Модификация записей

Операция вставки нового кортежа в отношение приводит к появлению новой записи на диске сервера. В ситуации, когда записи хранятся в произвольном порядке, задача добавления новой записи сводится к нахождению пустого места в ближайшем блоке, сохранения в нем данных и добавлению обновленных сведений в заголовок блока (в первую очередь в таблицу смещений).

Если же записи хранятся в упорядоченном виде (такой подход используется в кластерных индексах, см. главу 9), то сложность задачи возрастает, особенно в ситуации, когда в требуемом блоке отсутствует свободное место. В последнем случае системе приходится осуществлять предварительные операции расщепления одного блока на два (и более) с целью высвобождения свободного пространства для очередной записи. Если поток новых записей слишком велик, то операции расщепления могут существенно снизить производительность системы. Поэтому многие разработчики ПО пошли на хитрость и научили СУБД заносить новые записи в блок переполнения (overflow block). Позднее, когда нагрузка на систему спадает (например, в ночное время), СУБД самостоятельно осуществляет все необходимые действия по переносу новых данных из блока переполнения в требуемые места обычных блоков.

Операция удаления кортежа из таблицы высвобождает в блоке пространство, ранее занимаемое соответствующей кортежу записью. Об этом делается соответствующая отметка в заголовке блока, кроме того, в месте размещения удаленной записи (возможно, даже в ее заголовке) ставят специальный признак, остроумно названный «обелиском» (tombstone) [15]. Обелиск будет находиться в этом месте до тех пор, пока пространство удаленной записи не займет новая запись.

Если в результате удаления большой порции записей и блок (или несколько блоков) пустеет, то СУБД принимает решение о слиянии блоков с пустующим пространством в один блок.

Операция редактирования записи не оказывает влияния на структуру хранения, т. к. измененные поля займут то же самое место.

Особенности представления объектов

В БД, исповедующих объектно-ориентированную концепцию (см. главу 23), принцип представления данных так или иначе схож с подходом, применяемым в реляционной модели, с той лишь разницей, что здесь

вместо кортежа с его атрибутами главным действующим лицом выступает объект с его полями.

Однако есть и принципиальные различия. Так (в отличие от реляционных кортежей, которые не содержат ничего, кроме атрибутов с данными), в распоряжении объектов имеются специальные функции, в терминах ООП называемые методами. Методы объектов призваны управлять состоянием их полей и/или воздействовать на другие объекты БД. Код методов не может (да и не должен) храниться в полях или записях, так как он принадлежит не отдельному объекту, а классу, на основе которого конструируется объект. Поэтому код методов сохраняется отдельно в схеме БД. Для того чтобы объект получил возможность вызывать свои методы в состав представляющей объект записи, приходится внедрять дополнительные поля. Как минимум это поле содержит сведения, какому классу принадлежит объект.

Еще одна особенность представления объектов заключается в необходимости их уникального именования. Для этой цели объекту присваивается идентификатор (object identity). В дальнейшем идентификатор рассматривается СУБД как адрес конкретного объекта и при необходимости учитывается при построении связей между объектами. Проблема заключается в том, что зачастую заранее неизвестно, сколько взаимосвязанных объектов будет создано, поэтому для описания связей приходится пользоваться записями переменной длины [15].

Журнальная информация

В любой СУБД, помимо пользовательских данных, хранится и обрабатывается большой объем служебной информации, в котором особое место занимает важнейшая составляющая базы данных – **журнал транзакций**.

В журнале транзакций система сохраняет все изменения, осуществляемые в базе данных. Журнализации подлежат:

- 1) идентификатор транзакции, в рамках которой осуществляется модификация данных;
- 2) дата и время старта транзакции;
- 3) объект(ы), подвергший(е)ся модификации (файл, номер блока, номер записи, поле);
- 4) новое и предыдущее состояния объекта;
- 5) дата и время завершения транзакции.

Собранная информация позволит в случае необходимости (из-за отказа или сбоя в работе) вернуть базу данных в более раннее согла-

сованное состояние. Таким образом, благодаря журналу транзакций появляется возможность восстановления данных в любой из точек учтенного в журнале временного интервала.

Формат журнала определяется разработчиками СУБД, в простейшем случае это может быть обычный последовательный файл с записями переменного размера. Файл должен поддерживать операции просмотра в обоих направлениях и храниться не менее надежно, чем основные данные.

Замечание

При осуществлении резервного копирования БД вместе с основными данными обязательно включайте в копию журнал транзакций.

Резюме

За кажущейся простотой представления данных на логическом уровне в двумерных реляционных таблицах на физическом уровне БД скрывается сложный механизм хранения данных на вторичных устройствах.

В соответствие атрибуту таблицы ставится элементарная структурная единица хранения – поле, большинство стандартных типов данных сохраняется в полях в виде байтовой последовательности фиксированного размера. Коротежу таблицы соответствует объединение полей – запись. В свою очередь, записи группируются в блоки, а блоки объединяются в файлы. Записи, блоки и файлы, помимо собственно данных, еще содержат служебную информацию, позволяющую обеспечить быстрый доступ к элементам хранения.

Помимо собственно данных, БД хранит много дополнительной информации: индексы таблиц, хранимые процедуры и триггеры, разнообразную статистику и журнал транзакций.

Вопросы для самопроверки

1. Каким образом реляционная таблица представляется на устройстве хранения?
2. Каким образом на вторичном устройстве размещаются атрибуты таблицы, описываемые стандартными типами данных постоянной длины?
3. Прокомментируйте особенности физического хранения текстовых данных.

4. Что такое запись?
5. Какая служебная информация содержится в заголовке записи?
6. Что такое блок?
7. Какая служебная информация содержится в заголовке блока?
8. Что происходит, когда размер записи превышает размер блока?
9. Какой объект используется в качестве контейнера для хранения блоков?
10. Какие особенности представления объектов на вторичных устройствах хранения вам известны?
11. Для чего предназначен журнал транзакций?

Глава 9

Индексирование

Представьте себе, что вам в руки попался справочник крупного города. В увесистом томе собрано несколько миллионов телефонных номеров жителей и организаций мегаполиса с их фамилиями, почтовыми и электронными адресами, названиями фирм и многой другой полезной информацией. Воистину бесценная находка, если бы не одна маленькая загвоздка – строки в справочнике не упорядочены по алфавиту, а напечатаны в произвольном порядке. А ну-ка попробуйте воспользоваться таким «подарком» и найдите в нем необходимый телефонный номер! Полагаю, что энтузиазм даже самого большого оптимиста улетучится после бесплодного пролистывания первого десятка страниц, ведь столь нужные нам сведения с равной вероятностью могут оказаться на любой из нескольких сотен страниц. Парадоксальная ситуация – мы обладаем требуемыми данными, но не в силах ими воспользоваться. Ценность столь полезной информации свелась к нулю из-за ее неудобного представления. Несложно предугадать судьбу этого справочника – он окажется в мусорном ведре.

Электронная база данных и бумажный справочник преследуют единую цель – обеспечить хранение и поиск данных. Общая цель определяет схожесть принципов хранения и объясняет родство недостатков. И рукописная книга, и электронная реляционная таблица не предусматривают автоматического упорядочивания строк. И там, и там строки записываются в той последовательности, в которой были введены. Подобный подход нисколько не противоречит требованиям к реляционным таблицам, согласно задумке разработчиков реляционной модели, данные в них кортежи могут содержаться в любом порядке. В базах данных можно попытаться упорядочивать данные в момент их ввода, например оставляя в запасе пустые строки и заполняя их позднее. Эту идею реализовать несложно, но что делать, если возникнет необходимость изменить порядок следования строк? Сегодня нас устраивает тот

факт, что телефонный справочник отсортирован по фамилии владельца, но завтра мы захотим упорядочить данные по номерам телефонов, а послезавтра – по названиям улиц и номерам домов. Может показаться, что в такой ситуации единственным решением задач поиска и сортировки записей стало бы полное сканирование всех неупорядоченных строк отношения. Компьютеру (в отличие от человека) никогда не надоеет перебирать записи, и он когда-нибудь обязательно доберется до строки с интересующим нас телефонным номером. Но на операции сортировки и поиска будут постоянно затрачиваться существенные вычислительные ресурсы. Какой выход из сложившегося положения? Ответ таков – внедрить в СУБД механизм индексирования данных.

Основная задача индекса – обеспечение быстрого доступа к данным по некоторому коду. В любой бумажной энциклопедии, справочнике, словаре эта задача частично решена. В перечисленных категориях книг информация упорядочена по алфавиту, а в качестве кода (индекса) в простейшем случае выступает первая буква искомого слова. При поиске требуемого слова (термина, названия и т. п.) мы почти сразу попадаем на требуемую страницу, открыв книгу на нужной букве.

В отличие от бумажной книги, реляционная таблица может одновременно обладать несколькими индексами:

- индексом первичного ключа;
- несколькими индексами, обслуживающими внешние ключи;
- произвольным числом пользовательских индексов.

Индекс первичного ключа создается автоматически в момент назначения первичного ключа таблицы. Перед ним стоит одна-единственная задача – первичный индекс должен гарантировать, что во входящих в состав первичного ключа столбцах будут храниться уникальные данные.

Индексы внешнего ключа также создаются автоматически в момент назначения внешнего ключа. Основная задача индекса внешнего ключа – быстрый поиск записей во внешних таблицах для соединения с этими таблицами.

Пользовательские индексы создаются руками программиста. Они описывают дополнительные способы упорядочивания строк таблицы, ускоряют поиск данных и могут проверять данные на уникальность. Пользовательские индексы определяют только логический порядок следования строк в таблице, никак ни влияя на их физическое расположение. Число пользовательских индексов таблицы ограничено лишь здравым смыслом проектировщика и ограничениями СУБД.

Замечание

Основное назначение любого индекса состоит в обеспечении эффективного прямого доступа к кортежу отношения по его коду.

Быстрый доступ к данным – это главный козырь индекса, благодаря нему в БД реализуется несколько важных сервисных возможностей:

- упорядочивание строк отношений;
- поиск данных по полному или даже частичному совпадению;
- соединение таблиц, связанных по первичному и внешнему ключам;
- возможность поддержки уникальности данных за счет защиты от ввода повторяющихся значений.

За любую услугу приходится платить. Для получения существенно-го выигрыша в скорости доступа к данным в БД приходится создавать дополнительные структуры. В этих структурах хранятся код данных и указатель на местоположение этих данных, и так для каждой строки проиндексированной таблицы. Чем больше индексов, тем больше размерность дополнительных структур. Может случиться так, что суммарный размер служебной информации, отводимой для хранения индексов, превысит размер полезных данных. Поэтому умение создавать хорошие индексы основано на интуитивной способности профессионального программиста балансировать между двумя противоречиями: поддержанием высокой скорости обработки данных, с одной стороны, и затратами на хранение индексов в памяти – с другой.

Получив общее представление о назначении индексов, перейдем к рассмотрению способов индексирования данных в современных СУБД.

На сегодня существует несколько методов построения индексов. Мы рассмотрим наиболее популярные:

- индексы, базирующиеся на технологии хеширования;
- индексы на основе сбалансированных В-деревьев;
- битовые индексы.

Индексы на основе хеширования

Хеширование (hashing) полезно в том случае, когда требуется большой объем значений сохранить в сравнительно небольшой по размерности таблице (точнее, хеш-таблице) и обеспечить быстрый доступ к этим значениям.

Существует множество методов хеширования данных. Их общей идеей является применение к данным некоторой функции свертки (хеш-функции), по определенному алгоритму вырабатывающей значение меньшего размера (хеш-значение). Например, на вход функции поступает текст (допустим, название компании), а на выходе мы получаем какое-то число. Свертка исходных данных – однонаправленный процесс, получив хеш-значение, мы не сможем восстановить по нему исходные данные, но при построении индексов в этом и нет необходимости. Главное, чтобы хеш-функция на одни и те же входные данные всегда реагировала одним и тем же выходным значением.

Для демонстрации алгоритма хеширования рассмотрим процесс индексирования столбца SUPPLIER таблицы поставщиков из учебной базы данных склада (рис. 9.1).

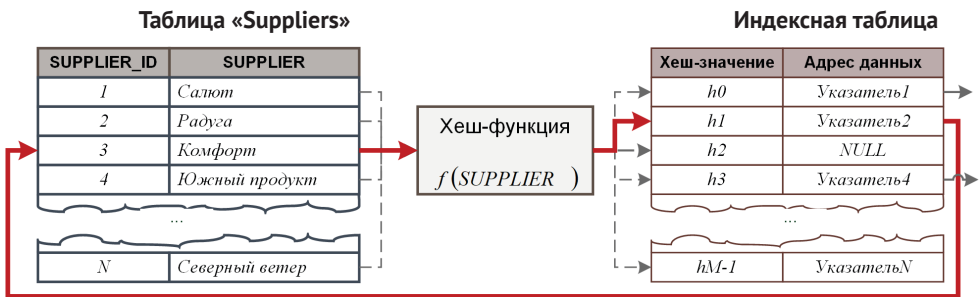


Рис. 9.1. Совместная работа таблицы с данными и таблицы с индексами

Для хранения индексных данных СУБД создает отдельную хеш-таблицу, состоящую из двух столбцов: хеш-значения и адресного поля, содержащего указатель на строку индексируемой таблицы «Suppliers». Только что созданная хеш-таблица не пуста и уже содержит какое-то число строк. Размерность таблицы определяется настройками СУБД и особенностями хеш-функции. Изначально адресное поле не заполнено и хранит неопределенный указатель NULL. Система начинает процесс индексирования – последовательно читает названия поставщиков и передает их на вход хеш-функции. Хеш-функция в соответствии с заложенным в нее математическим выражением осуществляет свертку исходных текстовых данных в числовое значение. Допустим, что хеширование текстовой строки «Комфорт» дало хеш-значение h_1 . Переходим к строке с порядковым номером h_1 в индексной таблице (хеш-таблице) и в ее столбец «Адрес данных» вносим указатель на физическую запись «Комфорт» индексируемой таблицы. Эта операция повторяется до тех

пор, пока не построится индекс для всей таблицы «Suppliers». По завершении индексирования в хеш-таблице мы обнаружим пары значений: хеш-значение и соответствующий этому значению указатель на строку с реальными данными таблицы.

Благодаря индексам существенно упрощается поиск данных, теперь вместо утомительного перебора всех строк исходной таблицы с целью нахождения совпадения система передаст содержание интересующей нас строки хеш-функции. Хеш-функция вычисляет хеш-значение и возвращает номер строки в индексной хеш-таблице. После этого нам остается перейти к искомой строке по хранящемуся в индексной таблице указателю.

После создания индекса СУБД обязана поддерживать его в актуальном состоянии. Это означает, что при вводе в базовую таблицу нового кортежа, редактировании и удалении строки СУБД индексирует изменения и обновляет данные в индексной таблице. Объем вычислительных ресурсов, затрачиваемых на обновление индексов, прямо пропорционален количеству модифицированных записей и в отдельных случаях может существенно снизить производительность БД.

В процессе построения индекса на основе технологии хеширования ключевая роль отводится хеш-функции, отвечающей за свертку исходных данных в хеш-значение заведомо меньшей размерности. Для того чтобы это происходило с наилучшим качеством, функция хеширования должна обладать следующими чертами:

- высокая скорость вычисления хеш-значения;
- умение обрабатывать входные данные любого типа;
- низкая вероятность совпадения хеш-значений для разных входных данных, говоря точнее, для хеш-таблицы размером в M строк эта вероятность в идеале должна быть ниже, чем $1/M$;
- функция хеширования должна быть детерминистической, т. е. для одного и того же значения всегда давать одно и то же хеш-значение.

К настоящему времени разработано достаточно много алгоритмов генерации хеш-значения. В России в соответствии с ГОСТ Р 34.11–2012 рекомендована к использованию функция хеширования «Стрибог». Исходный код функции написан на языке Си и находится в свободном доступе.

В природе идеальных функций хеширования не бывает, кроме того, размер хеш-таблицы существенно меньше исходной таблицы, по-

этому всегда имеется вероятность того, что в результате хеширования два (или более) различных входных значения свернутся в одно и то же хеш-значение и, как следствие, будут направлены в одну и ту же строку хеш-таблицы. Но строка хеш-таблицы обладает одним-единственным полем указателя, поэтому одновременно не способна ссылаться на несколько записей исходной таблицы.

На языке разработчиков ПО свертка различных исходных данных в одно и то же хеш-значение называется коллизией. Появление коллизий неприятно, но не фатально. На сегодня программистами разработано большое количество противоядий. Одно из наиболее распространенных – разрешение коллизий с помощью цепочек. Это тот случай, когда предусматривается возможность прикрепления к каждой из ячеек хеш-таблицы дополнительной структуры, например связанного списка. Как только в результате коллизии в уже занятую ячейку «неожиданно» попадает ссылка на еще одну строку, механизм построения индекса не впадает в панику, а создает дополнительный элемент связанного списка (который состоит из двух полей – поля данных и поля связи) и заносит указатель на строку с исходными данными уже не в ячейку таблицы, а во вновь созданный элемент списка (рис. 9.2). В результате выстраивается цепочка из элементов списка, а в ячейке хеш-таблицы теперь хранится указатель не на строку исходной реляционной таблицы, а на первый элемент списка.

Индексная таблица

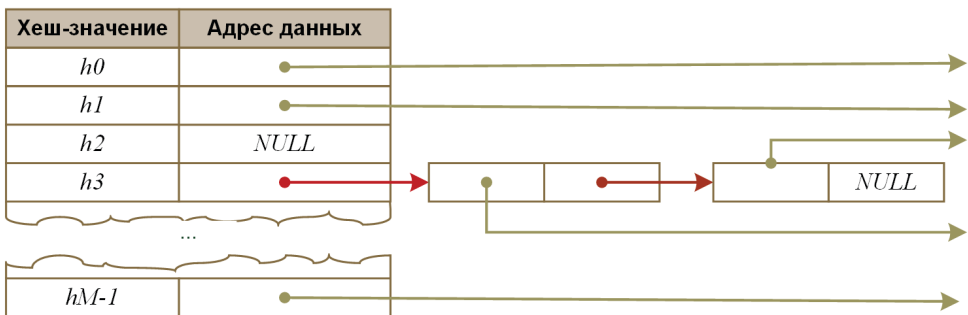


Рис. 9.2. Устранение коллизий с помощью цепочек

Наличие цепочки несколько замедляет процесс поиска данных, ведь теперь нам приходится просматривать все элементы связанного списка, но это все равно лучше, чем перебирать все записи неиндексированной таблицы.

Замечание

Надо отметить, что любая перестройка индексной таблицы – это микростресс для СУБД, ведь изменение размера хеш-таблицы заканчивается перерасчетом всех хеш-значений и перестановкой ее элементов.

Индексы на основе В-деревьев

Деревья очень удобны для описания различного рода иерархических структур, подобных структуре каталогов на диске вашего компьютера, организационной структуре предприятия, генеалогического древа и весьма эффективны при решении задачи индексирования таблиц.

Не вдаваясь в тонкости всех изобретенных программистами разновидностей электронных деревьев, сосредоточим наше внимание на В-деревьях. В-дерево содержит узлы двух типов: листья (не имеющие потомков узлы самого нижнего уровня) и страницы (имеющие потомков узлы верхних уровней). В-дерево должно обладать следующими качествами:

- сбалансированностью. Другими словами, расстояние от корня дерева до любого его листа (узла, не имеющего ссылок на другие элементы) должно быть одинаковым;
- ветвистостью. Любой узел дерева должен обладать возможностью ссылаться на большое число узлов-потомков.

После краткого экскурса по азам «ботаники» сместим акцент беседы ближе к области применения В-деревьев в СУБД. Для этого воспользуемся подопытной таблицей поставщиков из демонстрационной БД и построим возрастающий уникальный индекс по полю названий компаний-поставщиков SUPPLIER.

В момент появления на свет В-дерево представляет собой скромный «росток» со всего одной корневой страницей и одним-двумя листьями (рис. 9.3). В нашем примере у всех узлов дерева имеется по 5 пар полей данные–указатель, но на практике их может быть и больше.

По нашей команде система индексирования начинает последовательно считывать строки из таблицы SUPPLIER. На первом шаге в поле указателя первого элемента листа передается ссылка на первую строку таблицы (запись «Салют»). На втором шаге в лист попадает ссылка на запись «Радуга». Так как в алфавитном порядке буква «Р» предшествует «С», то указатель «Салют» смещается на вторую позицию, а первая позиция листа В-дерева передается указателю «Радуга». На пятом шаге построения

индекса лист заполняется полностью, поэтому после считывания из таблицы SUPPLIER шестой по счету записи «Инфобизнес» происходит весьма неординарное событие – создается очередной лист (рис. 9.4).

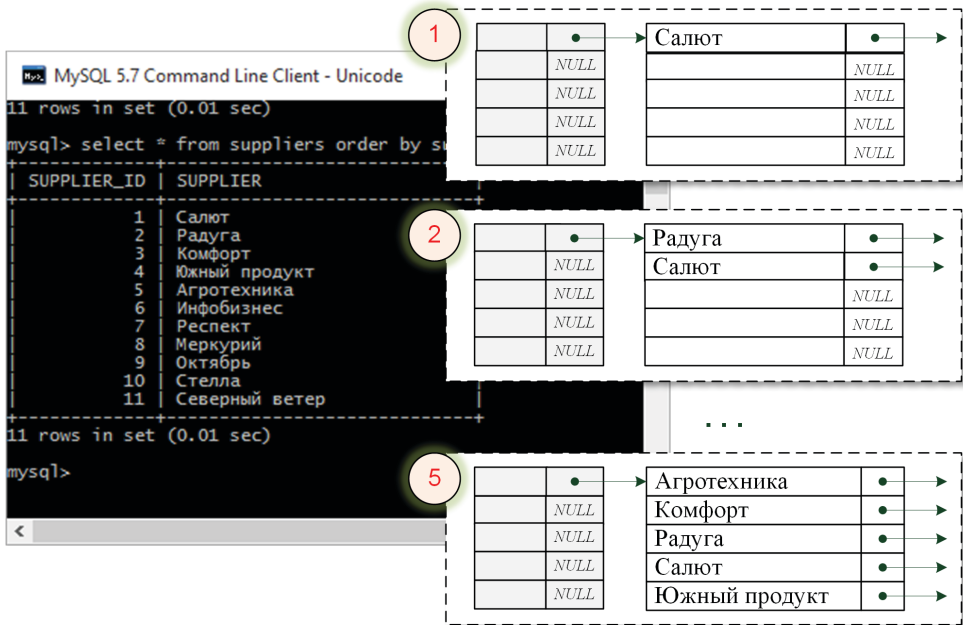


Рис. 9.3. Первые шаги заполнения В-дерева

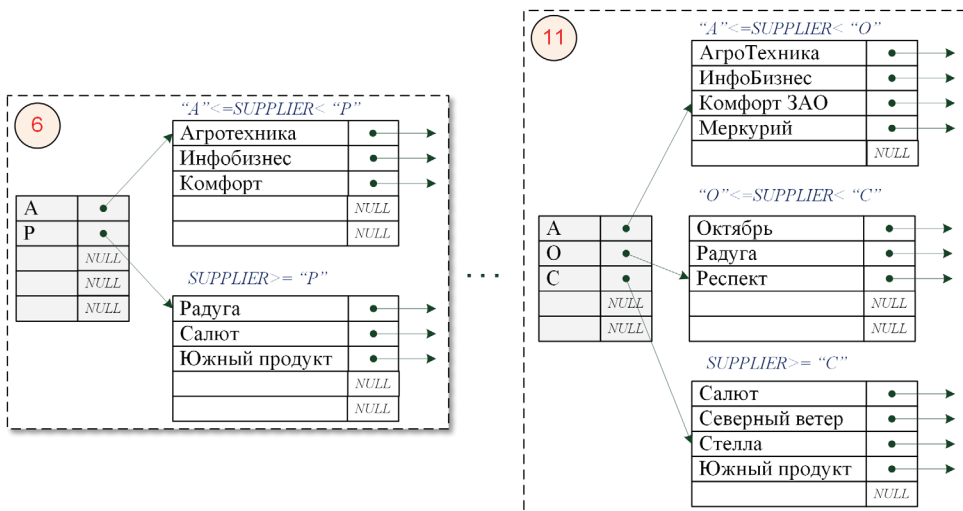


Рис. 9.4. Расщепление узлов В-дерева

После появления нового, 6-го по счету узла данные перераспределяются между листьями поровну. В первый лист попадут ссылки на три первые записи, во второй – на три заключительные. Корневая страница также подвергается глубокой модернизации, теперь в ней будут храниться указатели на вновь созданные дочерние листы и индекс этих листов. Суть индекса точно такая же, как и в любом многотомном словаре или энциклопедии, – индекс разъясняет нам, с какой буквы алфавита начинаются слова в конкретном томе энциклопедии. В примере в первом дочернем листе содержатся ссылки на поставщиков с именами от «А» до «П» включительно, а во второй от «Р» и далее. После расщепления процесс продолжается до тех пор, пока не заполнится любой из листов, это случится на 11-м шаге. В этом случае добавляется еще один лист, и перераспределяем в него часть данных из первых двух листов. Появление нового листа вынуждает сервер перестроить корневую страницу – теперь в ней хранится три указателя на листы и три индекса: «А», «О» и «С».

Вскоре в дереве появится 4-й, а затем 5-й лист. Но после того, как в индекс попадут первые 25 строк из таблицы, все листья дерева окажутся полностью заполненными. Теперь, прежде чем создать шестой лист, системе придется решить более сложную задачу. Корневая страница способна управлять только пятью дочерними узлами и не способна удержать шестой, поэтому сервер между корнем и листьями создаст новый уровень иерархии. На этом уровне размещаются две новые страницы, и указатели на них передаются в корневой лист. В свою очередь, под управление вновь созданным страницам передаются все листья дерева.

Процесс расщепления и создания новых листов продолжится до тех пор, пока все данные не попадут в индекс. Во время индексирования будут появляться все новые и новые страницы и листы, а дерево станет представлять многоуровневую сбалансированную иерархическую структуру. Страницы верхних уровней В-дерева станут выступать в роли индекса, а листья самого нижнего уровня станут хранить данные и указатели на записи в таблице (рис. 9.5).

Как только возникает задача поиска какой-нибудь записи, мы считываем индекс корневой страницы и по указателю переходим к нужной дочерней странице, затем читаем индекс у дочерней страницы и переходим к следующему узлу. Процесс продолжается до тех пор, пока мы не доберемся к требуемому листу с данными. Благодаря тому что В-дерево сбалансировано, поиск в индексе осуществляется за одинаковое количество переходов по индексным страницам, вне зависимости

от того, какие данные мы ищем. Такая особенность В-деревьев весьма благоприятно сказывается на производительности СУБД.

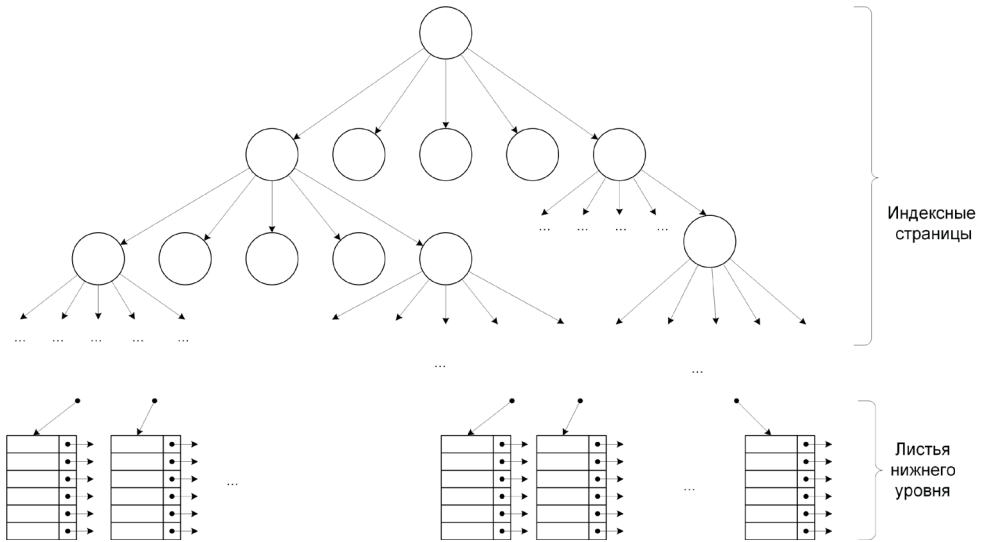


Рис. 9.5. Фрагмент индекса на основе В-дерева

Индексы на основе В-деревьев не лишены недостатков. Один из основных – большие затраты на перестроение индекса, в особенности если было изменено, добавлено или удалено большое количество записей. Это приводит к расщеплению большого количества переполненных страниц или, наоборот, сложению пустых или полупустых страниц и листьев.

Замечание

В-деревья являются основным средством построения индексов в больших многопользовательских СУБД, например таких, как SQL Server компании Microsoft, InterBase, Oracle и MySQL. В случае если индекс кластерный, то даже физическое хранение реляционной таблицы осуществляется в виде В-дерева, тогда строки данных реально хранятся в листьях нижнего уровня дерева.

Битовые индексы

Основное назначение **битовых индексов** (bitmap indexes) – индексирование столбцов с небольшим количеством различных значений. Классический пример – пол человека, с точки зрения БД его можно задать тремя значениями: «М», «Ж» и NULL. В подобных индексах для

представления существования определенного значения столбца целесообразно использовать битовую последовательность (строку из нулей и единиц).

Если индекс на основе В-деревьев хранит однозначное соответствие между строкой таблицы и записью в индексе, то битовый индекс обеспечивает ссылку на большое количество строк одновременно.

Индексы на основе битовых карт занимают малый размер (он состоит в прямой пропорции от кардинальности индексируемых данных) и поэтому обеспечивают гораздо более высокую скорость выборки.

Правила назначения пользовательских индексов

Настал черед обсудить особенности индексирования таблиц. Так как столбцы первичного и внешнего ключей индексируются автоматически, речь пойдет исключительно о порядке назначения пользовательских индексов. Принудительная дополнительная индексация ключевых полей может понадобиться только в том случае, когда они входят в перечень столбцов составного индекса.

В первую очередь следует индексировать столбцы отношений, которые наиболее часто используются в запросах на выборку данных. Последовательность столбцов в составном индексе должна соответствовать последовательности столбцов, следующих в инструкции `SELECT` сразу после директивы `ORDER BY` или `GROUP BY`. При этом далеко не обязательно плодить отдельные индексы для каждого варианта запроса, так как оптимизатор запросов перед выполнением инструкции `SQL` постарается подобрать наиболее подходящий индекс.

Наиболее эффективны индексы, накладываемые на столбцы, в которых не хранятся повторяющиеся значения.

Можно смело индексировать уже полностью заполненные справочные таблицы, этот же совет в равной степени подходит ко всем таблицам, содержимое которых или не изменяется вовсе, или слабо подвержено изменениям. В качестве примера приведем справочник поставщиков или таблицу-классификатор товаров из демонстрационной базы данных к этой книге.

Никогда не создавайте индекс только на основе предположения, что он когда-нибудь пригодится. Более того, не индексируйте все столбцы подряд, будет ли использоваться такой индекс, неизвестно, но то, что вы будете хранить еще одну копию таблицы, это точно.

Кроме того, в подавляющем большинстве случаев не стоит индексировать:

- столбцы большого размера (например, большие текстовые столбцы и столбцы, предназначенные для хранения бинарных объектов BLOB);
- столбцы, допускающие неопределенные значения NULL;
- столбцы с небольшим количеством многократно повторяющихся значений;
- столбцы с часто изменяющимися значениями.

Замечание

Основным средством управления индексами являются инструкции структурированного языка запросов: `CREATE INDEX`, `ALTER INDEX` и `DROP INDEX`. Эти и другие команды мы рассмотрим в главах, посвященных изучению SQL.

Внимание!

Если вы администрируете большую, активно эксплуатируемую многопользовательскую базу данных, то периодически стоит заставлять СУБД перестроить все индексы. Подобные операции следует проводить в период наименьшего использования БД (в выходные и праздничные дни или в ночное время), предварительно не забывая создать полную копию БД и журналов транзакций.

Избирательность индекса

Под избирательностью (селективностью) индекса понимается число кортежей, которые могут быть выбраны по каждому значению индекса во время поиска. Более избирательный индекс учитывает больше значений и отбирает меньший объем данных одним своим значением, и наоборот – индекс с низкой избирательной способностью (имеющий всего несколько значений в большой таблице) может оказаться бесполезным в запросах.

Формула расчета селективности S выглядит так:

$$S = \frac{1}{\text{число ключей} - \text{число повторяющихся ключей}}. \quad (1)$$

Именно по этой формуле работают оптимизаторы запросов в InterBase и FireBird [35]. Чем меньше результат, тем лучше с точки зрения оптимизатора. Например, в случае с уникальным индексом избирательность

стремится к 0. Здесь каждому значению индекса соответствует единственная запись в таблице, а повторяющихся ключей просто нет. Если же индекс не уникален и допускает хранение значений-дубликатов, то избирательность падает. При самом плохом развитии ситуации (когда все значения ключей одинаковы) избирательность стремится к 1.

Можно применять и альтернативную формулу расчета селективности:

$$S = \frac{\text{число экземпляров значения индекса для таблицы}}{\text{число ключей} - \text{число повторяющихся ключей}}. \quad (2)$$

Выражение (2) не противоречит (1), просто в данном случае селективность оценивается обратными значениями (1 соответствует лучшей избирательности, а стремящееся к 0 значение – худшей).

Замечание

Примерами индексов, обладающих наибольшей избирательностью, могут стать индекс первичного ключа и уникальный индекс.

Показатель избирательности индекса в первую очередь важен для оценки быстродействия индекса. Чем выше избирательность, тем производительнее индекс. И наоборот. Почему так происходит?

Мы говорили о том, что в большинстве СУБД индекс основан на структуре В-дерева, основным достоинством которого считается сбалансированность. Благодаря сбалансированности мы, находясь в корне дерева, достигнем любого из листьев нижнего уровня за одно и то же количество шагов и быстро обнаружим искомый ключ. Но только при одном условии – если индекс уникален. А если нет, то на самом нижнем уровне дерева появляются цепочки дубликатов (рис. 9.6).

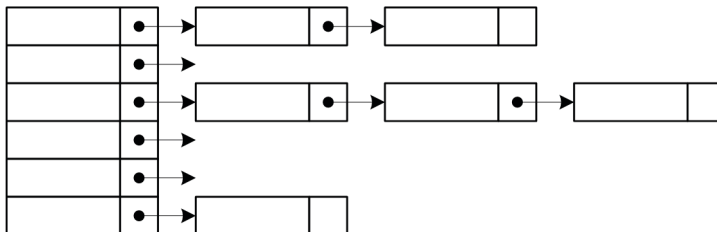


Рис. 9.6. Лист нижнего уровня с цепочками дубликатов

Каждая цепочка хранит только одинаковые элементы-дубликаты. С появлением очередного дубликата он вставляется в начало цепочки, а остальные элементы смещаются. Таким образом, чем больше в таблице

повторяющихся значений, тем длиннее их цепочка. Теперь при поиске нужного значения индекс будет вынужден затрачивать дополнительные усилия на просмотр соответствующей вереницы элементов.

Резюме

Современные СУБД позволяют создавать индексы трех типов: первичный, кластерный и вторичный. Основной структурой, используемой программистами для построения индексов, выступает сбалансированное В-дерево, но это не единственное решение. Существуют индексы на основе хеширования, битовых карт и R-деревьев.

При работе с индексами разработчику БД всегда приходится балансировать между двумя противоречиями: если индексов слишком много, то они занимают много места и снижают производительность СУБД во время перестроения индексов, а если индексов недостаточно, падает скорость выполнения запросов.

К тематике индексов мы еще возвратимся в главах, посвященных структурированному языку запросов.

Вопросы для самопроверки

1. Для чего необходимо индексирование таблиц?
2. Какие виды индексов вам известны?
3. Какие индексы в БД создаются автоматически?
4. Опишите принцип работы индекса на основе хеширования.
5. Какие требования предъявляются к хеш-функциям?
6. Что такое «коллизия», и как она устраняется в индексах на основе хеширования?
7. Опишите принцип построения индекса на основе В-дерева.
8. В каких случаях следует применять битовые индексы?
9. Какие правила следует соблюдать при создании пользовательского индекса?
10. Что такое избирательность индекса?

Глава 10

Безопасность данных

Наш мир построен так, что в нем недостаточно просто посадить дерево, спроектировать дом или разработать компьютерную программу. Никто не даст вам гарантии, что завтра дерево не срубят, дом не снесут, а программу не взломают. Окружающий нас мир не всегда доброжелателен, поэтому надо уметь защищать творение своих рук, а в идеале – научить свой продукт защищаться самому.

Актуальность проблемы защиты компьютерной информации вряд ли вызывает сомнение. Мы живем в такое время, когда практически ежедневно приходят новости о том или ином киберпреступлении. Утрата или хищение данных из БД особенно болезненна, так как одним махом затрагивает тысячи, а иногда и миллионы людей. Стоимость информации, которую эти люди доверили разработанной вами программе, зачастую имеет не просто денежное измерение. Задумайтесь, сколько будет стоить репутация программиста, в случае если спроектированная им БД рухнет от первой же атаки.

Под защитой данных понимается система мер, направленная на предотвращение доступа к данным несанкционированных пользователей, с целью их чтения, изменения или разрушения. Вместе с тем лицу, отвечающему за безопасность БД, не следует забывать, что даже санкционированный пользователь без всякого на то злого умысла способен принести массу бед.

Определение

Защищенная база данных – это БД, которая обеспечивает конфиденциальность, доступность и целостность данных пользователя.

Если говорить строго, то безопасность данных – это состояние защищенности, при котором обеспечиваются конфиденциальность, доступность и целостность данных. Каждая из составляющих безопасности

несет персональную ответственность за свой участок обороны [25, 34, 41, 51, 52]:

- **конфиденциальность** отвечает за обеспечение доступа к данным, только санкционированным пользователями;
- **целостность** исключает несанкционированное изменение структуры и содержания данных;
- **доступность** позволяет обеспечить доступ к данным, санкционированным пользователями, по их первому требованию.

Конфиденциальность направлена на сохранение в тайне данных, критичных для организации, и на обеспечение неприкосновенности личных данных. Прорвать брешь в защите конфиденциальных данных заинтересованы очень многие, список любопытных непрошенных гостей определяется важностью информации, находящейся в БД. Здесь и представители иностранных разведок, организации, занимающиеся промышленным шпионажем, конкурирующие предприятия, различные спецслужбы, спамеры, хакеры, преступные элементы и обычные частные лица, желающие покопаться в ваших данных просто из спортивного интереса.

Потеря целостности грозит искажением или даже разрушением хранимых в БД данных. И то, и другое как минимум приводит к остановке работы БД на период восстановления данных. Вынужденный простой компании – это не только денежные потери от незаключенных сделок, но и удар по престижу. Иногда потери от искажения данных приносят больше вреда, чем их частичное или даже полное разрушение. Это происходит тогда, когда персонал вовремя не заметил факта искажения хранимой в БД информации и, например, продал пассажиру билеты на несуществующий поезд.

Утрата доступности приводит к тому, что пользователь лишается возможности обращаться к хранимым в системе данным. Последствия от такого происшествия могут быть не просто печальными, но и самыми трагическими, в особенности если речь идет об информационной системе диспетчера аэропорта, системе управления вооружением, программном обеспечении опасного производства и т. п.

Конфиденциальность, целостность и доступность составляют минимальный набор требований к безопасности данных. Специалисты по защите информации могут дополнить его еще рядом пожеланий, например: аутентичность (подлинность данных), апеллируемость (подтверждение авторства), достоверность и т. д.

Откуда исходят угрозы?

Как бы это ни было обидно, но с точки зрения защиты информации наша база данных всегда будет выступать в незавидной роли жертвы, причем за добычей зачастую следит не один, а целая стая хищников. Посему нападения приходится ожидать в любой момент времени и с любой стороны. Как видите, игра с самого начала неравная. Дабы сгустить краски, отметим еще одну маленькую деталь «охоты». Жертва всегда обязана играть по правилам (правовым, моральным, этическим), а нападающий соблюдать их обычно не намерен. В такой игре мы никогда не победим, слишком не равны силы противоборствующих сторон и условия сражения. Поэтому целью защиты баз данных может выступать только одно – минимизация потерь от неминуемого нападения. Но откуда ждать нападения?

Разобравшись с базовыми требованиями к безопасности данных, сместим акцент нашей беседы к вопросу выявления источников угроз для защищаемой нами БД. Существует множество вариантов классификации угроз [41, 51] информационной системе: по природе возникновения, по источнику угроз, по способу доступа к защищаемому ресурсу, по степени воздействия на систему, по расположению источника и т. д. В контексте защиты БД нас устроит вариант классификации по способу осуществления угроз.

Явные угрозы:

- некорректная реализация механизма защиты;
- некорректная настройка механизма защиты;
- неполнота покрытия каналов доступа к информации средствами защиты.

Скрытые угрозы:

- нерегламентированные действия пользователя;
- ошибки и закладки в программном обеспечении.

На рис. 10.1 представлена типичная многопользовательская компьютерная система небольшой компании. Пользователи компании объединены в локальную вычислительную сеть, сеть включает в себя сервер БД и предоставляет возможность выхода в глобальную паутину интернет. Подобные системы развернуты в десятках тысяч офисов, отделов, служб. И практически везде вы без труда найдете все перечисленные выше угрозы.

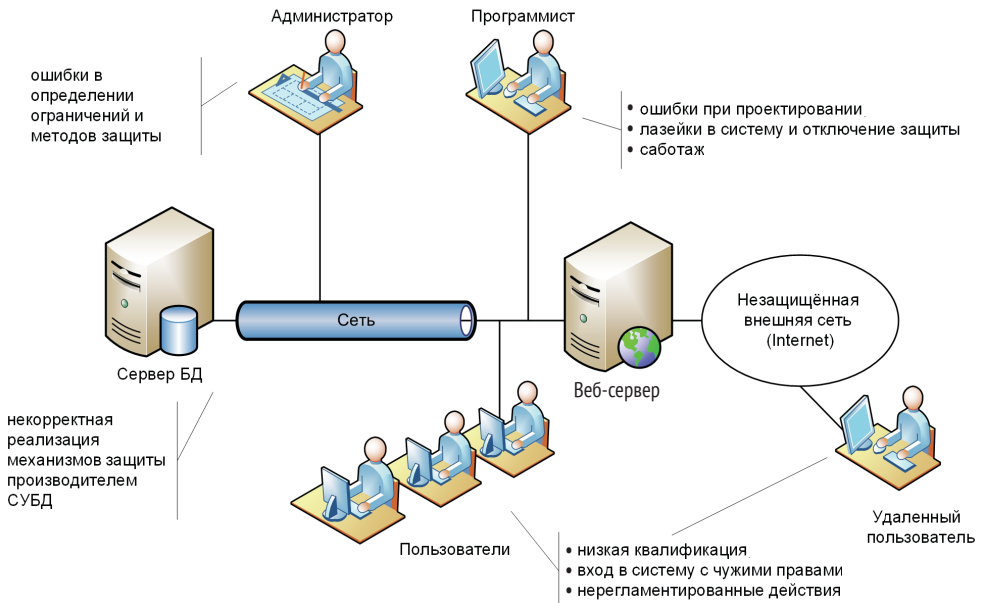


Рис. 10.1. Типичная многопользовательская компьютерная система

Сразу обратим внимание на ведущего специалиста информационной системы – администратора (в нашем примере это лицо собирательное и объединяет в себе администратора сети, администратора данных и администратора БД). Администратор отвечает за защиту компьютерной информации, пытаясь противодействовать всем явным угрозам. Но он сам выступает в качестве источника угроз, ведь все его промахи в настройке механизма защиты компьютерной системы снижают уровень безопасности.

Программист, даже самый подготовленный, также является источником скрытых угроз для системы, что уж говорить о программисте среднего уровня. Любой человек склонен ошибаться, и программист не является исключением из правил. Ошибки в исходном коде программ – это еще полбеда, в основном они вылавливаются во время тестирования и в начальный период эксплуатации. В период проектирования разработчик зачастую вынужден идти на сознательное нарушение правил разработки безопасного ПО и отключать или обходить элементы системы защиты. Тем более такие действия продиктованы объективными причинами. Представьте себе, что вы пишете какое-то клиентское приложение для работы с БД и каждый раз во время запуска программы она требует ввода имени и пароля пользователя для регистрации

на сервере. Окно, требующее ввести логин, выскакивает по сто раз на день на протяжении всего периода разработки. В подобных случаях даже самый правильный программист найдет способ обойти утомительную процедуру авторизации. Но ошибка не в создании лазейки, а в другом – программист не должен забыть удалить эту лазейку в систему из кода передаваемой в эксплуатацию окончательной версии приложения. К сожалению, это случается далеко не всегда. Если же программист создает программные закладки сознательно, с целью навредить компании, то выявить их крайне сложно.

В любой компьютерной системе основным действующим лицом является не администратор или программист, а обычный пользователь. Если пользователь обладает высокой квалификацией, то он становится первым помощником разработчика БД. Если высокая квалификация сочетается с не менее высокой ответственностью, то пользователь не станет совершать нерегламентированные действия: устанавливать на своем рабочем компьютере постороннее ПО, посещать не связанные с профессиональными обязанностями сайты, получать почту со спамом и вирусами, не забудет заблокировать компьютер во время обеденного перерыва. Одним словом, подготовленный пользователь постарается не причинить ущерб своей компании, ведь в ней он зарабатывает на жизнь. Другим полюсом магнита выступает обладающий правами доступа к БД низкоквалифицированный оператор. Даже самый отъявленный хакер вряд ли сможет нанести столько урона компьютерной системе, сколько он. Среди остроумных законов Мерфи для программистов есть один, полностью раскрывающий суть этой проблемы: «Невозможно создать программу с полной защитой от дураков, ибо они крайне изобретательны».

На СУБД воздействуют и весьма неординарными методами. Одна из специфичных для СУБД угроз называется SQL-инъекцией (SQL injection). Подобному виду атаки могут подвергаться СУБД, поддерживающие работу различных интернет-проектов, в первую очередь связанных с электронной коммерцией. Идея инъекции заключается во внедрении в текст запроса вредоносного кода. Если СУБД (а точнее, СУБД и администрирующий эту СУБД персонал) не в состоянии противодействовать SQL-инъекциям, то злоумышленник относительно легко сможет направить базе данных запрос любого содержания (от чтения и правки данных до удаления объектов БД).

Еще один пример нападения – загрузка СУБД бесполезной работой. Например, даже обладающий незначительными правами доступа к БД злоумышленник может сформировать ресурсоемкий запрос (допустим,

декартово произведение двух таблиц) и отправить его на вывод. Произведение двух таблиц размерностью N и M соответственно возвратит на выход результирующее отношение $N \times M$, что существенно нагрузит систему. А что произойдет, если сформировать несколько десятков подобных запросов? Как минимум пользователи столкнутся с существенным замедлением работы БД.

Третий пример характерных для СУБД угроз. Используя особенности обработки параллельных транзакций, злоумышленник может инициировать выполнение долгосрочной транзакции, которая заблокирует существенный блок данных (возможно, целую таблицу), что запретит обновление данных в таблице со стороны добропорядочных пользователей.

Мы перечислили далеко не все опасности, ко всему прочему добавьте кражу данных; разрушение данных в результате физического повреждения оборудования, стихийных бедствий, диверсий, заражения вирусами; отказ в доступе в результате обрывов соединительных кабелей, перебоев электропитания, сетевых атак.

Политика безопасности

К сожалению, угроз для информационной системы предприятия в целом и для БД, выступающей составным элементом ИС, в частности больше чем достаточно. Как снизить остроту проблемы и по возможности парировать угрозы?

Для сторонников системного подхода к решению проблем все начинается с определения политики безопасности предприятия, в которой формулируются общие принципы и конкретные правила работы с информационными ресурсами [41, 51, 52].

Определение

Политика безопасности – это совокупность норм и правил, определяющих принятые на предприятии (организации) меры по обеспечению безопасности обрабатываемой информации.

Политика информационной безопасности подлежит обязательному документальному оформлению. С этой целью должен быть сформирован пакет документов для всех уровней управления предприятия, а в части, касающейся исполнения должностных обязанностей, создаются соответствующие выписки из документов и доводятся до имеющих отношение к обработке данных подчиненных. Очень важно добиться

того, чтобы политика безопасности стала понятной всем сотрудникам предприятия, только после этого можно рассчитывать на требуемый результат.

Наша книга посвящена базам данных, поэтому, говоря об информационной безопасности, мы сместим акценты на правила защиты БД. Однако важно понимать, что политика безопасности должна охватывать весь информационный ресурс предприятия, а не замыкаться исключительно на БД.

Правила защиты БД

База данных как составной элемент информационной системы предприятия нуждается в комплексной системе защиты. В набор средств защиты обязательно должны входить как организационные, так и компьютерные мероприятия. В противном случае мы рискуем создать одностороннюю систему противодействия угрозам, вооруженную до зубов против определенного набора опасностей (допустим, вирусной или сетевой атаки), но абсолютно беспомощную против элементарной кражи.

Организационные меры включают:

- **подбор и расстановку кадров.** Поставить на ответственные посты компании лиц, не прошедших должную проверку, – все равно, что поселить лису в курятнике. Поэтому кадровые органы представляют собой самую первую линию обороны любой компании. Указанное замечание в особенности актуально относительно подбора лиц, имеющих доступ к любой информации или способных этот доступ получить в силу специфики выполняемых обязанностей. А это не кто иной, как различного рода администраторы, особенно администратор БД и системный администратор;
- **контроль за персоналом.** Люди – это источник основного риска для предприятия, в особенности персонал, имеющий беспрепятственный доступ к данным, представляющим государственную или коммерческую тайну. Ни одна внешняя угроза не сможет принести столько ущерба, сколько внутренняя. Поэтому даже сам факт того, что потенциальный «крот» будет знать, что все его действия проверяются, может если не свести к нулю, то, по крайней мере, существенно снизить риск прорыва системы безопасности;
- **защита служебных помещений и оборудования.** Грош цена всем стараниям всевозможных специалистов по безопасности, если можно просто войти в офис и снять жесткий диск с данными

из системного блока неохраняемого сервера компании. Поэтому во всех солидных организациях организуется пропускной режим, устанавливаются сигнализации, системы видеонаблюдения. Одним словом, осуществляется контроль за физическим доступом к оборудованию и к носителям информации;

- **планирование действий сотрудников в чрезвычайных ситуациях.** Никто не может быть застрахован от чрезвычайных ситуаций природного или техногенного характера. Пожары, наводнения, землетрясения и другие катаклизмы могут причинить невосполнимый ущерб любому из нас. Спасти жизни персонала и плюс к этому минимизировать экономические последствия от катастрофы поможет наличие четких и внятных инструкций по поведению сотрудников в тех или иных ситуациях. Одна-единственная строка в инструкции по пожарной безопасности, в которой будет написано, что при эвакуации системный администратор (или другое лицо) должен изъять и вынести в безопасное место архивы на электронных носителях информации, поможет сократить издержки компании по восстановлению своих баз данных.

Книга посвящена изучению особенностей разработки баз данных, а не обсуждению вопросов подбора персонала или изучению должностных обязанностей шефа безопасности компании. Поэтому читатель должен понимать, что мы рассмотрели организационные меры защиты информационных систем даже не просто поверхностно, а лишь вскользь упомянули об их существовании.

Перенесем акцент в более близкую нам область компьютерных мер защиты. База данных представляет собой элемент информационной системы и функционирует в рамках этой информационной системы. Поэтому она подвергается всем тем же опасностям, что и вычислительная сеть и компьютеры компании. Исходя из этого факта, информационная система предприятия должна быть защищена рядом мер, не всегда имеющих прямое отношение к БД:

- сетевая защита;
- защита от вирусов;
- настройка безопасности операционных систем;
- аутентификация сетевого пользователя;
- криптографическая защита трафика;
- резервное копирование;
- аудит безопасности.

Все перечисленные меры относятся к ведению системного администратора, но многие из них решаются во взаимодействии с администратором СУБД:

- аутентификация и авторизация пользователя;
- криптографическая защита БД;
- резервное копирование данных;
- аудит событий безопасности БД;
- модернизация системного и прикладного ПО;
- доступ к данным только при посредничестве представлений и хранимых процедур.

Идентификация, аутентификация и авторизация

Любой пользователь (или процесс), получающий доступ к БД, на этапе создания пользовательской сессии подлежит обязательной **идентификации** (identification). Все дальнейшие его действия так или иначе будут требовать предъявления этого идентификатора.

Одним из основных способов обеспечения конфиденциальности и целостности информации в БД выступает механизм аутентификации. **Аутентификация** (authentication) – это процедура проверки подлинности пользователя (точнее, его идентификатора). Обычно пользователь подтверждает то, что он является именно тем, за кого он себя выдает, путем ввода в систему уникальной (неизвестной другим) информации о себе. В простейшем случае это символьный пароль, но возможен и более творческий подход подтверждения подлинности. Все чаще используется биометрическая аутентификация (дактилоскопия, узор радужной оболочки и сетчатки глаз, анализ голоса) или электронные способы аутентификации (контактные и бесконтактные смарт-карты, радиочастотные идентификаторы, USB-ключи). Часто программисты изобретают нестандартные способы аутентификации, например приложение может попросить пользователя сообщить о своем домашнем адресе или расписаться в электронной форме мышкой и затем сравнить полученные данные с имеющимися в системе образцами.

В случае если пользователь осуществляет работу с базой данных в течение продолжительного периода времени, то стоит осуществлять периодическое выполнение процедур аутентификации, чтобы убедиться, что за компьютером находится санкционированный оператор, а не случайный прохожий. Кроме того, имеет смысл связывать наиболее критичные действия оператора (допустим, удаление наиболее важных

данных или уничтожение таблицы) с повторной процедурой аутентификации.

Если пользователь успешно прошел процедуру аутентификации, СУБД осуществляет его авторизацию. **Авторизация** (authorization) – это процедура предоставления пользователю определенных ресурсов и прав на их использование.

Замечание

В языке SQL предусмотрены специальные команды, предназначенные для определения прав пользователя, – это инструкции GRANT и REVOKE.

В современных СУБД используется один из двух методов определения ограничений на доступ к данным: избирательный (discretionary) или мандатный (mandatory). А иногда и оба совместно.

В случае избирательного доступа каждому пользователю, стремящемуся получить доступ к системе, предоставляются определенные полномочия по обращению с объектами БД. Диапазон полномочий достаточно широк – от ограниченного доступа к отдельному атрибуту или строке таблицы до всеобъемлющих прав относительно всей базы данных. Набор прав назначается администратором БД. Обычно пользователь вводится в группу с заранее предопределенными ролями (администратор сервера, администратор БД, пользователь БД, гость). Все дальнейшее взаимодействие пользователя с объектами БД строго регламентируется в соответствии с назначенной ролью. Пользователи могут обладать разными правами на один и тот же объект, одни могут лишь просматривать данные, другие – только добавлять, третьи – осуществлять чтение, вставку и редактирование.

Если система безопасности СУБД основана на мандатном доступе, то в этом случае каждому из объектов БД назначается определенный уровень допуска, например «особая важность», «совершенно секретно», «секретно», «для служебного пользования», «не секретно». Доступ к объекту получит только пользователь с полномочиями не ниже, чем уровень допуска к объекту.

Криптографическая защита

Криптографическая защита данных – это одно из самых сильных средств противодействия несанкционированному просмотру данных. В основе подавляющего большинства криптографических систем данных защиты выступает шифрование. Шифрование – это процесс преоб-

разования открытых данных с использованием специального алгоритма, после чего эти данные не могут быть восстановлены к исходному виду без ключа дешифрования.

Замечание

Большинство СУБД, помимо шифрования данных в таблицах БД, еще обеспечивает криптографическую защиту учетных записей пользователя, исключаящую их кражу.

Схематично процесс шифрования и дешифрования данных представлен на рис. 10.2. Исходные открытые данные S подвергаются криптографическому преобразованию, для этого они обрабатываются шифрующей функцией $E()$, в качестве параметра которой выступает назначаемый пользователем ключ шифрования $k1$. В результате мы получаем зашифрованные данные R , которые для стороннего наблюдателя выглядят как последовательность случайных символов. Восстановление открытых данных S из зашифрованных R возможно только в том случае, если мы обладаем функцией дешифрования D и знаем ключ дешифрования $k2$.

Различают два вида криптосистем: на основе симметричного и асимметричного шифрования. В симметричной криптосистеме для шифрования и расшифровки данных применяется один и тот же ключ. В таких системах для исключения несанкционированного доступа к информации ключ следует держать в секрете. В асимметричной криптосистеме применяется два ключа. Шифрование данных осуществляется несекретным ключом (для вскрытия данных он не подходит), а расшифровка проводится с помощью второго секретного ключа.

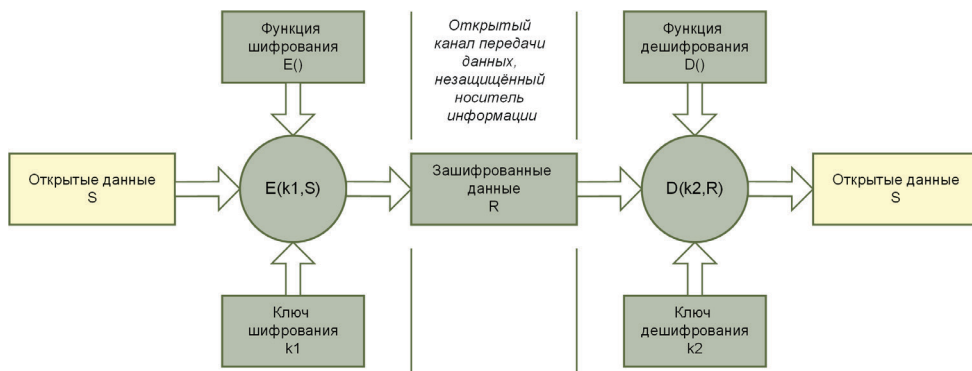


Рис. 10.2. Схема шифрования данных

На сегодня в мире широко используется несколько стандартов шифрования. Наиболее старый из них, появившийся на свет в 1977 году – симметричный алгоритм DES (Data Encryption Standard), он шифрует 64-битные блоки данных с помощью 64-битного ключа (56 бит значащих и 8 проверочных). Таким образом, ключ шифра DES имеет 2^{56} комбинаций. Для взлома криптосистемы DES с помощью современного компьютера обычному хакеру потребуется не один десяток лет, если же системой заинтересуются профессионалы, обладающие соответствующими техническими средствами, то DES сможет защитить ваши данные всего несколько минут.

Европейский стандарт симметричного криптоалгоритма называется IDEA (International Data Encryption Algorithm), он действует с 1990 года. Этот алгоритм весьма быстр и более стоек, чем DES. Длина ключа IDEA составляет 128 бит. В США с 2000 года в качестве стандарта принята симметричная криптосистема AES (Advanced Encryption Standard). Основным показателем криптостойкости AES выступает поддерживаемая длина ключа – 128, 192 и 256 бит. Как видите, алгоритм AES даже на основе самого короткого ключа превосходит 56-битный алгоритм DES. В России в качестве криптографического преобразования используется симметричный алгоритм, определяемый ГОСТ 29147–89. Это достаточно стойкий алгоритм с длиной ключа 256 бит.

Асимметричные алгоритмы применяются реже, чем симметричные. В первую очередь это объясняется их низкой скоростью и сложностью реализации. Наиболее известный асимметричный алгоритм шифрования называется RSA (первые буквы фамилий авторов идеи Rivest, Shamir и Adleman). Стойкость этого алгоритма с открытым ключом сопоставима с DES при условии применения длинного ключа (1024 бит и выше).

Замечание

Защищенная СУБД должна уметь шифровать: собственно хранящиеся в ней данные (включая служебную информацию), исходный код запросов, хранимых процедур и триггеров, данные, передаваемые к другим компьютерам по незащищенным каналам.

Помимо шифрования находящихся в таблицах данных, некоторые СУБД позволяют разработчику шифровать и тексты хранимых процедур и триггеров, это позволит исключить вероятность несанкционированной модификации кода.

Резервное копирование и восстановление

Для минимизации ущерба от вероятной потери данных необходимо регулярно создавать копию БД. В такую копию входят не только собственно данные, но и служебная информация. Если вы эксплуатируете простейшую настольную СУБД, то для резервирования достаточно скопировать папку с файлами БД. В продвинутых системах предусмотрены специальные утилиты, позволяющие не просто копировать данные, но и определять график, порядок сохранения данных и создавать инкрементные копии.

Определение

Резервной копией называют копию данных, которая может использоваться для восстановления данных в случае возникновения ошибки или для восстановления копии БД на другом сервере.

Для создания качественных резервных копий БД следует применять только специальное программное обеспечение, входящее в состав поставки вашей СУБД. Использование для этих целей сторонних утилит (например, программ-архиваторов) – плохая идея, в особенности если во время процедуры архивации продолжается работа пользователей с БД. Подобные копии наряду с данными будут содержать весь накопившийся в БД мусор от некорректных или незавершенных транзакций.

Администраторы данных для своих предприятий разрабатывают стратегии резервного копирования и восстановления, в которых учитываются такие факторы, как:

- 1) бизнес-требования организации к БД, в особенности в части, касающейся доступности данных и их защиты от потерь;
- 2) индивидуальные особенности БД (модель данных, размер, типичное использование);
- 3) ресурсы, привлекаемые к процедуре резервного копирования и восстановления (оборудование и персонал).

Важно заранее сформулировать предпочтительные сценарии восстановления БД. Будет ли это полное протоколирование, неполное протоколирование или простой способ восстановления. Выбранный сценарий восстановления определяет, каким образом СУБД станет управлять процессом регистрации транзакций.

Так, простой сценарий восстановления предполагает, что резервные копии журналов транзакций не ведутся. В такой ситуации процедура восстановления просто переведет БД в точку, соответствующую момен-

ту создания резервной копии, без какой-либо гарантии, что это устойчивое состояние.

В свою очередь, сценарии полного и неполного протоколирования предполагают ведение резервной копии журналов транзакций (в первом случае туда отображаются все операции с БД, а во втором – наиболее важные). Благодаря журналу транзакций возможно не просто восстановление последнего состояния БД, но и откат от этого состояния к наиболее приемлемой точке.

На предприятии должно существовать расписание резервного копирования. Хотя современные СУБД позволяют создавать резервные копии, не прекращая при этом обслуживать пользователей системы, расписание следует планировать так, чтобы процедура резервирования проводилась в часы наименьших нагрузок, допустим по окончании рабочего дня (недели, месяца).

Внимание!

Резервная копия должна защищаться от посягательств третьих лиц не хуже, чем сама БД. Одним из наиболее надежных способов защиты может стать шифрование данных при создании резервных копий.

Создание резервной копии – необходимое, но не достаточное условие для восстановления данных. Стоит разобраться и с местом хранения копий. Самое распространенное (и одновременно самое нелепое) решение сводится к размещению резервных копий на том же самом жестком диске, на котором находится сама БД. Как следствие при физическом повреждении диска копии уходят в небытие на таких же правах, что и сама база данных. Поэтому архивы следует содержать на альтернативных носителях. Архивные носители желательно не складировать в ящике письменного стола, а направлять в ячейку несгораемого сейфа. В свою очередь, сейф следует располагать в другом помещении. Помещение должно быть защищенным (охраняемым, пожаробезопасным, не подверженным затоплению и т. п.).

Аудит событий безопасности

Аудит событий безопасности БД представляет собой процесс получения и анализа данных о происходящих в системе событиях и степени их соответствия требованиям к защите данных. В результате аудита мы получаем оценку состояния защищенности базы данных, выявляем бреши в защите БД и вырабатываем рекомендации по совершенствованию действующих механизмов безопасности.

В идеале сбор информации о состоянии системы безопасности БД должен осуществляться непрерывно, для этого очень многие СУБД автоматически ведут журнал аудита (контрольный журнал). Журнал аудита обеспечивает индивидуальную ответственность пользователя за все его действия, для этого все события аудита однозначно связываются с учетной записью пользователя в СУБД. В журнале содержится:

- описание стандартного набора событий (авторизации пользователя, доступа к тем или иным данным и операций с ними; создания, модификации и уничтожения объектов БД; выполнение нештатных SQL-команд и т. д.);
- настраиваемый перечень атрибутов в отдельной записи журнала аудита (даты и времени события, идентификатор пользователя, имя и сетевой адрес компьютера, описание события, связанные с событием объекты, признак успешного или неудачного завершения события).

Для удобства анализа данных журнал обязан позволять фильтровать и сортировать свои записи. Не помешает отметить то, что журнал аудита сам по себе должен быть защищен от несанкционированного доступа.

В перечень событий аудита должны входить:

- 1) события предоставления прав доступа к БД;
- 2) события по созданию, изменению и удалению объектов БД;
- 3) события по вставке, редактированию и удалению данных;
- 4) события по изменению настроек сервера и прикладного ПО;
- 5) любые отказы в обслуживании пользователя;
- 6) попытки осуществить потенциально опасные операции без наличия соответствующих прав;
- 7) различного рода исключительные ситуации и ошибки в работе ПО.

Собранная информация передается на анализ независимому эксперту в области защиты информации. Заметьте, что слово «независимый» здесь ключевое. Многие компании даже ни на йоту не сомневаются в лояльности своего «специалиста» и безоговорочно доверяют ему проведение аудита. Такой подход хорош только с точки зрения экономии денежных средств. Даже если местный специалист кристально чист (что, к счастью, еще часто встречается), он далеко не обязательно является высоким профессионалом в области защиты информации и не всегда склонен замечать свои собственные ошибки. Посему работа независимого специалиста в любом случае принесет больше пользы, чем поверхностный аудит своих собственных сотрудников.

Замечание

Журнал аудита – очень важный, но далеко не единственный источник информации для эксперта. Полноценная аудиторская проверка обязательно проведет исследование всей информационной системы компании, изучит техническую и распорядительную документацию, побеседует с ключевыми сотрудниками, одним словом, получит много дополнительной информации об организации, которая позволит рассмотреть проблему безопасности в комплексе.

Модернизация программного обеспечения

Подавляющее большинство производителей программного обеспечения регулярно вносит доработки в свои продукты. Доработки содержат улучшенные алгоритмы, исправления выявленных ошибок, «заплатки» для противодействия недавно появившимся угрозам безопасности. Обычно доработки выкладываются на сервер производителя, а если вы находитесь на гарантийном обслуживании, то производитель лично извещает своего клиента о необходимости модернизировать ПО.

Администратору СУБД следует всегда находиться в курсе событий всех изменений и нововведений в эксплуатируемом им программном обеспечении и регулярно обновлять пакет своих программ. Если компания эксплуатирует эксклюзивное (разработанное специально для нее) ПО, то администратору СУБД стоит обязательно напомнить руководству компании о необходимости заключения (или продления) договора с производителем на сопровождение развернутого на предприятии ПО.

Внимание!

Перед любыми операциями, связанными с модернизацией программного обеспечения, следует создать резервную копию БД.

Безопасный доступ к данным

Рассмотренные выше способы защиты информации с помощью аутентификации, криптозащиты и резервного копирования в том или ином виде имеются во всех компьютерных системах. Наряду с ними СУБД обладает своими собственными механизмами самообороны, один из них базируется на применении **представлений** (view).

Специалисты по безопасности БД рекомендуют разработчикам воздерживаться от предоставления доступа пользователям непосредственно к таблицам. Вместо этого общение с данными должно осуществляться только через специальных посредников – представления.

Представление получается в ходе проведения над одной или несколькими таблицами нескольких реляционных операций, в результате получается новое отношение. С точки зрения пользователя полученное отношение выглядит как вполне полноценная таблица, хотя на самом деле она виртуальная. Так как виртуальная таблица собирается руками разработчика, он может исключить из нее какую-то часть атрибутов и записей, к которым пользователь не имеет прав доступа.

Замечание

В языке SQL имеется ряд команд, предназначенных для работы с представлениями: CREATE VIEW, ALTER VIEW и DROP VIEW.

В целях защиты от SQL-инъекций доступ к хранимым в сетевых БД данным должен осуществляться не через динамический SQL, а только с помощью **хранимых процедур** (stored procedure). Введя такое ограничение, мы резко снижаем вероятность прорыва наших оборонительных линий, так как код хранимой процедуры написан разработчиком БД и (конечно же, при условии достаточной квалификации программиста) не угрожает данным. Если без динамического SQL обойтись никак невозможно, то хорошей идеей станет проверка всех значений, передаваемых в параметры запроса, так как нельзя исключить вероятность того, что злоумышленник попытается встроить сюда свой вредоносный код.

Экономическая оправданность

Возможно, принцип экономической оправданности должен открывать (а не закрывать, как в нашем случае) перечень правил защиты БД. Суть принципа заключается в том, что для защиты ресурса (речь не только о БД) должен использоваться простейший (читай: дешевый) из всех вариантов защиты, главное, чтобы он позволял достичь поставленную цель – обеспечение заданной степени защиты ресурса. Иначе мы рискуем уйти в крайность, когда стоимость системы защиты будет превышать стоимость потенциального ущерба, наносимого нашей БД при самом неблагоприятном стечении обстоятельств.

Резюме

Безопасность хранимой в БД информации определяется не только возможностями системы управления базами данных. Кроме того, на безопасность оказывают влияние многочисленные составляющие, та-

кие как возможности ОС и используемого прикладного ПО, применяемые аппаратные и программные средства защиты, особенности решения организационных вопросов в компании, физические условия эксплуатации БД, степень подготовленности персонала, особенности законодательства страны и многие другие факторы. По понятным причинам на нескольких страницах невозможно раскрыть все перечисленные аспекты, но эта цель и не преследовалась. Задача главы была другая. Во-первых, убедить читателя в необходимости учета вопросов безопасности начиная с первых шагов работы над проектом БД. Во-вторых, доказать, что защищенная БД создается только за счет комплексного использования всех имеющихся в нашем распоряжении возможностей.

Тема безопасности БД будет продолжена в *главе 17*, в которой мы обсудим особенности определения прав пользователей и связанные с этим возможности языка SQL.

Вопросы для самопроверки

1. Что понимается под угрозой информационной системе?
2. Что понимается под безопасностью данных?
3. Раскройте смысл терминов:
 - a) конфиденциальность;
 - b) доступность;
 - c) целостность данных.
4. Что, на ваш взгляд, является источником угроз для БД?
5. Какие специфичные (присущие только БД) угрозы вам известны?
6. Дайте определение политики безопасности.
7. Что понимается под терминами:
 - a) идентификация;
 - b) аутентификация;
 - c) авторизация?
8. Прокомментируйте правила защиты БД.
9. Какие сценарии резервного копирования вам известны, в чем их отличия?
10. Какие события подлежат учету в журнале аудита безопасности?
11. Какие события должны протоколироваться в журнале аудита?
12. Что следует понимать под экономической оправданностью при организации защиты БД?

Глава 11

Знакомимся с SQL

В середине 70-х годов XX века, сразу после появления реляционной модели, специалисты БД приступили к разработке принципиально нового языка, предназначенного для управления данными. Среди огромного количества пожеланий, предъявляемых к делающему первые шаги языку, мы выделим самые ключевые. Перспективный язык реляционных баз данных должен был позволять:

- создавать базы данных, таблицы и другие объекты БД;
- выполнять основные операции редактирования данных в таблицах (вставка, модификация и удаление);
- выполнять запросы пользователя к данным, преобразующие хранящиеся в таблицах данные в выходные отношения.

Ко всему прочему разрабатываемый язык должен был в принципе отличаться от высокоуровневых языков программирования тех лет. Во-первых, базы данных работают в трехзначной логике. У них наряду с классическими для любого языка понятиями истина/ложь (True/False) предусмотрено третье значение неопределенности UNKNOWN. Во-вторых, новый язык создавался не только в интересах программистов, но и в интересах пользователей, поэтому в идеале он должен быть не процедурным, а декларативным. В соответствии с этим пользователь лишь ставит БД задачу (указывает, что ему нужно от БД), а каким образом СУБД станет решать поставленную задачу, пользователя не интересует.

Стандартом SQL (Structured Query Language) стал в 1986 году благодаря Американскому национальному институту стандартов (American National Standards Institute, ANSI) и Международной организации стандартизации (International Organization for Standardization, ISO). Кстати, первый стандарт SQL иногда называют по имени принявшей его организации – ANSI SQL.

В 1990-х годах официально действующим и общепризнанным стал считаться стандарт SQL:92, принятый, как вы уже догадались, в 1992 году. Практически любая серьезная компания, разрабатывающая СУБД, старается поддерживать требования SQL:92.

В 1999 году на сцене появился очередной стандарт. В этом году было опубликовано пять частей стандарта SQL-3 (SQL:99):

- Framework – концептуальная структура стандарта;
- Foundation – базисное описание SQL.
- Call-Level Interface (SQL/CLI) – уточнения к интерфейсу уровня вызовов;
- Persisted Stored Modules (SQL/PSM) – уточнение описания хранимых процедур;
- Host Language Bindings (SQL/Bindings) – определение правил взаимодействия SQL и ряда стандартных языков программирования.

Спустя некоторое время появилось еще три части стандарта:

- Management of External Data (SQL/MED) – управление внешними данными;
- Object Language Bindings (SQL/OLB) – правила взаимодействия с объектно-ориентированными языками;
- Information and Definition Schemas (SQL/Schemata) – информационная схема.

Однако многие специалисты вновь скептически отнеслись к SQL-3, обвинив его в незавершенности. Вместе с этим новый стандарт сделал важный шаг в направлении поддержки объектных БД. В частности, здесь были объявлены структурные определяемые пользователем данные (User Defined Type, UDT) и типизированные таблицы (Typed Table).

Во многом по этой причине в 2003 году к вопросу модернизации SQL вернулись вновь. В обновленный стандарт с необходимыми изменениями вошли все части прежнего SQL:99 (правда, часть SQL/Bindings в самостоятельном виде существовать перестала и была включена во вторую часть стандарта SQL/Foundation). В дополнение к перечисленным выше частям SQL:2003 приобрел еще несколько документов:

- Routines and Types Using the Java Programming Language (SQL/JRT) – взаимодействие с языком Java;
- XML-Related Specifications (SQL/XML) – работа с XML-документами.

После 2003 года очередные версии стандарта SQL выходили примерно раз в 3–5 лет (рис. 11.1), так что на момент написания этих строк последним действующим стандартом считается SQL:2016, этот стандарт вобрал в себя наработки всех своих предшественников.

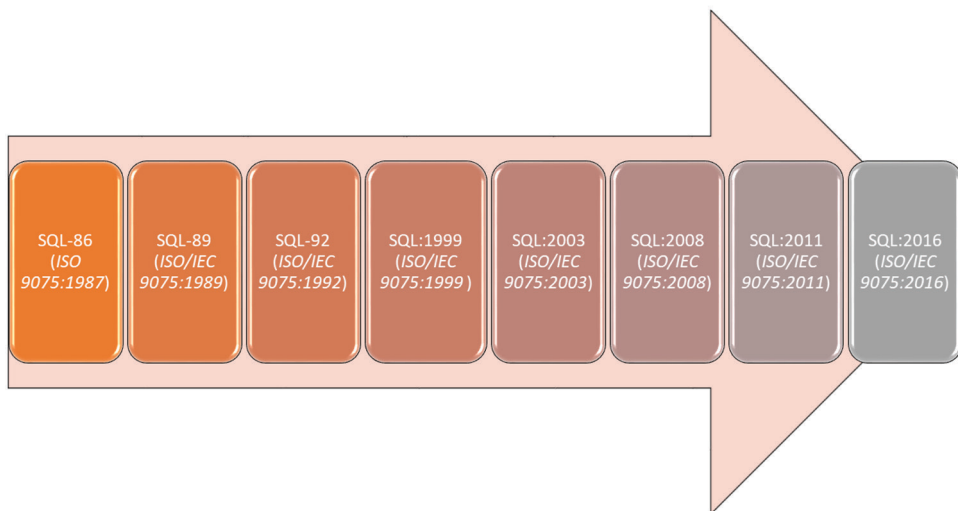


Рис. 11.1. Хронология выхода стандартов SQL

История совершенствования SQL отчасти подтверждает один из неписаных законов программирования: лучшее – враг хорошего. С каждым очередным витком развития стандарта все меньше и меньше производителей программного обеспечения могут его поддерживать в строгом соответствии с его требованиями. У признанного гуру в области баз данных Криса Дейта на этот счет есть хорошее высказывание, сделанное еще на рубеже веков: «...в наши дни ни один программный продукт не поддерживает полностью даже SQL:92; вместо этого такие продукты, как правило, поддерживают то, что можно было бы назвать “надмножеством подмножества” стандарта...» [19]. Как следствие стандарт не поспевает за производителями, а это неминуемо ведет к появлению различных ветвей языка, что с каждым годом все более и более минимизирует вероятность появления редакции SQL, однозначно на 100 % поддерживаемой всеми разработчиками ПО.

Замечание

Полную спецификацию стандарта SQL:2016 вы можете найти в интернете на сайте ISO, например по адресу <https://www.iso.org/standard/63555.html>.

Возможности SQL

Если читатель только начинает знакомиться с SQL, то необходимо сразу заметить, что это весьма мощный, но далеко не всемогущий язык. В сферу интересов SQL не попали задачи, стоящие перед прикладным и тем более системным программистом. Ни реализации низкоуровневых операций ввода-вывода, ни вопросы построения пользовательского интерфейса, ни организации работы с периферийными устройствами и т. п. Одним словом, на SQL не напишешь ни одного даже самого элементарного приложения для Windows, OS X, Linux или для любой другой ОС.

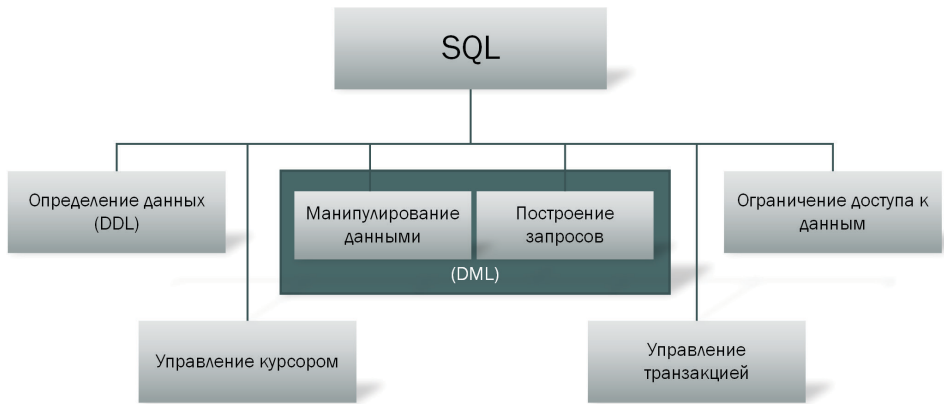


Рис. 11.2. Основные задачи языка SQL

Язык SQL выступает неотъемлемой частью реляционных СУБД и применяется только в интересах обработки данных. В общем случае можно выделить следующие задачи SQL (рис. 11.2):

- **определение данных.** Реализуется средствами подъязыка определения данными (DDL, Data Definition Language). Язык нацелен на решение вопросов создания и удаления базы данных и ее объектов. Перечень объектов БД достаточно велик, это таблицы, представления, индексы, курсоры, определения доменов. Визитной карточкой DDL выступают операторы `CREATE`, `ALTER` и `DROP`;
- **манипулирование данными (DML, Data Manipulation Language)** обеспечивает проведение операций вставки, редактирования и удаления данных из таблиц БД:
 - **манипулирование данными.** Для модификации данных в распоряжении DML предоставлено три команды: `INSERT`, `UPDATE` и `DELETE`;

- построение запросов. Вторая и наиболее востребованная часть DML, основанная на инструкции `SELECT`, позволяет извлекать данные из одной или нескольких таблиц;
- ограничение доступа к данным. Определяет набор прав пользователей при работе с объектами БД. В основу положены две команды `GRANT` и `REVOKE`;
- управление курсором позволяет обрабатывать данные построчно. Задача решается за счет квартета команд: `DECLARE CURSOR`, `OPEN CURSOR`, `FETCH CURSOR`, `CLOSE CURSOR`;
- управление транзакцией. Включает инструкции `SET TRANSACTION`, `BEGIN TRANSACTION`, `COMMIT` и `ROLLBACK`. Язык позволяет определять уровень изоляции транзакции, стартовать, фиксировать или возвращать транзакцию в исходное состояние.

В последующих главах книги мы узнаем, каким образом с помощью SQL решается большинство из перечисленных выше задач.

Типы данных SQL

Знакомство с языком начнем с рассмотрения поддерживаемых им типов данных. На рис. 11.3 представлен перечень стандартных типов данных SQL. В распоряжении СУБД, поддерживающей SQL:92, имеется богатый набор предопределенных типов данных, который включает:

- точные числовые типы;
- приближенные числовые типы;
- типы для работы с датой и временем;
- временные интервалы;
- логические типы данных;
- строки символов;
- битовые строки.

Если СУБД совместима с более новыми стандартами, то к перечню добавляется еще несколько типов. В стандарте SQL:2003 их называют непредопределенными (non-predefined) и даже более жестко – типами данных, не относящимися к SQL (non-SQL types). С некоторой степенью допущения их можно причислить к структурным типам данных, имеющимся в большинстве высокоуровневых языков программирования:

- коллекции;
- последовательности;

- типы данных, определяемые пользователем;
- ссылочные типы.

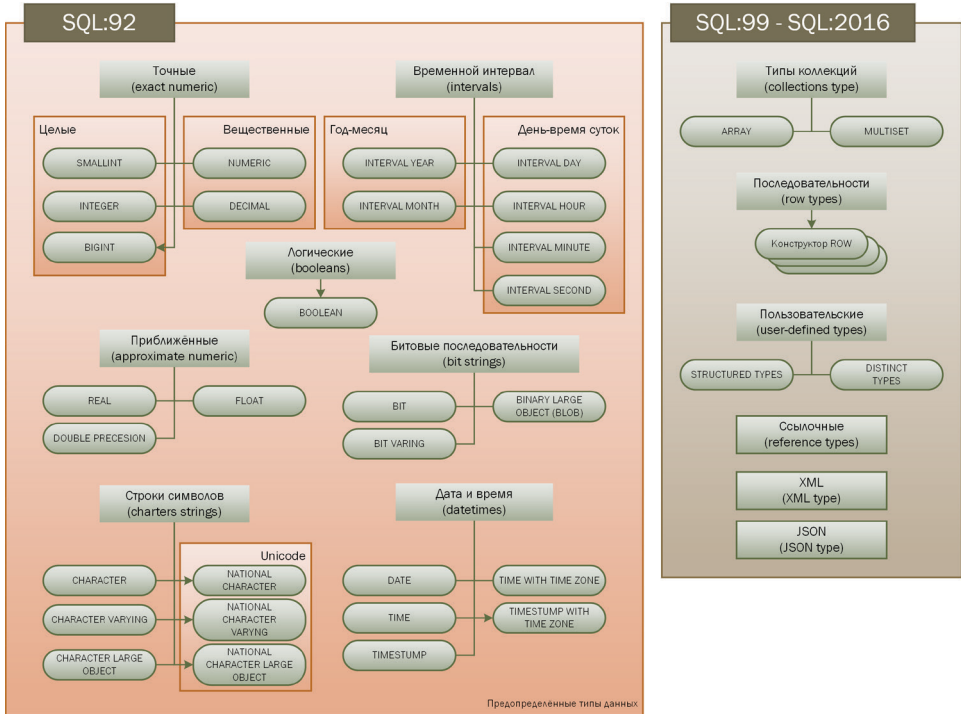


Рис. 11.3. Типы данных SQL

Кроме того, современные стандарты SQL способны работать с популярными сегодня форматами данных XML и JSON.

Где используются перечисленные типы данных? Во-первых, при определении столбцов таблиц. Во-вторых, при объявлении переменных в хранимых процедурах, функциях, определяемых пользователем, и триггерах. В-третьих, при организации обмена данными между БД и клиентским приложением.

Предопределенные типы

Точные числовые типы (exact numeric) предназначены для обслуживания целочисленных значений и значений, имеющих дробную часть без потерь точности. При задании такого типа данных необходимо указать два аргумента: точность (n) и масштаб (m). Точность задает общее число значащих цифр, используемых при отображении числа. Масштаб опре-

деляет число значащих цифр справа от десятичной точки. Обязательно должно соблюдаться условие: точность больше масштаба ($n > m$). Масштаб не является обязательным аргументом, если его не указывать, то он считается равным 0. Например, тип данных `NUMERIC(5, 2)` определяет число, состоящее не более чем из 5 цифр, включая две цифры после запятой.

В табл. 11.1 представлены основные типы точных чисел. Единственным дополнением к набору точных числовых данных, существовавших с первых версий стандарта SQL, стал введенный в 1999 году тип данных `BIGINT`.

Таблица 11.1. Точные числовые типы

Спецификация	Описание
<code>NUMERIC [(n[, m])]</code>	Точное число, описываемое аргументами n и m
<code>DECIMAL [(n[, m])]</code> или <code>DEC [(n[, m])]</code>	В отличие от <code>NUMERIC</code> , способно хранить число дальше с большей точностью, чем определено в аргументе m . Поэтому говорят, что <code>NUMERIC</code> задает реальное значение точности, а <code>DECIMAL</code> – минимальное значение точности
<code>BIGINT</code>	Тип данных, предназначен для хранения больших целых чисел (обычно 64 бит). Типы данных <code>BIGINT</code> , <code>INTEGER</code> и <code>SMALLINT</code> являются частным случаем типа данных <code>NUMERIC</code> , у которого масштаб установлен в 0, а точность определена возможностями СУБД
<code>INTEGER</code> или <code>INT</code>	Тип данных, предназначен для хранения целых чисел (обычно 32 бит)
<code>SMALLINT</code>	Тип данных, предназначен для хранения малых целых чисел (обычно 16 бит)

Количество байт, необходимых для хранения значений `NUMERIC` и `DECIMAL`, состоит в прямой зависимости от размерности аргумента n . Например, если вы намерены хранить число из 15–16 сохраняемых знаков, то вам потребуется 8 байт, 34–36 знаков – 16 байт.

Замечание

Практически во всех СУБД для работы с денежными величинами на базе `NUMERIC` реализован свой собственный специализированный тип данных. Такие типы данных хранят действительные числа с точностью до 4-го знака после запятой.

Приближенные числовые типы (`approximate numeric`) представляют собой тип данных для определения чисел с плавающей точкой, осуществ-

вляющий хранение числа в научном формате (мантисса плюс порядок). Имеет аргумент точность (n), но, в отличие от типа NUMERIC, не обладает масштабом. Термин «приближенный» вовсе не означает, что, внося в столбец таблицы число, допустим 4 целых 5 десятых, то на следующий день вы на этом месте найдете приблизительно 5. Просто тип данных предназначен для обслуживания значений, не требующих высокой точности (табл. 11.2).

Таблица 11.2. Приближенные числовые типы данных

Спецификация	Описание
REAL	Точность и предел значений зависят от СУБД. Как правило, занимает в памяти 6 байт и в состоянии хранить число в интервале от $-3,4E - 38$ до $+3,4E + 38$ с точностью до 7 цифр после запятой
FLOAT [(n)]	Аргументом n определяется минимальное значение точности. Если значение n не указано, то потребует 8 байт памяти. Тип данных способен хранить число в интервале от $1,7E - 308$ до $+1,7E - 308$ с точностью до 15 знаков
DOUBLE PRECISION	Точность определяется версией СУБД, превышает точность REAL

Логический тип данных (boolean type) языка SQL весьма неординарен. Особенность заключается в том, что два классических элемента (true/false) булевой логики здесь дополнены третьим значением – неопределенностью Unknown. Как следствие логика становится более сложной – трехзначной. В табл. 11.3 представлены результаты основных операций логического «И» (AND) и «ИЛИ» (OR).

Таблица 11.3. Таблица основных логических операций

Операция AND	TRUE	FALSE	UNKNOWN
TRUE	TRUE	FALSE	UNKNOWN
FALSE	FALSE	FALSE	FALSE
UNKNOWN	UNKNOWN	FALSE	UNKNOWN

Операция OR	TRUE	FALSE	UNKNOWN
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	UNKNOWN
UNKNOWN	TRUE	UNKNOWN	UNKNOWN

Есть особенность и у операции отрицания NOT. Если оператор NOT, примененный к истине, возвращает ложь – NOT (TRUE) IS FALSE, а ко лжи, наоборот, – NOT (FALSE) IS TRUE, то операция отрицания неизвестности вернет неизвестность NOT (UNKNOWN) IS UNKNOWN.

Тип данных строки символов (characters strings) специализируется на обслуживании текстовых данных (табл. 11.4). Во всех случаях работы с текстовыми данными (при объявлении строковой переменной, описании столбца таблицы, параметра хранимой процедуры и т. п.) следует указывать размерность строки n.

Таблица 11.4. Строки символов

Спецификация	Описание
CHARACTER [n], или сокращенно CHAR[n]	Тип данных предназначен для создания текстовой строки фиксированной длины. Количество символов в строке определяется в квадратных скобках после указания типа данных. Если в поле типа CHAR помещается текстовое значение меньшего размера, чем размерность поля, то оставшиеся позиции символов заполняются пробелами
CHARACTER VARYING[n], или сокращенно VARCHAR[n]	Текстовая строка переменной длины. Максимальный размер строки определяется в квадратных скобках. Преимущество такого типа данных над типом CHAR заключается в том, что здесь пустые позиции не заполняются пробелами и, соответственно, таблица требует меньшего размера оперативной и дисковой памяти
NATIONAL CHARACTER [n], или сокращенно NCHAR [n]	Строка национального символьного набора фиксированной длины
NATIONAL CHARACTER VARYING [n], или сокращенно NCHAR VARYING [n]	Строка национального символьного набора переменной длины
CHARACTER LARGE OBJECT [n], или сокращенно CLOB [n]	Тип данных CHARACTER LARGE OBJECT предназначен для определения столбцов таблиц, хранящих большие группы символов
NATIONAL CHARACTER LARGE OBJECT [n]	Тип данных NATIONAL CHARACTER LARGE OBJECT предназначен для определения столбцов таблиц, хранящих большие группы национальных символов

Данные типа CHAR и NCHAR в операторах SQL должны выделяться одинарными кавычками, кроме того, при использовании символов национальных алфавитов следует указать спецификацию набора символов, воспользовавшись командой CHARACTER SET.

STR = 'Символьные переменные берутся в кавычки'

Для данных символьного типа допускается операция сложения (конкатенация).

STR = 'Сложение ' + 'строка'

Битовая последовательность (bit strings) предназначена для хранения любой двоичной информации. Тип данных универсален и позволяет описывать как простейшие логические данные, так и сложные объекты, например файлы мультимедиа (табл. 11.5).

Таблица 11.5. Битовые последовательности

Спецификация	Описание
BIT [n]	Битовая последовательность фиксированной длины. Аргумент n устанавливает длину последовательности в битах. Если аргумент отсутствует (или установлен в 1), то тип данных используется для создания полей логического типа (Да/Нет). Особенность типа данных фиксированной длины в том, что попытка записать в поле этого типа значения меньшей длины, чем указано в аргументе n, приведет к ошибке
BIT VARYING [n]	Битовая последовательность переменной длины. Максимальное значение битовой последовательности указывается в аргументе n
BINARY LARGE OBJECT [n], или сокращенно BLOB [n]	Тип данных предназначен для хранения больших объектов. Например, файлов мультимедиа и изображений

За описание значений даты и времени (datetime) отвечает пять типов данных. Дата представляется в формате общепринятого в большинстве стран мира григорианского календаря (табл. 11.6).

Таблица 11.6. Тип данных дата и время

Спецификация	Описание
DATE	Тип данных включает три поля: YEAR (год) – от 0001 до 9999; MONTH (месяц) – от 01 до 12; DAY (день) – от 01 до 31. Формат записи: «уууу-мм-дд». Полное число позиций, требуемых для отображения даты (вместе с разделителями), – 11
TIME [(n)]	Тип данных включает три поля: HOUR (часы), MINUTE (минуты), SECOND (секунды). Если аргумент точности (n) не определен, то полное число позиций (вместе с разделителями) равно 8, а формат записи: «hh:mm:ss». Если вы укажете аргумент точности, то получите возможность работать с долями секунд
TIMESTAMP [(n)]	Метка даты-времени, представляющая собой комбинацию типов данных DATE и TIME
TIME WITH TIME ZONE	Тип данных аналогичен TIME плюс два дополнительных значения, характеризующих смещение от Гринвичского меридиана в часах TIMEZONE_HOUR и минутах TIMEZONE_MINUTE. Из-за учета временной зоны количество позиций увеличивается с 8 до 14
TIMESTAMP WITH TIME ZONE	Метка даты-времени плюс смещение от Гринвича. Число позиций для отображения даты, времени и временной зоны максимальное – не менее 25

Значения даты и времени допускается задавать литералами в одинарных кавычках, перед значением следует указать название типа, например: `TIMESTUMP '2018-11-21 12:00:00'`.

Интервал (interval) представляет собой производную от типа данных дата-время и предназначен для описания промежутка времени между двумя временными отсчетами. Стандарт SQL различает две категории интервалов: год-месяц (year-month) и день-время (day-time). Как можно догадаться по названию, первая разновидность интервалов оперирует сравнительно большими значениями год YEAR и месяц MONTH. Вторая категория интервалов обладает меньшим диапазоном, но в ка-

честве компенсации может похвастаться точностью до долей секунды. В диапазон интервала день-время входят такие значения, как день DAY, час HOUR, минута MINUTE и секунда SECOND. Синтаксических конструкций для определения интервала несколько, в самом общем виде достаточно рассмотреть следующий вариант:

```
INTERVAL start (p) [TO end (q)]
```

где: «start» и «end» – YEAR, MONTH, HOUR, MINUTE и SECOND. Явного определения параметров «p» и «q» обычно не требуется, в этом случае в них передается значение по умолчанию – 2. Двойка является минимальным значением для «p», а верхняя ограничивающая планка зависит от конкретной реализации СУБД. Например, мы планируем задать столбец таблицы, предназначенный для хранения временного интервала до 10 000 лет, тогда в команде SQL появится следующая строка:

```
INTERVAL YEAR (4)
```

Цифра 4 скажет SQL о том, что в столбце таблицы могут храниться значения временного интервала до 4 значащих цифр – диапазон от 0 до 9999.

Параметр «q» используется только в тех случаях, когда точность интервала задается в секундах, тогда «q» определит точность интервала до долей секунды. По умолчанию q=2, это означает, что мы учитываем только две значащие цифры перед запятой. Если мы намерены хранить значение времени с максимальной точностью (до 4 знаков после запятой) – присвоим q значение 6:

```
INTERVAL MINUTE TO SECOND(6)
```

Для определения конкретного значения в формате типа данных интервал следует воспользоваться следующим синтаксисом:

```
INTERVAL '12:54' HOUR TO MINUTE
```

Как вы догадались – это интервал, равный 12 часам 54 минутам.

С интервалами можно проводить операции сложения и вычитания, кроме того, интервал можно умножить или разделить на вещественное число. В операциях с интервалами могут использоваться типы данных дата-время, например:

```
DATE '2015-01-01' + INTERVAL '0005-1'
```

Операция прибавляет к дате интервал в пять лет и один месяц. В результате сложения мы получим новую дату: DATE '2020-02-01'.

Непредопределенные типы

Большинство нестандартных, или, как их еще называют, непредопределенных (non-predefined) типов данных вошло в состав SQL сравнительно недавно. В SQL:1999 появилась спецификация коллекции (collection type) и массива (array), а в SQL:2003 к стандарту добавилось мультимножество (multiset). Кроме того, стандарт признал право на существование таких типов, как последовательности, пользовательский тип, тип данных XML, JSON и ссылочный тип (рис. 11.2).

Основная особенность непредопределенных типов в том, что даже в действующем стандарте SQL их называют типами данных, не соответствующими SQL (non-predefined and non-SQL types). Несоответствий много, но первое, что бросается в глаза, – нарушение требования к атомарности данных, которое предписывает, чтобы в одной ячейке таблицы хранилось одно-единственное неделимое значение. И действительно, массив, мультимножество и последовательность тяжело рассматривать как атомарный тип, и если такой тип данных определяет колонку таблицы, то мы сразу сталкиваемся с проблемами нормализации данных.

Указанное противоречие появилось из-за стремления разработчиков стандарта расширить возможности реляционных БД по хранению данных и в первую очередь разрешить хранить в БД сложные объекты, дабы позволить разработчикам создавать объектно-реляционные БД.

Массив

Синтаксическая конструкция по определению массива выглядит следующим образом:

```
тип_данных ARRAY [n];
```

В качестве типа данных может выступать любой допустимый в стандарте тип данных, параметр *n* описывает число элементов в массиве. Как видите, стандартный SQL предусматривает лишь задание одномерного массива, определение массивов большей размерности не предусмотрено.

```
INT ARRAY[10];
```

Внимание!

Отсчет элементов в массиве SQL начинается с 1, а не с 0, как в большинстве языков программирования!

Множество и мультимножество

Множество SET и мультимножество MULTiset отличаются друг от друга лишь тем, что множество не допускает повтора значений, а мультимножество допускает.

Для задания множества (мультимножества) следует указать тип подлежащих хранению данных:

тип_данных MULTiset

Последовательность

Последовательность строится средствами конструктора типа ROW и представляет собой пары <имя_поля> <тип_данных>. В сконструированную последовательность может входить несколько пар. Для демонстрации возможностей последовательностей воспользуемся следующим примером: допустим, что в таблице DEMOTABLE мы собираемся хранить фамилию, имя и отчество человека в одном столбце FIO. В таком случае последовательность ROW позволит нам разделить столбец на три части (листинг 11.1).

Листинг 11.1. Пример использования последовательности

```
CREATE TABLE DEMOTABLE
(DEMOTABLE_KEY INTEGER PRIMARY KEY,
 FIO ROW (SNAME VARCHAR(20),
          FNAME VARCHAR(15),
          LNAME VARCHAR(15)),
 BDAY DATETIME);
```

Пользовательский тип

Предопределенные типы данных далеко не всеильны и зачастую не в состоянии охватить все потребности проектируемой БД. В подобных случаях стандарт предусматривает возможность проектирования **пользовательских типов данных** (user-defined types), описание которых должно сохраняться в системном каталоге. При создании нового типа данных следует опираться на уже существующие типы.

Объявленная стандартом SQL:2003 синтаксическая конструкция определения пользовательского типа весьма громоздкая, мы остановимся на несколько сокращенном варианте:

```
CREATE TYPE имя типа
[UNDER имя супертипа]
AS тип данных
[DEFAULT] значение
[[NOT] FINALL]
```

В простейшем случае для создания пользовательского типа «короткая строка» можно воспользоваться следующей конструкцией:

```
CREATE TYPE SHORT_STRING_TYPE AS CHAR(10)
```

Стандарт не ограничивает сложности пользовательского типа данных, что теоретически позволяет нам объявлять достаточно замысловатые структуры (листинг 11.2).

Листинг 11.2. Создание родительской структуры ADDRESS_TYPE

```
CREATE TYPE ADDRESS_TYPE AS
(ZIPCODE CHAR(6),
CITYNAME VARCHAR(20) NOT NULL,
STREET VARCHAR(20) NOT NULL,
HOME VARCHAR(3) NOT NULL)
NOT FINALL
```

Необязательное ключевое слово **FINALL** указывает, что пользовательский тип не может иметь подтипов, соответственно, **NOT FINALL** предполагает, что мы можем создавать дочерние типы данных. Выше был приведен пример родительского типа данных, специализирующегося на хранении адреса (почтовый индекс, город, улица и дом). А теперь подумаем о том, как добавить к нему номер телефона (листинг 11.3).

Листинг 11.3. Создание дочерней структуры ADDRESSEX_TYPE

```
CREATE TYPE ADDRESSEX_TYPE
UNDER ADDRESS_TYPE
AS
(PHONENUM CHAR(11))
FINALL
```

Пример демонстрирует, что дочерний подтип данных в состоянии не только унаследовать родовые характеристики родительского типа данных, но и дополнить их своими.

Вполне естественно, что возможности пользовательских типов данных определяются особенностями диалекта SQL. В некоторых СУБД пользовательские типы пока вообще не поддерживаются, в других отличаются от стандарта, в третьих сильно упрощены.

Другие типы

Ссылочный тип (reference types, REF) может применяться при определении переменных и параметров. Значение ссылки REF может указывать на строку в типизированной таблице (таблице, описанной на основе какого-то структурированного типа данных).

Язык XML (Extensible Markup Language) более подробно будет рассмотрен немного позднее (см. главу 19), поэтому мы пока ограничимся лишь пояснением, что XML представляет собой язык наращиваемой разметки, позволяющий описывать структурированные данные.

Тип JSON представляет собой текстовый формат обмена данными, основанный на JavaScript.

Подводя итоги разговора о дополнительных типах данных SQL, следует отметить, что в рамках стандарта появились типы данных, противоречащие ряду фундаментальных требований к реляционной модели. Это указывает на стремление совершенствовать концепцию реляционных баз данных, внедряя в нее возможности объектно-ориентированного подхода (см. главу 22).

Замечание

Практически в каждой из СУБД имеются специфичные для нее типы данных, не имеющие аналогов в стандарте. В качестве характерного примера стоит привести PostgreSQL, в котором реализованы экзотические типы, предназначенные для хранения пространственных и геометрических данных (BOX, CIRCLE, LINE и т. д.).

Константы

Для числовых типов данных определены константы в виде последовательности цифровых символов с необязательным заданием знака числа и десятичной точкой:

- 1000.5

Для определения строковой константы следует воспользоваться одинарными кавычками:

```
'Иван Иванович Иванов'
```

При назначении даты и времени лучше всего руководствоваться требованиями ISO. Дата описывается в формате `yyyy-mm-dd`, время – `hh:mm:ss`.

Преобразование данных

Существование многочисленных типов данных подразумевает возможность взаимного преобразования значений из одного формата в другой. В большинстве СУБД поддерживаются две разновидности преобразования типов данных: **неявное** (*implicit type conversions*) и **явное** (*explicit type conversions*). Неявное преобразование осуществляется автоматически, без вмешательства разработчика. Например, в MySQL вполне допустим подход, предложенный в листинге 11.4.

Листинг 11.4. Пример неявного преобразования в MySQL

```
SELECT 2+'2';  
-> 4
```

В нашем примере MySQL автоматически конвертирует литерал '2' в число и возвратит результат сложения. Но в этом случае, по крайней мере, присутствует некоторая логика, чего нельзя сказать о следующем примере (листинг 11.5).

Листинг 11.5. Демонстрация недостатка неявного преобразования в MySQL

```
SELECT 'Hello'+2;  
-> 2
```

Попытка «сложить» абсолютно несовместимые величины (текст и целочисленное значение) не вызовет у MySQL никаких эмоций, и сервер возвратит число 2, хотя логичнее было бы генерировать исключительную ситуацию. Таким образом, можно сделать вывод, что неявное преобразование типов далеко не всегда станет вашим надежным помощником, и его можно применять только в тех случаях, когда вы однозначно уверены в типах исходных данных.

В основу явного преобразования положено выражение CAST.

CAST <исходные данные> AS <целевой тип данных>

Функция способна преобразовать к типу CHARACTER все типы данных, работающих с датой и временем (листинг 11.6).

Листинг 11.6. Функция CAST в инструкции SELECT

```
SELECT DNNUM, CAST(DNDATE AS CHAR(24)) AS DNDATE_TXT
FROM DELIVERYNOTE
```

Надо понимать, что функция CAST() не всесильна и обладает рядом ограничений, обусловленных логикой и здравым смыслом (табл. 11.7). Для обозначения типов данных применялись следующие аббревиатуры:

- EN = Exact Numeric,
- AN = Approximate Numeric,
- C = Character (Fixed- или Variable-length, или character large object),
- VC = строка Character переменной длины,
- CL = Character Large Object,
- D = Date,
- T = Time,
- T = Timestamp,
- YM = Year-Month Interval,
- DT = Day-Time Interval,
- BO = Boolean,
- UDT = пользовательский тип,
- BL = Binary Large Object,
- RT = ссылочный тип,
- CT = коллекция,
- RW = последовательность.

Таблица 11.7. Допустимые преобразования типов данных

Исходный тип данных	Целевой тип данных															
	EN	AN	VC	C	D	T	TS	YM	DT	BO	UDT	CL	BL	RT	CT	RW
Exact Numeric	+	+	+	+				?	?		?	+		?		
Approximate Numeric	+	+	+	+							?	+		?		
Character	+	+	+	+	+	+	+	+	+	+	?	+		?		

Исходный тип данных	Целевой тип данных															
	EN	AN	VC	C	D	T	TS	YM	DT	BO	UDT	CL	BL	RT	CT	RW
Date			+	+	+		+				?	+		?		
Time			+	+		+	+				?	+		?		
Timestamp			+	+	+	+	+				?	+		?		
Year-Month Interval	?		+	+				+			?	+		?		
Day-Time Interval	?		+	+					+		?	+		?		
Boolean			+	+						+	?	+		?		
User Defined Type	?	?	?	?	?	?	?	?	?	?	?	?	?	?		
Binary Large Object											?		+	?		
Reference type	?	?	?	?	?	?	?	?	?	?	?	?	?	?		
Collection type															?	
Row types																?

Само собой разумеется, что далеко не все типы данных допускают взаимную конвертацию своих значений. Например, метку времени `TIMESTAMP` не представить в виде интервала `INTERVAL`, а такой экзотический тип данных, как последовательность `ROW`, не всегда можно преобразовать даже в другую последовательность. Поэтому в описании стандарта SQL определен перечень допустимых преобразований типов данных (табл. 11.7). Символом «+» в таблице обозначен тот случай, когда преобразование реально и в результате его осуществления мы не теряем точности данных, если преобразование возможно при стечении определенных условий, то оно отмечено символом «?». Пустая ячейка свидетельствует о невозможности корректной трансформации.

Операторы

В SQL, как, впрочем, и в любых других языках программирования, существует стандартный набор операторов, применяемых для осуществления математических, логических операций, операций сравнения и т. п.

Операция присваивания

В SQL операция присваивания обычно осуществляется с помощью оператора «:=» (реже «:=»). Этот оператор широко применяется в теле инструкции SELECT (листинг 11.7).

Листинг 11.7. Узнаем число записей в таблице и сохраняем в переменной @X

```
SELECT @X:=COUNT(*) FROM SUPPLIERS;
```

Арифметические операторы

Если речь идет о числовых типах данных, то с ними могут осуществляться следующие стандартные арифметические операции:

- «+» сложения;
- «-» вычитания;
- «*» умножения;
- «/» деления.

Кроме того, в ряде диалектов SQL можно встретить операторы:

- «DIV» целочисленного деления, например 7 DIV 2; -- результат 3;
- «%» (MOD) остатка от деления, например 7 % 2; -- результат 1.

Еще раз подчеркнем, что, за небольшим исключением, в качестве операндов должны выступать только числа. Технически, если вы допускаете неявное приведение типов, то арифметические операции можно осуществлять и со строковыми данными, при условии что они содержат числовые значения. Кроме того, арифметические операторы могут использоваться при работе с такими типами данных, как дата, время и интервал (табл. 11.8).

Таблица 11.8. Операторы, применимые с датой, временем и интервалами

Операнд	Оператор	Операнд	Результат
Datetime	-	Datetime	Interval
Datetime	+ или -	Interval	Datetime
Interval	+	Datetime	Datetime
Interval	+ или -	Interval	Interval
Interval	* или /	Numeric	Interval
Numeric	*	Interval	Interval

Логические операторы

Язык SQL поддерживает стандартный перечень логических операций:

- AND – логическое умножение («И»);
- OR – логическое сложение («ИЛИ»);
- NOT – логическое отрицание («НЕ»).

Ряд диалектов поддерживает оператор:

- XOR – исключение «ИЛИ».

В табл. 11.9 представлены результаты основных операций логического «И» и «ИЛИ».

Таблица 11.9. Таблица основных логических операций

Операция AND	TRUE	FALSE	NULL
TRUE	TRUE	FALSE	NULL
FALSE	FALSE	FALSE	FALSE
NULL	NULL	FALSE	NULL

Операция OR	TRUE	FALSE	NULL
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	NULL
NULL	TRUE	NULL	NULL

Операция XOR	TRUE	FALSE	NULL
TRUE	FALSE	TRUE	NULL
FALSE	TRUE	FALSE	NULL
NULL	NULL	NULL	NULL

Есть особенность и у операции отрицания NOT. Если оператор NOT, примененный к истине, возвращает ложь – NOT (TRUE) = FALSE, а ко лжи, наоборот, – NOT (FALSE) = TRUE, то операция отрицания неизвестности вернет неизвестность NOT (UNKNOWN) = UNKNOWN.

Операторы сравнения

В результате выполнения операторы сравнения (отношения) возвращают булевы значения TRUE или FALSE (табл. 11.10).

Таблица 11.10. Операторы сравнения

Оператор	Операция
<	Меньше
>	Больше
<=	Меньше или равно
>=	Больше или равно
=	Проверка равенства
<=>	Не равно, в том числе неопределенности NULL
!=, <>	Проверка неравенства

Внимание!

Осуществляя операции сравнения, учитывайте, что среди сравниваемых значений может оказаться неопределенность NULL.

Замечание

При желании в таблицу операторов сравнения можно добавить специфичные реляционные операторы IS NULL и IS NOT NULL, осуществляющие проверку равенства и неравенства NULL.

Проверка на неопределенность NULL

В диалектах SQL различных производителей обычно предусмотрено несколько способов проверки значения на неопределенность.

Синтаксис проверки выглядит следующим образом:

<значение> **IS** [**NOT**] **NULL**

Простой пример проверки предложен в листинге 11.8.

Листинг 11.8. Проверка на неопределенность с помощью IS NULL

```
SELECT NULL IS NULL, 1 IS NULL;
-> 1 0
```

Кроме того, часто встречается функция ISNULL(). Функция возвращает значение 1, если ее аргумент не определен (листинг 11.9).

Листинг 11.9. Проверка на неопределенность с помощью ISNULL

```
SELECT ISNULL(NULL), ISNULL(1);
-> 1 0
```

Конкатенация строк

Операция конкатенации (соединения) строк обычно осуществляется с помощью оператора «+», но, как всегда, существуют исключения. Например, в диалектах SQL, применяемых в InterBase и FireBird, строки соединяются оператором двойной вертикальной черты «||», например: TXT='Hello, '||'InterBase!';.

Встроенные функции

Классический SQL вооружен сравнительно небольшим набором встроенных функций (табл. 11.11).

Таблица 11.11. Основные функции SQL

Функция	Описание
BIT_LENGTH(битовая строка)	Функция позволяет выяснить длину строки в битах
CAST(значение AS тип данных)	Функция преобразования исходного значения к новому типу данных. Допустимые варианты конвертации типов данных представлены в табл. 11.7
CHAR_LENGTH(символьная строка)	Возвращает длину строки в символах
CURRENT_DATE	Функция возвращает текущую дату
CURRENT_TIME(точность)	Функция возвращает текущее время с определенной точностью
CURRENT_TIMESTAMP(точность)	Функция информирует о дате и времени с указанной точностью
LOWER(строка)	Преобразование текстовой строки к верхнему регистру
POSITION(подстрока IN строка)	Позволяет выяснить позицию, с которой начинается вхождение подстроки в строку
SUBSTRING(строка FROM n FOR длина)	Возвращает часть строки, начиная с <i>n</i> -го символа с указанной длиной

Функция	Описание
TRANSLATE(строка USING функция)	Преобразование строки с использованием указанной функции
TRIM(LEADING TRAILING BOTH символ FROM строка)	Удаление из строки всех первых (LEADING), последних (TRAILING) или первых и последних (BOTH) символов. Например: TRIM(LEADING '#' BOTH ' ###Москва') удалит все символы '#' из строки ' ###Москва'
UPPER(строка)	Преобразование всех символов строки к верхнему регистру

Резюме

Структурированный язык запросов SQL появился на свет вместе с реляционными базами данных в конце 70-х годов XX века. Это один из немногих языков, которому за очень короткий промежуток времени удалось добиться высокого статуса стандарта. На сегодняшний день действующим стандартом языка реляционных баз данных является SQL:2016, но, к сожалению, ни одна из имеющихся на рынке коммерческих СУБД не в состоянии похвастаться абсолютным соответствием не только SQL:2016, но и его предшественникам. Большинство компаний, работающих на рынке программного обеспечения, уже давно идут своим путем, развивая свои собственные диалекты SQL.

Неформально язык SQL можно разбить на несколько подязыков: подязык определения данных, манипулирования данными, ограничения доступа к данным, управления курсором и управления транзакциями.

Вопросы для самопроверки

1. Когда вышел первый стандарт языка SQL?
2. Какая версия стандарта SQL является актуальной на сегодня?
3. Для решения каких задач предназначен SQL?
4. Что имеется в виду, когда говорят, что SQL предназначен для работы с 3-значной логикой?
5. Какие достоинства и недостатки, на ваш взгляд, есть у SQL?

6. Дайте классификацию predefined типов данных в SQL.
7. Какие типы данных SQL предназначены для работы с:
 - a) текстом;
 - b) числовыми значениями;
 - c) датой и временем;
 - d) булевыми значениями;
 - e) большими объектами (например, файлами мультимедиа)?
8. Какие операторы поддерживает SQL?

Глава 12

Манипулирование данными SQL

В языке SQL наиболее широко востребованы инструкции, позволяющие просматривать и модифицировать данные. Эти инструкции зачастую выделяют в подязык манипулирования данными (Data Manipulation Language, DML). Язык DML в целом сформировался в SQL:92, и поэтому он большей частью поддерживается всеми современными СУБД, а отличия, неизбежно возникающие в разных диалектах SQL, в основном чисто косметические. Базис DML составляют:

- 1) оператор `SELECT`, отвечающий за выборку данных;
- 2) оператор `INSERT`, предназначенный для вставки в таблицу новых записей;
- 3) оператор `UPDATE`, позволяющий редактировать записи в таблице;
- 4) оператор `DELETE`, осуществляющий удаление записей.

Перечисленные инструкции могут применяться различными способами (которые в первую очередь определяются структурой таблицы) и охватывать одну, несколько или все строки таблицы одновременно.

Запрос, инструкция `SELECT`

Инструкции `SELECT` являются едва ли не самым важным элементом языка SQL, ведь именно для выбора данных изначально и создавался язык запросов (в переводе с англ. «select» означает «выбрать»). Исходным материалом для операций выборки выступают одна или несколько таблиц и представлений. Результатом выполнения SQL-запроса, основанного на инструкции `SELECT`, всегда будет отношение. В интерактивном режи-

ме оно просто выводится на экран компьютера, но при желании содержимое полученного отношения даже может быть отредактировано и сохранено (правда, не средствами SELECT).

Содержимое отношения, полученного в результате запроса, определяется программистом, в простейшем случае оно может полностью повторять исходные данные, может представлять собой объединение нескольких таблиц, может представлять собой определенную выборку из исходных данных. Ко всему прочему запросы SELECT активно используются при создании ряда ключевых объектов БД (представлений, хранимых процедур и триггеров).

Базовая синтаксическая конструкция запроса выглядит следующим образом:

SELECT [DISTINCT|ALL]

```
{ имя поля [AS псевдоним] [,...]
  |функция_агрегирования [AS псевдоним] [,...]
  |выражение для вычисления значения [AS псевдоним] [,...]
  |спецификатор.* }
```

FROM

```
{имя_таблицы [AS псевдоним] [,...]
  |имя_представления [AS псевдоним][,...]}
```

[WHERE условия отбора]

```
[GROUP BY имя поля [,...]] [HAVING условие]
[ORDER BY имя поля [ASC | DESC] [,...]]
```

Замечание

В предложенной конструкции не отражен порядок описания запроса, охватывающего несколько таблиц и представлений. Многотабличные запросы SELECT мы рассмотрим немного позднее.

Запросы на выборку данных всегда начинаются с ключевого слова SELECT, затем следует перечень столбцов, которые мы планируем увидеть в результате выполнения запроса. В перечень, кроме названий столбцов таблиц, могут входить агрегирующие функции, выражения и просто константы. За списком столбцов следует еще одно обязательное слово FROM, а за ним – имена таблиц или представлений, из которых будет производиться отбор данных.

В самом простейшем случае запрос должен включать в себя операторы SELECT и FROM, имя таблицы и спецификатор «*» (листинг 12.1).

Листинг 12.1. Выборка всех данных из таблицы

```
SELECT * FROM suppliers;
```

Символ «*» говорит о том, что в результирующее отношение попадут все столбцы из таблицы поставщиков `suppliers`. Это не всегда желательный способ организации запроса, так как выборка всех столбцов значительно повышает объем запрашиваемых данных и замедляет выполнение запроса. Поэтому рекомендуется быть более конкретным и указывать минимально необходимый набор столбцов, допустим так, как предложено в листинге 12.2.

Листинг 12.2. Выборка отдельного столбца из таблицы

```
SELECT supplier FROM suppliers;
```

Единственная строка кода потребует, чтобы в результирующем наборе данных присутствовал только один столбец с именем поставщика. Если понадобится выбрать несколько столбцов, то их имена разделяются запятыми (листинг 12.3).

Листинг 12.3. Выборка двух столбцов из таблицы

```
SELECT dnum, ddate FROM deliverynotes;
```

Если таблица содержит повторяющиеся строки, которые вы не хотите видеть в результирующем отношении, то сразу после инструкции `SELECT` следует вставить предикат `DISTINCT` (листинг 12.4).

Листинг 12.4. Исключение дубликатов строк

```
SELECT DISTINCT vendor FROM vendors;
```

Обратный предикат `ALL` указывает на то, что мы намерены получить все строки данных, включая и дубликаты. Впрочем, в явном применении `ALL` необходимости нет, так как это режим по умолчанию.

Замечание

В большинстве диалектов SQL с помощью инструкции `SELECT` можно вычислять значения без обращения к какой-либо таблице. Проверьте это утверждение, отправив на выполнение запрос `SELECT 2+2`.

Псевдонимы имен столбцов и таблиц

В тех случаях, когда запрос достаточно сложен и содержит длинные названия столбцов и таблиц, а также при использовании агрегирующих функций или столбцов, появившихся в результате выполнения выражений, программисты могут ввести псевдонимы столбцов и таблиц. Для этого применяется ключевое слово **AS** (листинг 12.5).

Листинг 12.5. Создание искусственного столбца с псевдонимом

```
SELECT CONCAT('№ ', dnum, ' от ', ddate) AS dinfo FROM deliverynotes;
```

В представленной выше строке запроса мы описываем выражение, объединяющее номер и дату накладной, и присваиваем этому столбцу псевдоним «dinfo». Обратите внимание на то, что в примере для конкатенации строк мы задействовали функцию `CONCAT()`. Однако это не догма, например в InterBase для соединения строк используют оператор «||».

В SQL предусмотрена возможность определения псевдонимов и для таблиц, в таком случае ключевое слово **AS** нам не понадобится – псевдоним просто указывается после названия таблицы (листинг 12.6). Псевдонимы таблиц позволяют сократить объем кода и способны повысить его наглядность.

Листинг 12.6. Создание псевдонимов для таблиц

```
SELECT * FROM goodslist g, transfers t  
WHERE t.goodslist_id=g.goodslist_id;
```

Запрос SQL может быть адресован сразу нескольким таблицам и даже таблицам, физически расположенным в разных базах данных. В таких случаях при перечислении столбцов перед собственно именем следует предварительно указывать имя БД и имя таблицы (листинг 12.7).

Листинг 12.7. Фрагмент инструкции SELECT с полным именем столбца таблицы

```
SELECT warehouse.goodlists.goods, warehouse_2.goodlists.amount, ...
```

Сразу после предложения **FROM** может следовать не только таблица, но и альтернативный источник данных, например представление.

Порядок сортировки, ORDER BY

Полученный в результате выполнения запроса набор строк может быть упорядочен. Порядок сортировки данных определяется при помощи команды `ORDER BY` и перечня столбцов, принимающих участие в сортировке. Пример, представленный в листинге 12.8, осуществит сортировку строк таблицы `vendors` по имени поставщика.

Листинг 12.8. Сортировка таблицы поставщиков по одному столбцу

```
SELECT vendor, phonenum FROM vendors ORDER BY vendor;
```

В порядке сортировки допускается указать сразу несколько столбцов, разделяя их запятыми.

По умолчанию записи упорядочиваются по возрастанию алфавита (от А до Я). Если необходимо осуществить сортировку в порядке убывания, то после имени столбца поставьте оператор `DESC` (листинг 12.9).

Листинг 12.9. Сортировка данных в порядке убывания даты

```
SELECT * FROM deliverynotes ORDER BY dndate DESC;
```

Вне зависимости от используемых в запросе предикатов и инструкций, фраза `ORDER BY` всегда должна быть последним элементом в предложении `SELECT`.

В запросах MySQL с помощью функции `INSTR` (возвращающей первую позицию вхождения подстроки в строку) можно создать достаточно экзотический способ сортировок – сортировку не с первых символов столбца, а начиная с символов вхождения подстроки. Допустим, необходимо сделать так, чтобы в верхнюю часть результатов выборки попали товары, в названиях которых встречается подстрока «ASUS». В таком случае нам поможет пример из листинга 12.10.

Листинг 12.10. Сортировка данных по вхождению подстроки в строку

```
SELECT * FROM goodslist ORDER BY INSTR(goods, 'ASUS') DESC;
```

Обратите внимание на то, что в примере после обращения к функции `INSTR()` нам пришлось применить оператор `DESC`, иначе строки, включающие «ASUS», были бы помещены в конец списка. Результат обычной сортировки и сортировки по подстроке представлен на рис. 12.1.

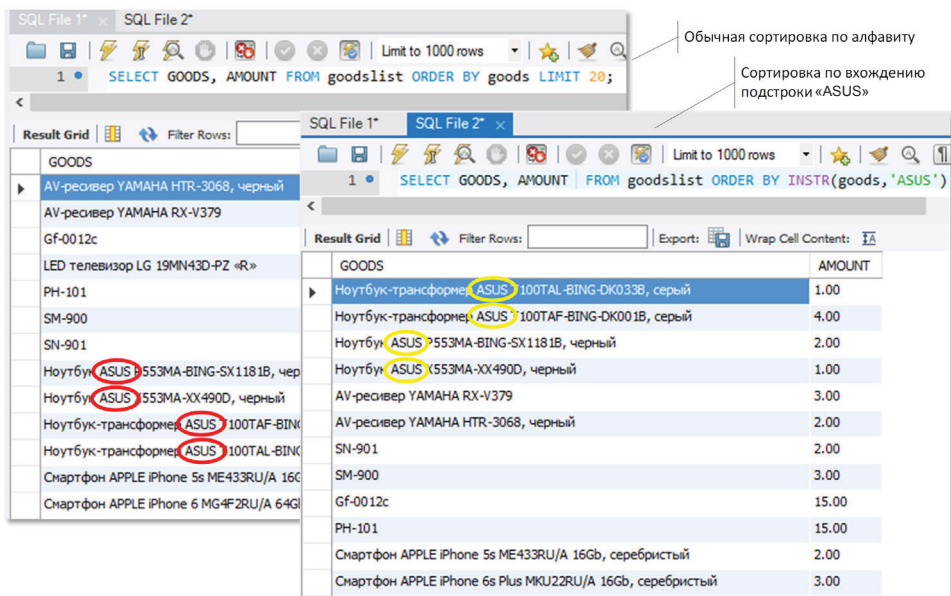


Рис. 12.1. Сравнение результатов сортировок в MySQL

Условие отбора данных, предложение WHERE

В результате выполнения запроса пользователь рассчитывает получить не все данные, а какую-то логическую выборку только необходимых в данный момент строк. Например, товары для кухни, товары, поступившие на склад за последний месяц, или товары, пользовавшиеся наибольшим спросом в предыдущем квартале. Для того чтобы ограничить набор выводимых данных, необходимо в конструкцию SELECT добавить предложение WHERE. Совместно с WHERE применяется следующий перечень типов ограничений.

1. Сравнение. Проводится сравнение результатов вычисления одного выражения с другим. Сравнения осуществляются с помощью операторов, представленных в табл. 11.10. Результат сравнения может принимать значения: TRUE, FALSE и UNKNOWN.
2. Попадание в диапазон. Проверяется, попадает ли результат вычисления выражения в определенный диапазон значений.
3. Соответствие шаблону. Проверяется, соответствует ли некоторое строковое значение заданному шаблону.
4. Неопределенность. Содержит ли поле неопределенное значение NULL.

5. Проверка существования. Проверяется факт существования значения в выходных результатах вложенных подзапросов к другим отношениям БД.

Все перечисленные ограничения описываются сразу после инструкции WHERE. В одном запросе допускается комбинировать несколько ограничений, с этой целью задействуются логические операторы AND, OR и NOT.

Сравнение

Сравнение – наиболее распространенное условие ограничения результирующего набора данных (листинг 12.11).

Листинг 12.11. Выборка строк с указанием минимальной даты

```
SELECT * FROM deliverynotes WHERE dndate>='01/01/2017';
```

Приведенное выражение выберет информацию о контрактах, заключенных начиная с 1 января 2017 года.

Другой вариант более сложного запроса (листинг 12.12) возвратит данные о товарах, габаритный размер которых не превышает 1 метра по осям X, Y и Z (при условии что в столбцах, отвечающих за хранение размеров, за единицу измерения принят миллиметр).

Листинг 12.12. Сложное условие выборки на основе AND

```
SELECT * FROM goodslist WHERE  
    SizeX<=1000 AND SizeY<=1000 AND SizeZ<=1000;
```

В листинге 12.13 для объединения трех условий мы использовали ключевое слово AND (логическое «И»). Кроме этого, в тексте SQL запроса можно использовать условие OR (логическое «ИЛИ»). Например, с помощью запроса из листинга 12.13 товаровед сможет узнать, есть ли на складе продукция с весом брутто более 100 Кг или габаритами более 1 кубометра.

Листинг 12.13. Сложное условие выборки на основе AND и OR

```
SELECT goodslist.*, (SizeX*SizeY*SizeZ) AS volume  
FROM goodslist  
WHERE amount>0 AND (grossweight>100 OR SizeX*SizeY*SizeZ>1000000000);
```

Замечание

Если необходимо построить запрос на основе комбинации нескольких условий AND и OR, то следует пользоваться круглыми скобками. Условия, заключенные в круглые скобки, должны выполняться совместно.

Благодаря тому что все текстовые символы идентифицируются их числовым кодом, вполне допустимо в условия отбора вводить сравнения с текстовыми данными (листинг 12.14).

Листинг 12.14. Сравнение с текстовым символом

```
SELECT supplier FROM suppliers WHERE supplier <'П';
```

В результате выполнения запроса мы получим список производителей, чьи названия начинаются с символа с кодом, меньшим, чем у «П».

Попадание в диапазон, BETWEEN

Предикат BETWEEN осуществляет контроль за тем, чтобы интересующий нас результат входил в определенный диапазон значений.

```
<значение> BETWEEN <начало диапазона> AND <окончание диапазона>
```

Или наоборот, чтобы результат не принадлежал определенному диапазону.

```
<значение> NOT BETWEEN <начало диапазона> AND <окончание диапазона>
```

Один из примеров применения BETWEEN представлен в листинге 12.15.

Листинг 12.15. Ограничения диапазона дат

```
SELECT * FROM deliverynotes WHERE dndate  
        BETWEEN '2018/01/01' AND '2018/12/31';
```

Пример вернет все контракты, заключенные в 2018 году начиная с первого января и заканчивая последним днем декабря.

Соответствие шаблону, LIKE

Если вы столкнулись со столь сложным условием выборки, что с ним оказываются не способны справиться ни STARTING WITH, ни CONTAINING, то обязательно обратите внимание на предикат LIKE, который определяет условия поиска в форме шаблона.

<значение> [NOT] LIKE <шаблон подстроки>
 [ESCAPE служебный символ подстановки]

Например, для выбора всех поставщиков, в имени которых есть символ «о», мы задействуем запрос из листинга 12.16.

Листинг 12.16. Поиск поставщиков, в имени которых есть символ «о»

```
SELECT * FROM suppliers WHERE supplier LIKE '%o%';
```

Результат выполнения представленного запроса проиллюстрирован на рис. 12.2.

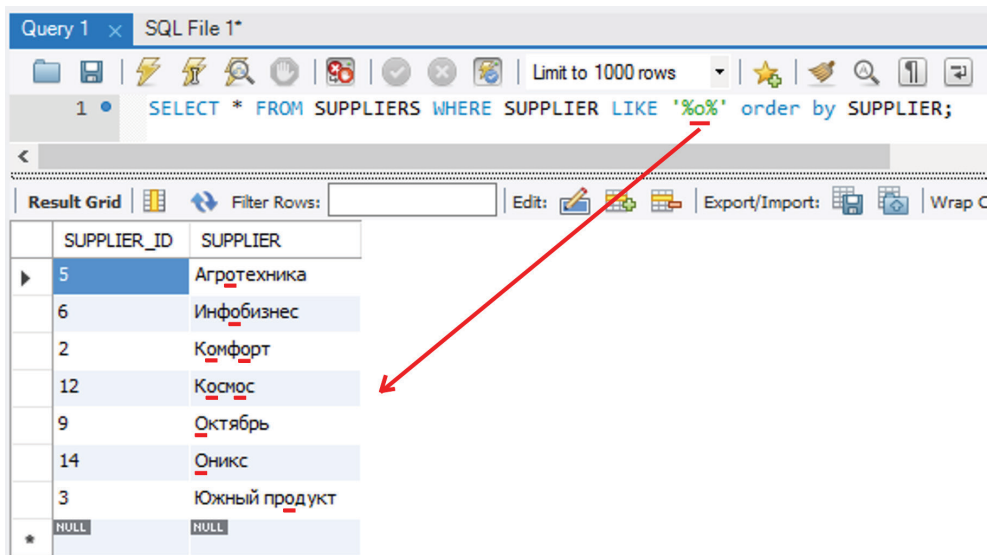


Рис. 12.2. Применение LIKE в запросе SELECT

В шаблоне подстроки, помимо набора интересующих нас символов, допускается применять два трафаретных символа:

- 1) символ подчеркивания «_», он используется вместо любого одиночного символа в выражении;
- 2) символ процента «%», заменяющий последовательность любых символов.

Например, для отбора всех поставщиков, у которых во второй позиции названия есть символ «к» и название заканчивается на «о», выражение должно выглядеть так: `'_к%о'`.

Замечание

Ряд диалектов SQL расширяет возможности SQL по проверке текстовых данных на соответствие шаблону, применяя механизм регулярных выражений (см. главу 15).

Выбор неопределенных или определенных значений

Нам уже знаком предикат `IS NULL`, предназначенный для проверки факта отсутствия или наличия значений в столбце отношения. Посмотрим, как он будет работать в предложении `WHERE`.

Пример из листинга 12.17 возвратит все записи, у которых в столбце `WEIGHT` отсутствует значение.

Листинг 12.17. Выборка строк с неопределенным значением в столбце

```
SELECT * FROM goodslist WHERE weight IS NULL;
```

Если вы, наоборот, не намерены выбрать строки, в которых имеется неопределенность, то запрос должен выглядеть иначе, чем в листинге 12.18.

Листинг 12.18. Выборка строк с определенным значением в столбце

```
SELECT * FROM deliveries WHERE amount IS NOT NULL;
```

Замечание

При проверке значения на неопределенность не стоит повторять хрестоматийную ошибку начинающих разработчиков БД – не используйте конструкции, подобные следующим: `WHERE X=NULL` или `WHERE X<>NULL`.

Вложенные запросы и проверка существования

Инструкция `SELECT` допускает вложение в запрос неограниченного количества подзапросов, благодаря которым можно создавать весьма сложные многоярусные конструкции выборки данных. Простейшая форма подзапроса предложена в примере 12.19.

Листинг 12.19. Подзапрос

```
SELECT * FROM SUPPLIERS WHERE SUPPLIER_ID=
    (SELECT SUPPLIER_ID FROM DELIVERYNOTES
     WHERE DNDATE BETWEEN '01.01.2018' and '31.01.2018');
```

Здесь мы пытаемся узнать данные поставщика, с которым был заключен договор на поставку продукции в январе 2018 года. Однако предложенный подход не пользуется популярностью у программистов, потому что предназначен для возврата всего одной строки.

Поэтому обычно вложенные подзапросы обладают более сложным синтаксисом и создаются на основе специальных предикатов: `IN`, `EXISTS`, `ALL`, `ANY`, `SINGULAR` и `SOME`. Благодаря перечисленным предикатам программист получает возможность проверить факт существования определенного значения в выходном наборе данных вложенного подзапроса.

Подзапрос `IN`

Оператор подзапроса `IN` производит проверку, входит ли результат вычисления в заданное множество. Множество может описываться как поэлементно, так и быть результатом другого, вложенного подзапроса.

<значение> [**NOT**] **IN** (подзапрос **SELECT**|элементы множества)

Например, в листинге 12.20 мы осуществляем выборку производителей продукции, соответствующих элементам заданного множества.

Листинг 12.20. Выборка строк, соответствующих элементам множества

```
SELECT * FROM VENDORS WHERE VENDOR IN ('Asus', 'Intel', 'IBM');
```

В основной запрос разрешено вкладывать подзапрос, он следует за инструкцией `WHERE`. Подзапрос описывается благодаря уже знакомому нам оператору `SELECT`. Допустим, что нам требуется узнать, имеются ли в таблице `VENDORS` производители, продукция которых никогда не поступала на склад. Решение поставленной задачи представлено в листинге 12.21.

Листинг 12.21. Пример вложения подзапроса с помощью предиката `IN`

```
SELECT * FROM VENDORS  
WHERE VENDORS_ID NOT IN (SELECT VENDORS_ID FROM GOODSLIST);
```

Предикат `IN` определяет, будут ли значения строки обнаружены в наборе значений, который либо определен, либо получен по результатам табличного подзапроса. Обратная задача решается при удалении ключевого слова `NOT`.

Представленный выше пример принадлежности ко множеству одновременно демонстрирует порядок создания вложенных подзапросов. Язык SQL не лимитирует количество вложений, поэтому механизм вложенных подзапросов является весьма мощным оружием при построении сложных отчетов.

Проверка существования EXISTS

Задача предиката EXISTS заключается в проверке факта существования строк, удовлетворяющих определенному критерию. Предикат может использоваться только совместно с подзапросами.

[NOT] EXISTS (SELECT * FROM <имя_таблицы> WHERE <условие отбора>)

Результатом выполнения предиката окажется не какой-то выходной набор данных, а просто булево значение TRUE или FALSE. Логическая истина получится в том случае, если в отобранном подзапросом наборе данных присутствует хотя бы одна запись. Ключевое слово NOT EXISTS использует обратные правила обработки.

Листинг 12.22 предлагает один из вариантов использования предиката EXISTS.

Листинг 12.22. Пример проверки существования

```
SELECT * FROM VENDORS WHERE EXISTS
(SELECT VENDORS_ID FROM GOODSLIST WHERE AMOUNT>0
AND GOODSLIST.VENDORS_ID=VENDORS.VENDORS_ID);
```

В рассмотренном примере мы получим список производителей, чьи товары имеются на нашем складе. Запрос будет выполнен очень быстро за счет того, что подзапрос выборки поставщиков не стремится пройти по всем строкам таблицы GOODSLIST, а просто находит первую соответствующую критерию поиска строку в запросе и возвращает TRUE.

Замечание

Основное достоинство предиката существования – высокая скорость выполнения подзапроса, и если вы хотите сформировать запрос, основанный на предположении о существовании каких-то данных, то EXISTS станет вашим лучшим помощником.

Многократное сравнение

При выполнении подзапроса IN проверяется равенство некоторого значения, обрабатываемого основным запросом какому-то из значений, содержащихся в столбце результатов вложенного запроса. В случае

совпадения данная запись включалась в результирующий набор запроса. В SQL предусмотрены операторы, осуществляющие более сложные правила сравнения, поддерживающие не только элементарную проверку на равенство, но и все остальные способы сравнения (табл. 11.10).

Предикат ALL

При включении предиката **ALL** в запрос следует руководствоваться следующим синтаксисом:

<сравниваемое значение> <оператор отношения> **ALL** (<подзапрос>)

Проверяемое значение поочередно сравнивается с каждым элементом, возвращаемым подзапросом, если все сравнения дают TRUE, то и вся проверка **ALL** возвратит TRUE, а если хотя бы один результат сравнения равен FALSE, то общим результатом также станет FALSE. Еще одна особенность проверки **ALL** заключается в том, что если вложенный подзапрос вернет пустой набор данных, то предикат **ALL** возвратит TRUE.

В качестве примера работы с предикатом **ALL** предлагаю рассмотреть листинг 12.23.

Листинг 12.23. Пример использования предиката ALL

```
SELECT * FROM SUPPLIERS WHERE
    (SUPPLIER_ID <> ALL (SELECT SUPPLIER_ID FROM DELIVERYNOTES WHERE
                        DNDATE BETWEEN '2018/11/01' AND '2018/11/30'))
ORDER BY SUPPLIER;
```

Задача предложенного запроса – выявить всех поставщиков, от которых не было поставок продукции в ноябре 2018 года.

Замечание

Оператор **ALL** в своем роде является эквивалентом логической операции **AND** («И»), он выдаст значение TRUE, если подзапрос не возвратит ни одной записи или, наоборот, когда все строки в наборе выдерживают сравнение. Соответственно, результат FALSE мы получим в случае, когда сравнение не выдержит хотя бы одна строка в наборе.

Предикат ANY (SOME)

Синтаксическая конструкция для предиката **ANY (SOME)** выглядит следующим образом:

<сравниваемое значение> <оператор отношения> **ANY | SOME** (<подзапрос>)

Замечание

Предикаты ANY и SOME эквивалентны и их поведение абсолютно одинаково.

Как и в случае с ALL, проверяемое значение последовательно сравнивается с элементами, возвращенными подзапросом. Если хотя бы одно из сравнений даст положительный результат, то проверка ANY (SOME) также завершится с результатом TRUE, иначе подзапрос просигнализирует значением FALSE. Если вложенный подзапрос вообще не возвратит данные, то сравнение ANY (SOME) заканчивается значением FALSE.

Порядок работы с предикатом ANY (SOME) представлен в листинге 12.24.

Листинг 12.24. Пример использования предиката ANY

```
SELECT * FROM SUPPLIERS WHERE
    (SUPPLIER_ID = ANY (SELECT SUPPLIER_ID FROM DELIVERYNOTES WHERE
                        DNDATE BETWEEN '2018/11/01' AND '2018/16/30'))
ORDER BY SUPPLIER;
```

В примере мы намерены выяснить, кто из поставщиков отгружал продукцию на наш склад в ноябре 2018 года.

Агрегирующие функции

Агрегирующие (обобщающие) функции в качестве исходных параметров принимают значения, указанные в запросе (после слова SELECT), и вычисляют результат. Как правило, эти функции применяются в запросах группировки с помощью команды GROUP BY. Наиболее распространенные функции представлены в табл. 12.1.

Таблица 12.1. Агрегирующие функции

Функция	Описание
COUNT	Возвращает количество строк. Тип возвращаемого значения – целое число. SELECT COUNT(*) FROM имя_таблицы;
AVG	Вычисляет среднее арифметическое для указанных элементов. Используется только для полей цифровых типов данных. Тип возвращаемого значения – вещественное число. SELECT AVG(имя_столбца) FROM имя_таблицы;

Функция	Описание
SUM	Вычисляет сумму значений. Применяется только для цифровых типов данных. SELECT SUM(имя_столбца) FROM имя_таблицы; Будьте внимательны. Если в результате вычисления суммы итоговое число превысит максимальное значение используемого типа данных, то возникнет ошибка
MAX	Возвращает наибольшее из всех значений. SELECT MAX(имя_столбца) FROM имя_таблицы;
MIN	Возвращает наименьшее из всех значений. SELECT MIN(имя_столбца) FROM имя_таблицы;

Замечание

Все агрегатные функции работают с единственным полем таблицы и возвращают единственное значение. Функции COUNT, MIN и MAX применимы как к числовым, так и к текстовым полям. Функции SUM и AVG сохраняют работоспособность только при обслуживании числовых полей. Совместно с перечисленными функциями разрешено применять квантификатор DISTINCT.

Группировка данных GROUP BY

Предложение GROUP BY предназначено для осуществления группировки выходных строк по какому-либо признаку, например по равенству значений каких-либо столбцов:

GROUP BY имя_столбца1 [,имя_столбца2, ...];

В группирующих запросах зачастую применяются рассмотренные чуть ранее в табл. 12.1 агрегирующие функции.

Замечание

В результате выполнения группирующего запроса для каждой отдельной группы создается одна-единственная группирующая строка.

Допустим, что мы хотим узнать, сколько всего единиц продукции каждого из наименований поступило на склад по накладной с определенным первичным ключом. Для этого нам следует обратиться к таб-

лице GOODSLIST и, используя агрегирующую функцию SUM(), построить группирующий запрос 12.25.

Листинг 12.25. Группировка по наименованию товара

```
SELECT GOODS, SUM(AMOUNT) AS SUM_AMOUNT FROM GOODSLIST
WHERE DELIVERYNOTE_ID=100
GROUP BY GOODS;
```

Обратите внимание на то, каким образом мы присвоили имя результату, возвращенному агрегирующей функцией, – для этого мы воспользовались предикатом **AS**. Еще одно замечание, касающееся особенностей построения группирующих запросов, – все имена столбца, приведенные в списке группирующего запроса SELECT, должны присутствовать и во фразе GROUP BY. Единственное исключение делается для полей, обрабатываемых агрегирующей функцией.

Дополнительная фильтрация группы строк, HAVING

Предикат HAVING используется после группировки (предложения GROUP BY) и предназначен для дополнительной фильтрации уже сформированных групп строк. Поведение элемента HAVING подобно предложению WHERE, но он работает не со всеми строками таблиц, а исключительно со строками результирующего набора. Фильтрация может осуществляться с помощью агрегирующих функций (COUNT, SUM, AVG, MAX и MIN). Для примера несколько модифицируем текст предыдущего запроса и научим его выводить информацию только о накладных с товарами, чья суммарная стоимость превышает 1000 (листинг 12.26).

Листинг 12.26. Применение предиката HAVING

```
SELECT GOODS, SUM(AMOUNT * FACTORYPRICE) AS SUM_PRICE FROM GOODSLIST
GROUP BY GOODS
HAVING SUM(AMOUNT * FACTORYPRICE)>1000;
```

Замечание

Условие отбора HAVING должно основываться на агрегирующей функции, если это не так, то вместо HAVING следует воспользоваться предложением WHERE.

Соединение таблиц в запросе SELECT

Рассмотренный в начале главы способ применения инструкции SELECT раскрывает только одну из сторон процесса выборки данных из

таблицы и представляет собой частный случай. Значительно чаще возникает необходимость объединения информационных потоков из двух и более таблиц. Для этого в рамках инструкции `SELECT` задействуют ряд дополнительных синтаксических структур, позволяющих работать одновременно с несколькими таблицами. В SQL предусмотрено три базовых способа построения запросов ко множеству таблиц (рис. 12.3).

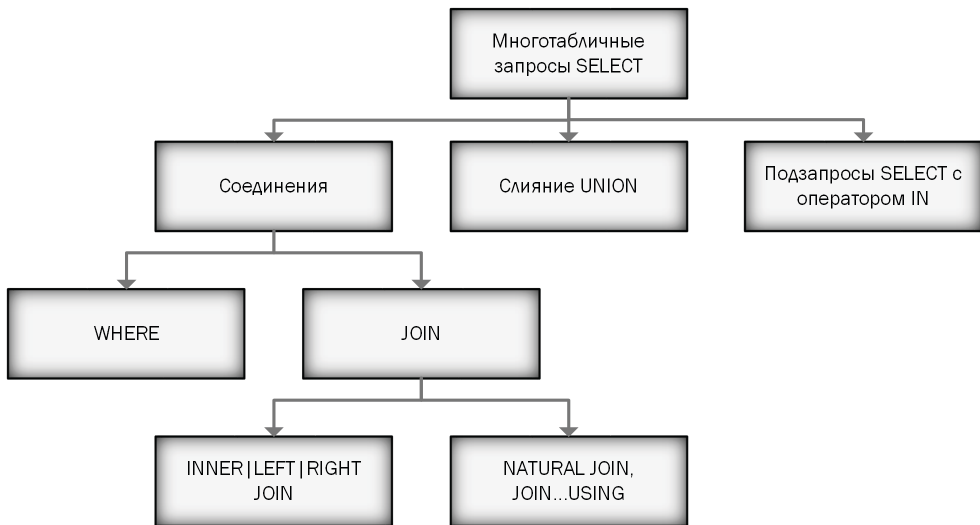


Рис. 12.3. Классификация многотабличных запросов в MySQL

Нам уже известен порядок построения подзапроса с применением оператора `IN`, этот способ выборки нам уже встречался (листинг 12.20). Так что нам предстоит сосредоточить свое внимание на соединениях, осуществляемых с помощью ключевых слов `WHERE`, `JOIN`, и слиянии таблиц `UNION`.

Внутреннее соединение `WHERE`

Простейшим способом соединения двух и более отношений является использование инструкции `WHERE` с определением столбцов, применяемых для соединения. Это один из самых старых способов объединения таблиц в реляционных базах данных – он появился на свет вместе с первым стандартом SQL.

Для выполнения объединения после ключевого слова `FROM` необходимо перечислить имена соединяемых таблиц, разделяя их запятыми. Например, мы хотим соединить таблицы поставщиков `SUPPLIERS` и таблицу договоров о поставке `DELIVERYNOTE` (листинг 12.27). Указанные таблицы

объединяются по ключевым полям SUPPLIER_ID (в таблице поставщиков это первичный ключ, в таблице договоров – внешний).

Листинг 12.27. Внутреннее соединение таблиц с помощью предложения WHERE

```
SELECT SUPPLIERS.SUPPLIER_ID, SUPPLIERS.SUPPLIER,  
DELIVERYNOTES.DELIVERYNOTE_ID, DELIVERYNOTES.DNNUM, DELIVERYNOTES.DNDATE  
FROM SUPPLIERS, DELIVERYNOTES  
WHERE DELIVERYNOTES.SUPPLIER_ID=SUPPLIERS.SUPPLIER_ID;
```

Тип представленного в листинге 12.27 соединения называют внутренним соединением, таблицы объединились на основе точного равенства между значениями, хранящимися в двух столбцах, – «внешний ключ = первичный ключ». Основная особенность внутреннего соединения в том, что из упомянутых в запросе таблиц в результирующее отношение попадут исключительно те строки, у которых имеются «напарники». Это проиллюстрировано рис. 12.4 – запрос возвратит только поставщиков «Салют» и «Радуга», ведь именно с ними были заключены контракты. Остальные упомянутые в таблице SUPPLIER поставщики в выходное отношение не попадают.

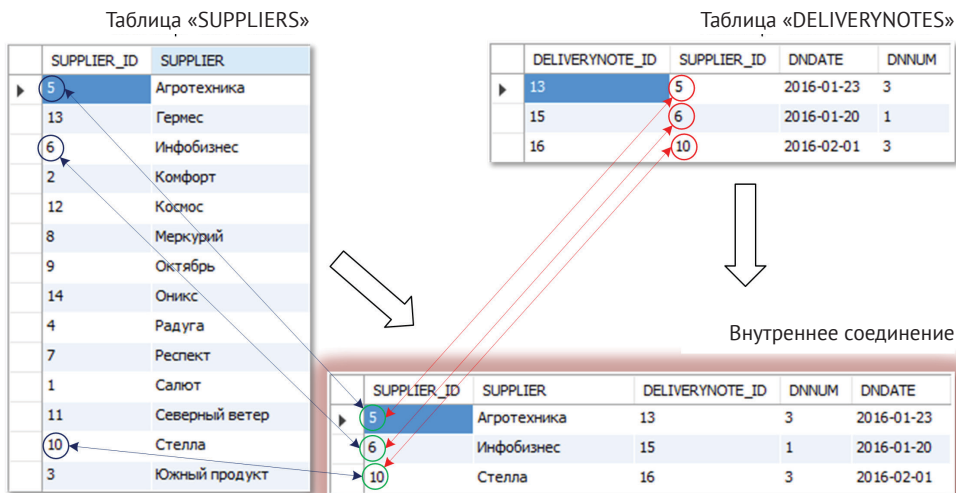


Рис. 12.4. Пример внутреннего соединения двух таблиц

Хотя метод построения многотабличных запросов, основанный на WHERE, поддерживается и сегодня, разработчику БД целесообразно отдавать предпочтение более совершенному (появившемуся вместе со стан-

дартом SQL-92) способу соединения таблиц с применением ключевого слова JOIN.

Соединение JOIN

С выходом стандарта SQL-92 вместо соединения с помощью предложения WHERE стали использовать синтаксические конструкции, опирающиеся на ключевое слово JOIN. В SQL таких конструкций несколько. Во-первых, предусмотрено классическое внутреннее соединение:

```
SELECT столбец1 [, столбец2 ...] | *
FROM <левая_таблица> [INNER | CROSS] JOIN <правая_таблица>
[ON <правило соединения>]
[WHERE <условие отбора>];
```

Как видите, в данном случае необходимо задействовать отдельный оператор JOIN или дополнять его необязательным ключевым словом INNER, подчеркивающим тот факт, что речь идет именно о внутреннем соединении. С помощью INNER JOIN можно добиться результата, абсолютно идентичного полученному ранее за счет WHERE (листинг 12.27). Для этого достаточно немного изменить структуру запроса SELECT (листинг 12.28).

Листинг 12.28. Внутреннее соединение таблиц с помощью INNER JOIN

```
SELECT Sp.SUPPLIER, Dn.*
FROM SUPPLIERS Sp
INNER JOIN DELIVERYNOTES Dn ON Dn.SUPPLIER_ID=Sp.SUPPLIER_ID
ORDER BY Sp.SUPPLIER;
```

Во-вторых, в SQL осуществлена поддержка так называемого внешнего соединения, на этот раз программисту следует явным образом указать, какого рода соединение он рассчитывает получить – левое (LEFT), правое (RIGHT).

```
SELECT столбец1 [, столбец2...] | *
FROM <левая_таблица> {LEFT | RIGHT} JOIN
    <правая_таблица> [ON <правило соединения>]
[WHERE <условие отбора>];
```

При написании кода запроса важную роль играет последовательность соединения таблиц, именно поэтому были введены понятия «левая» и «правая» таблицы. По правилам построения запроса JOIN имя левой

таблицы следует сразу после ключевого слова FROM, имя правой таблицы – после JOIN.

Замечание

Левое внешнее соединение LEFT JOIN возвратит все (даже несвязанные) строки левой таблицы и дополнит их связанными строками правой таблицы.

Рассмотрим листинг 12.29, в котором в качестве левой мы задействуем таблицу поставщиков SUPPLIER, а на роль правой назначим таблицу договоров DELIVERYNOTES на поставку товаров на склад.

Листинг 12.29. Левое соединение таблиц

```
SELECT Sp.SUPPLIER, Dn.*  
FROM SUPPLIERS Sp  
LEFT JOIN DELIVERYNOTES Dn ON Dn.SUPPLIER_ID=Sp.SUPPLIER_ID  
ORDER BY Sp.SUPPLIER;
```

Результат, возвращаемый запросом левого внешнего соединения, будет существенно отличаться от внутреннего. На этот раз (рис. 12.5) в выходное отношение оказались включены все строки из левой таблицы SUPPLIERS. И это произошло, даже несмотря на то что с большинством поставщиков пока не заключены контракты (об этом свидетельствуют многочисленные неопределенные значения NULL в столбцах DELIVERYNOTE_ID, DNNUM, DNDATE). С правой таблицей DELIVERYNOTES все обстоит по-прежнему – в результирующее отношение попали только строки, имеющие соответствие по ключу.

Порядок выполнения правого внешнего соединения RIGHT JOIN практически повторяет алгоритм левого, с той лишь разницей, что на этот раз выводу подлежат все записи из правой таблицы и только соответствующие им строки из левой. Пример использования RIGHT JOIN представлен в листинге 12.30.

Листинг 12.30. Правое соединение таблиц

```
SELECT G.GOODSLIST_ID, G.GOODS, G.AMOUNT, V.VENDORS_ID, V.VENDOR  
FROM GOODSLIST G  
RIGHT JOIN VENDORS V ON V.VENDORS_ID=G.VENDORS_ID;
```

В рассматриваемом примере объединятся таблицы производителей VENDORS и товаров GOODSLIST. При описании операции соединения RIGHT JOIN таблица производителей объявляется правой, поэтому в итоговое

отношение попадут все имеющиеся в ней строки. Что касается левой таблицы GOODSLIST, то на этот раз в выборку попадут только записи, имеющие связь со строками из правой таблицы.

Левая таблица «SUPPLIERS»

SUPPLIER_ID	SUPPLIER
5	Агротехника
13	Гермес
6	Инфобизнес
2	Комфорт
12	Космос
8	Меркурий
9	Октябрь
14	Ониск
4	Радуга
7	Ресpekt
1	Салют
11	Северный
10	Стелла
3	Южный продукт

Правая таблица «DELIVERYNOTES»

DELIVERYNOTE_ID	SUPPLIER_ID	DNDATE	DNNUM
19	6	2016-01-18	1
20	14	2016-01-19	2
21	6	2016-01-20	3
22	5	2016-01-23	4
24	10	2016-02-01	5
26	10	2016-02-01	6

SUPPLIER	DELIVERYNOTE_ID	SUPPLIER_ID	DNDATE	DNNUM
Агротехника	22	5	2016-01-23	4
Гермес	NULL	NULL	NULL	NULL
Инфобизнес	21	6	2016-01-20	3
Инфобизнес	19	6	2016-01-18	1
Комфорт	NULL	NULL	NULL	NULL
Космос	NULL	NULL	NULL	NULL
Меркурий	NULL	NULL	NULL	NULL
Октябрь	NULL	NULL	NULL	NULL
Ониск	20	14	2016-01-19	2
Радуга	NULL	NULL	NULL	NULL
Ресpekt	NULL	NULL	NULL	NULL
Салют	NULL	NULL	NULL	NULL
Северный ветер	NULL	NULL	NULL	NULL
Стелла	24	10	2016-02-01	5
Стелла	26	10	2016-02-01	6
Южный продукт	NULL	NULL	NULL	NULL

Результат левого внешнего соединения

Рис. 12.5. Пример левого внешнего соединения двух таблиц**Замечание**

В ряде СУБД поддерживается полное (FULL JOIN) объединение, которое возвращает все строки из левой и правой таблиц.

Соединение NATURAL JOIN и JOIN ... USING

Стандарт SQL постоянно совершенствуется, предлагая разработчикам СУБД вектор развития структурированного языка запросов. По рекомендациям стандарта появились дополнительные синтаксические конструкции JOIN:

- NATURAL JOIN – естественное соединение;
- JOIN ... USING – соединение с явным указанием столбца.

Суть конструкции естественного соединения `NATURAL JOIN` заключается в том, что `SQL` самостоятельно выбирает способ соединения таблиц. В первую очередь проверяются одноименные столбцы в левой и правой таблицах если таковые отсутствуют, то соединение осуществляется по однотипным столбцам (как вы понимаете, в последнем случае результат, скорее всего, не будет соответствовать вашим ожиданиям).

Автор книги при проектировании связей между таблицами всегда в подключаемой таблице создает столбец внешнего ключа с тем же именем, что и у столбца первичного ключа в главной таблице. Поэтому проблем с натуральным соединением (листинг 12.31) в демонстрационной базе данных не возникнет.

Листинг 12.31. Натуральное соединение таблиц `NATURAL JOIN`

```
SELECT * FROM suppliers NATURAL JOIN deliverynotes  
WHERE dndate BETWEEN '01.01.2016' AND '31.01.2016';
```

Результат выполнения натурального соединения иллюстрирует рис. 12.6. Для сравнения на нем, кроме экранного снимка `NATURAL JOIN`, предложен снимок запроса `INNER JOIN`, как видите, результаты идентичны – натуральное соединение отработало корректно.

Соединение с явным указанием столбца `JOIN ... USING` можно рассматривать как доработку естественного соединения `NATURAL JOIN`. В данном случае исключается неверная трактовка имен соединяемых столбцов (листинг 12.32).

Листинг 12.32. Соединение таблиц `JOIN ... USING`

```
SELECT * FROM suppliers  
JOIN deliverynotes USING (supplier_id)  
WHERE dndate BETWEEN '01.01.2016' AND '31.01.2016';
```

И вновь мы получаем корректное итоговое отношение, это утверждение подтвердит рис. 12.6.

Замечание

По своей сути соединения `NATURAL JOIN` и `JOIN ... USING` выступают просто сервисными расширениями традиционных конструкций `[INNER | LEFT | RIGHT] JOIN` и предназначены для упрощения труда программиста только в случае, если при проектировании таблиц строго соблюдались правила именования столбцов первичного и внешнего ключей.

INNER JOIN

```

1 select * from suppliers s
2 inner join deliverynotes d on s.supplier_id=d.supplier_id
3 where dndate between '01.01.2016' and '31.01.2016';

```

SUPPLIER_ID	SUPPLIER	DELIVERYNOTE_ID	SUPPLIER_ID	DNDATE	DNNUM
6	Инфобизнес	19	6	2016-01-18	1
14	Онекс	20	14	2016-01-19	2
6	Инфобизнес	21	6	2016-01-20	3

NATURAL JOIN

```

1 select * from suppliers natural join deliverynotes
2 where dndate between '01.01.2016' and '31.01.2016';

```

SUPPLIER_ID	SUPPLIER	DELIVERYNOTE_ID	DNDATE	DNNUM
6	Инфобизнес	19	2016-01-18	1
14	Онекс	20	2016-01-19	2
6	Инфобизнес	21	2016-01-20	3

JOIN USING

```

1 select * from suppliers
2 join deliverynotes using (supplier_id)
3 where dndate between '01.01.2016' and '31.01.2016';

```

SUPPLIER_ID	SUPPLIER	DELIVERYNOTE_ID	DNDATE	DNNUM
6	Инфобизнес	19	2016-01-18	1
14	Онекс	20	2016-01-19	2
6	Инфобизнес	21	2016-01-20	3
5	Агротехника	22	2016-01-23	4
10	Стелла	24	2016-02-01	5
10	Стелла	26	2016-02-01	6

Рис. 12.6. Сравнение результатов внутреннего и натурального соединения двух таблиц

Соединение большого количества таблиц

При необходимости в запросе `SELECT` теоретически допускается соединять бесконечное число таблиц. В нашем примере мы рассмотрим вариант запроса SQL, формирующий единое отношение из 7 таблиц демонстрационной БД (листинг 12.33).

Листинг 12.33. Внутреннее объединение нескольких таблиц

```

SELECT * FROM GOODSLIST GL
JOIN GOODSCCLASS GC ON GC.GOODSCCLASS_ID=GL.GOODSCCLASS_ID
JOIN VENDORS V ON GL.VENDORS_ID=V.VENDORS_ID
JOIN DELIVERYNOTES D ON D.DELIVERYNOTE_ID=GL.DELIVERYNOTE_ID
JOIN SUPPLIERS S ON S.SUPPLIER_ID=D.SUPPLIER_ID
JOIN TRANSFERS T ON T.GOODSLIST_ID=GL.GOODSLIST_ID
JOIN WAREHOUSES W ON W.WAREHOUSES_ID=T.WAREHOUSES_ID;

```

Внимание!

Порядок объединения таблиц имеет критическое значение для скорости выполнения запроса. Если таблицы соединяются в правильном порядке, то общее число обрабатываемых строк будет меньше. Описывая запрос, следует выполнять сначала максимально ограничивающий поиск, чтобы отфильтровать как можно большее число строк на ранних фазах выполнения запроса с соединениями.

Запросы к иерархическим данным

Соединение JOIN как нельзя лучше подходит для отбора строк из таблиц, содержащих иерархические данные, построенные по принципу родитель–потомок. В подобных таблицах дерево формируется за счет того, что поле внешнего ключа строки-потомка ссылается на поле первичного ключа строки-родителя. В демонстрационной БД у нас имеется одна такая таблица GOODSCLASS (приложение 1), она специализируется на хранении классификаторов товаров, находящихся на складе.

Во время проектирования запроса SELECT нам придется пойти на хитрость и сделать вид, что обработке подвергаются данные не из одной, а из двух (при желании и более двух) таблиц. Для этого нам понадобится задать для одной и той же таблицы два псевдонима (листинг 12.34). Получив разные псевдонимы, СУБД станет трактовать их как ссылки на две различные таблицы и создаст два отдельных курсора для их обслуживания.

Листинг 12.34. Инструкция JOIN при работе с иерархическими таблицами

```
SELECT C1.GOODSCLASS_ID AS ID1, C1.PARENT_ID AS PARENT_ID1,
       C1.GOODSCLASS,
       C2.GOODSCLASS_ID AS ID2, C2.PARENT_ID AS PARENT_ID2,
       C2.GOODSCLASS AS GOODSCLASS2
FROM GOODSCLASS C1
LEFT JOIN GOODSCLASS C2 ON C1.GOODSCLASS_ID=C2.PARENT_ID
WHERE C2.GOODSCLASS IS NOT NULL
ORDER BY C1.NODEINDEX, C2.NODEINDEX;
```

Представленный в листинге 12.34 пример позволяет обрабатывать дерево с неограниченным числом уровней, но при работе с возвращаемым запросом отношением следует учитывать ряд особенностей.

1. У всех родительских записей самого верхнего уровня значение родительского ключа не определено (состояние NULL).

2. В каждой строке результирующего отношения отражаются сведения только об узлах уровней N и $N - 1$. Например (рис. 12.7), для дочерних узлов третьего уровня «Газовая плита» и «Электрическая плита» выведен родительский узел уровня 2 «Плиты».

ID1	PARENT_ID1	GOODSCLASS	ID2	PARENT_ID2	GOODSCLASS2
3	NULL	Компьютеры	15	3	Ноутбук
3	NULL	Компьютеры	16	3	Комплекующие
9	NULL	Кухонная техника	10	9	Холодильное оборудование
9	NULL	Кухонная техника	24	9	Вытяжки
9	NULL	Кухонная техника	29	9	Водонагреватели
9	NULL	Кухонная техника	30	9	Плиты
10	9	Холодильное оборудов...	11	10	Холодильники
10	9	Холодильное оборудов...	12	10	Морозильные камеры
30	9	Плиты	31	30	Газовая плита
30	9	Плиты	32	30	Электрическая плита
21	NULL	Фото и видео	22	21	Фотоаппараты
21	NULL	Фото и видео	23	21	Видеокамеры
25	NULL	Контроль за климатом	26	25	Кондиционеры
25	NULL	Контроль за климатом	27	25	Сплит-системы

Рис. 12.7. Результаты запроса к иерархическим данным

Слияние UNION

Слияние необходимо в случае, когда требуется объединить результаты нескольких запросов в одну результирующую таблицу. Для этих целей предназначено выражение UNION.

```
SELECT ...
UNION [ALL | DISTINCT] SELECT ...
[UNION [ALL | DISTINCT] SELECT ...]
```

Допустим, что у нас имеется две идентичные по структуре таблицы VENDORS_OLD и VENDORS, хранящие сведения о производителях товаров. Для того чтобы получить суммарную информацию, стоит воспользоваться оператором UNION (листинг 12.35).

Листинг 12.35. Слияние UNION

```
SELECT * FROM VENDORS_OLD
UNION
SELECT * FROM VENDORS;
```


Замечание

Слияние UNION может осуществляться только у совместимых по объединению отношений.

Допустимо объединять и разные отношения, но при условии совпадения количества и типа данных объединяемых столбцов.

По умолчанию запрос UNION удаляет повторяющиеся строки из результирующего отношения (этому режиму соответствует необязательный ключ DISTINCT), однако если их следует сохранить, то стоит позвать на помощь предикат ALL (листинг 12.36).

Листинг 12.36. Слияние UNION с сохранением дубликатов

```
SELECT * FROM VENDORS_OLD
UNION ALL
SELECT * FROM VENDORS;
```

Внимание!

При слиянии таблиц с помощью UNION из результирующего отношения по умолчанию исключаются строки-дубликаты.

Вставка, инструкция INSERT

Назначение инструкции INSERT – вставка в таблицу одной или нескольких строк. Практически все диалекты SQL на все 100 % поддерживают синтаксическую конструкцию, рекомендованную стандартом.

```
INSERT INTO {имя таблицы | имя представления}
{имя столбца[, ...]}
{DEFAULT VALUES | VALUES (Значение1[, ...]) | инструкция SELECT}
```

Имя пополняемой таблицы указывается после ключевых слов INSERT INTO. С помощью INSERT допускается вставка новой записи и в представление, но при условии, что представление обслуживает одну таблицу. Сразу за именем таблицы следует перечень столбцов, в которые мы собираемся внести новую информацию. Если список столбцов окажется неполным, то в оставшиеся поля записывается значение по умолчанию. Если, в свою очередь, значение по умолчанию не было определено заранее (на этапе создания таблицы) и на столбец наложено ограничение NOT NULL, то работа оператора прерывается. Еще одним поводом для приостановки операции вставки новой записи может послужить попытка

поместить данные в столбец, который самостоятельно генерирует новое значение (например, в столбцы автоинкрементного типа).

Рассмотрим наиболее распространенную форму применения инструкции `INSERT INTO`, в которой нам требуется добавить новую строку в таблицу производителей `VENDORS` (листинг 12.37).

Листинг 12.37. Вставка новой строки с помощью `INSERT...INTO`

```
INSERT INTO VENDORS  
(VENDOR, PHONENUM, EMAIL, COUNTRY_ID)  
VALUES  
( 'SAMSUNG', '88002000001', 'support@SAMSUNG.com', 12 );
```

В результате таблица производителей товаров пополнится новой записью. Заметьте, что в тексте запроса отсутствует обязательное значение для столбца первичного ключа таблицы. Дело в том, что задача вставки значения первичного ключа автоинкрементного типа обычно решается автоматически.

Замечание

В команде на добавление новой строки перечисление столбцов не является обязательным, если вы осуществляете вставку данных во все столбцы. Но в этом случае последовательность передаваемых значений должна четко соответствовать физическому порядку столбцов таблицы.

При реализации команды `INSERT` некоторые разработчики СУБД расширили стандартные возможности SQL. Например, MySQL и DB2 позволяют за одно обращение к команде `INSERT` вставлять в таблицу несколько строк (листинг 12.38).

Листинг 12.38. Вставка нескольких строк с помощью `INSERT...INTO`

```
INSERT INTO COUNTRYS  
(COUNTRY)  
VALUES  
( 'Австрия' ), ( 'Бельгия' ), ( 'Южная Корея' ), ( 'Япония' );
```

Такое решение весьма удобно, особенно когда требуется добавить все строки в таблицу в рамках единственной транзакции.

Если вместо ключевого слова `VALUES` в инструкции на вставку новой записи применить выражение `DEFAULT VALUES`, то в столбцах таблицы

окажутся значения, определенные по умолчанию, в оставшихся полях (не имеющих определения DEFAULT) вы обнаружите неопределенные значения NULL.

Внимание!

Инструкция INSERT совместно с выражением DEFAULT VALUES часто служит источником ошибок, например в ситуации, когда таблица содержит столбцы внешних ключей (FOREIGN KEY) или столбцы, не допускающие неуникальных значений.

Стандартом SQL:2003 рекомендуется производителям доработать свои диалекты SQL так, чтобы они позволяли вставлять в целевую таблицу строки, извлекаемые инструкцией SELECT из другой таблицы. Пример решения этой задачи раскрывает листинг 12.39.

Листинг 12.39. Вставка записей INSERT...SELECT

```
INSERT INTO SUPPLIERS
(SUPPLIER)
(SELECT SUPPLIER FROM SUPPLIERS_2);
```

Условием корректности выполнения такой команды должно быть соответствие типов столбцов целевой и донорской таблиц.

Внимание!

С операцией вставки новой строки в таблицу могут быть связаны триггеры BEFORE INSERT и AFTER INSERT.

Модификация, инструкция UPDATE

Вам никогда не хотелось все исправить? Если да, то сейчас мы поговорим о самой главной команде в нашей с вами жизни – инструкции редактирования данных UPDATE.

```
UPDATE {имя таблицы | имя представления}
SET {{имя_столбца=
    выражение | значение | NULL | DEFAULT | ARRAY | ROW=строка}[,...]}
[WHERE условие отбора | WHERE CURRENT OF имя курсора]
```

Минимальный код, позволяющий за один присест изменить все записи в таблице, выглядит так, как представлено в листинге 12.40.

Листинг 12.40. Модификация значения столбца для всех строк таблицы

```
UPDATE VENDORS SET EMAIL='VENDOR@MAILOPERATOR.COM';
```

Предложенную строку вряд ли стоит когда-нибудь применять в реальных проектах. Почему? Хотя бы потому, что она одним махом поменяет адреса электронной почты всех производителей товара на адрес `VENDOR@MAILOPERATOR.COM`. Как вы понимаете, в данном случае такой поступок не вполне корректен.

Неоспоримое удобство инструкции `UPDATE` заключается в возможности практически мгновенного изменения данных не только в отдельной, но и в нескольких строках или даже во всей таблице. Допустим, что существует некая таблица `SALES`, в одном из полей которой хранится значение заработной платы сотрудников компании. Представленный пример разработан специально для тех, кто хочет попробовать проявить себя на поприще благотворительности (листинг 12.41).

Листинг 12.41. Удвоение зарплаты

```
UPDATE SALES SET SALARY=SALARY*2 WHERE DEPARTMENT_ID=1;
```

Благодаря единственной строке кода мы умудрились повысить зарплату всем сотрудникам отдела, идентифицированного ключом `DEPARTMENT_ID=1`, ровно в 2 раза. Если после этого вас не уволят, то, скорее всего, вы не программист, а руководитель предприятия...

Если мы ошиблись очень сильно и нам необходимо одновременно внести изменения в несколько полей, то следует разделять пары «имя поля = значение» запятыми (листинг 12.42).

Листинг 12.42. Исправление значений в двух столбцах таблицы

```
UPDATE GOODSLIST  
SET WEIGHT=100, GROSSWEIGHT=110  
WHERE GOODSLIST_ID=2567;
```

При присваивании нового значения столбцу с помощью предложения `SET` мы можем не только передавать в столбец константу или результат выражения, но и значение по умолчанию, для этого надо воспользоваться ключевым словом `DEFAULT`.

Замечание

Во многих СУБД редактирование данных осуществляется следующим образом: строка со старой информацией помечается как удаленная, а в таблицу вставляется

новая строка, с обновленными данными. А для того чтобы не нарушить целостность данных, в эту запись переносится старое значение первичного ключа.

Удаление, инструкция DELETE

Инструкция DELETE специализируется на удалении строк из таблиц данных.

```
DELETE FROM имя_таблицы  
[{WHERE предикат}]  
| {WHERE CURRENT OF имя_курсора}
```

Замечание!

В большинстве СУБД, в том числе и у MySQL, операция удаления не приводит к немедленному физическому стиранию строки. Вместо этого специальный служебный бит удаления строки помечается соответствующим образом, после чего сервер начинает полагать, что строки не существует.

Минимальная команда включает инструкцию DELETE FROM и имя таблицы (листинг 12.43).

Листинг 12.43. Удаление всех строк из таблицы

```
DELETE FROM ARTICLE
```

Такая команда полностью очистит таблицу, что в обычных случаях не применяется, так как скорее напоминает вредительство. Для того чтобы осуществить адресное удаление строк, необходимо после предиката WHERE описать условие удаления (листинг 12.44).

Листинг 12.44. Составное условие удаления строк из таблицы

```
DELETE FROM DELIVERYNOTES  
WHERE DNNUM>100 AND DNDATE BETWEEN '2017-01-10' AND '2017-01-15';
```

Как видите, условия удаления могут быть достаточно сложными, в данном примере мы удалили только те записи из таблицы, которые хранят сведения о договорах с номером больше 100 и были заключены в январе 2017 года.

Благодаря возможностям WHERE можно создавать весьма неординарные конструкции по очистке таблиц от излишних данных. Хорошим

подтверждением этого утверждения может стать часто встречающаяся задача удаления повторяющихся значений из таблицы. Смоделируем такую ситуацию в таблице поставщиков SUPPLIERS. Допустим, что по нашему недосмотру таблица пополнилась строками с дублирующимися названиями. Это не просто неприятная новость, а в первую очередь потенциальная угроза для бизнес-логики и целостности данных БД, поэтому от дубликатов следует немедленно избавиться.

Для решения задач подобного рода следует вспомнить о возможностях группировки данных (листинг 12.45).

Листинг 12.45. Удаляем дубликаты

```
DELETE FROM SUPPLIERS
      WHERE SUPPLIERS_ID NOT IN
      (SELECT MIN(SUPPLIERS_ID) FROM SUPPLIERS GROUP BY SUPPLIER);
```

Вся соль этого примера сосредоточена в последней строке кода, содержащей вложенный подзапрос SELECT. Здесь вы ввели условие группировки по полю PUBLISHER, содержащему повторяющиеся значения с условием вывода только записей с минимальным значением первичного ключа. Как следствие подзапрос отбросит все дубликаты данных. Остальное – дело техники, во второй строке инструкции SQL мы определяем, что удалению подлежат все записи, не попавшие в список группировки, для этого нам понадобилось выражение NOT IN.

Вариант удаления с предложением WHERE CURRENT OF предназначен для осуществления позиционного удаления. Удаляется строка, на которой находится курсор (листинг 12.46).

Листинг 12.46. Удаляем строку курсора

```
DELETE FROM SUPPLIERS WHERE CURRENT OF SUPPLIERS_CURSOR
```

Заострим внимание читателя на том, что предложенный пример приобретает функциональность только при работе с курсором (см. главу 14).

Слияние данных, инструкция MERGE

Инструкция MERGE развивает возможности базовых команд INSERT, UPDATE и DELETE. Она предоставляет возможность выполнять операции вставки, редактирования и удаления данных над целевой таблицей, основываясь на некотором наборе правил, которые определяются в результате сравнения строк целевой и исходной таблиц.

Стандартом SQL рекомендована следующая простейшая синтаксическая конструкция обращения к инструкции:

```
MERGE INTO <целевая_таблица> [ [ AS ] <псевдоним_слияния> ]  
USING <исходная_таблица>  
ON <условия_совпадения>  
    <спецификация_операции_слияния>
```

Ключевое слово **INTO** определяет таблицу (представление), в которую будут вставлены (обновлены или удалены) строки. Необязательный псевдоним результатов слияния следует за словом **AS**. Имя исходной таблицы (представления или выражения), из которой будут браться строки, сопоставляемые с исходной таблицей, должно следовать после **USING**. После **ON** задаются условия, которые должны применяться для определения факта совпадения строк в целевой и исходной таблицах.

Спецификация операции слияния задается с помощью выражений:

- **WHEN MATCHED THEN** <спецификация слияния обновлением>, указывающего на то, что запись целевой таблицы будет изменена, если существует соответствие между строкой целевой и исходной таблиц;
- **WHEN NOT MATCHED THEN** <спецификация слияния вставкой>, указывающего на то, что запись будет добавлена в целевую таблицу, если сопоставимая строка не найдена;
- **WHEN NOT MATCHED BY SOURCE THEN** <спецификация>, определяющего операцию редактирования или удаления над строками, которые присутствуют в целевой таблице и отсутствуют в исходной.

Резюме

В этой главе мы познакомились с основными инструкциями, составляющими основу языка SQL, точнее его подмножества – языка манипулирования данными (Data Manipulation Language).

Запросы на выборку данных из таблиц и представлений строятся на основе инструкции **SELECT**. Инструкция **SELECT** позволяет отбирать только часть данных в соответствии с правилами, задаваемыми пользователем, сортировать и группировать полученный набор данных. Условия отбора данных строятся с помощью сравнения, попадания в диапазон, соответствия шаблону, принадлежности множеству и другими способами. Условия отбора могут усложняться за счет применения логических

операций. В результате выполнения запроса пользователь получает отношение, содержащее ноль и более строк.

Базовые синтаксические конструкции, связанные с операциями вставки INSERT, редактирования UPDATE и удаления DELETE записей, позволяют нам модифицировать данные в таблицах.

Основные инструкции DML поддерживаются всеми диалектами SQL, благодаря чему обеспечивается совместимость СУБД различных производителей.

Вопросы для самопроверки

1. Объясните назначение следующих предложений, входящих в инструкцию SELECT:
 - a) ORDER BY;
 - b) GROUP BY;
 - c) WHERE.
2. Что позволяет сделать ключевое слово DISTINCT?
3. В чем отличие между предложениями WHERE и HAVING?
4. Для чего применяются агрегирующие функции?
5. Каким образом в запросе можно обработать значения NULL?
6. Что такое многократное сравнение, какие операторы для этого применяются?
7. Каким образом можно соединить 2 (и более) таблицы в SQL?
8. Объясните отличия внутреннего, левого, правого и внешних соединений.
9. Каким образом можно создать подзапрос в инструкции SELECT?
10. Каким образом с помощью SQL можно обратиться к иерархическим данным?
11. Что понимается под слиянием двух отношений, и какая инструкция SQL для этого понадобится?
12. Раскройте порядок применения инструкции INSERT.
13. Почему не следует применять инструкции UPDATE и DELETE без предложения WHERE?
14. Для чего предназначена инструкция MERGE?

Глава 13

Определение данных средствами SQL

В этой главе мы изучим порядок создания, изменения и удаления основных объектов базы данных средствами подязыка определения данных (Data Definition Language, DDL). Без знания DDL немыслима работа разработчика баз данных, ведь этот подязык позволяет создавать, редактировать и удалять основные объекты БД, такие как схемы, домены, таблицы и т. д.

Весь подязык DDL опирается на три команды: CREATE, ALTER и DROP, – отвечающие соответственно за создание, изменение и удаление объекта БД.

При изучении инструкций определения данных, как всегда, за основу будет взят стандарт SQL, вместе с тем мы постараемся раскрыть возможности наиболее популярных диалектов языка SQL современных СУБД.

Базы данных (схемы)

В SQL под **схемой** понимается поименованная коллекция связанных между собой объектов базы данных. Определение схемы приводит к созданию собственно физической базы данных, так что с точки зрения SQL между понятиями «схема» и «база данных» можно поставить знак равенства.

Для управления БД в структурированном языке запросов SQL предусмотрено три базовые инструкции:

- CREATE DATABASE – создание нового экземпляра БД;
- ALTER DATABASE – модификация существующего экземпляра БД;
- DROP DATABASE – удаление БД.

Перечисленные инструкции в том или ином виде вы можете встретить практически во всех современных СУБД. В Informix Universal Server, MySQL, SQL Server, PostgreSQL используется инструкция CREATE DATABASE. Изначально в СУБД DB2 и Oracle схема данных формировалась по умолчанию во время развертывания сервера, и все создаваемые таблицы принадлежали одной схеме. Позднее Oracle отказался от идеи общей схемы и стал позволять создавать отдельные экземпляры БД, для этого в состав диалекта SQL этой СУБД была введена инструкция CREATE DATABASE.

Для получения полного представления о создании БД рассмотрим несколько примеров создания новой БД «WAREHOUSE» на различных диалектах SQL. Популярные СУБД MySQL и PostgreSQL удовлетворятся одной строкой.

Листинг 13.1. Инструкция для создания БД в MySQL

```
CREATE DATABASE 'WAREHOUSE';
```

При работе с СУБД InterBase (Firebird) допускается воспользоваться следующим примером (листинг 13.2).

Листинг 13.2. Инструкция для создания БД в InterBase

```
CREATE DATABASE 'd:\data\warehouse.ib'  
USER 'SYSDBA' PASSWORD 'masterkey';
```

В простейшем случае для SQL Server компании Microsoft также достаточно передать инструкцию CREATE DATABASE и имя создаваемой БД, но для создания новой базы с настройками файлов данных и журнала транзакций одной-двух строк кода явно недостаточно (листинг 13.3).

Листинг 13.3. Инструкция для создания БД в SQL Server

```
CREATE DATABASE WAREHOUSE  
ON  
    (NAME = warehouse_dat,  
    FILENAME = 'c:\data\warehouse.mdf',  
    SIZE = 50MB,  
    MAXSIZE = 200MB,  
    FILEGROWTH = 5MB)  
LOG ON
```

```
(NAME = warehouse_log,
  FILENAME = 'c:\data\warehouse.ldf',
  SIZE = 10MB,
  MAXSIZE = 50MB,
  FILEGROWTH = 5MB)
```

В этом примере мы определили не только параметры БД, но и журнала транзакций.

Несмотря на всеобщее признание команды `CREATE DATABASE` производителями СУБД, в стандарте SQL она в явном виде отсутствует, а вместо нее предусмотрена следующая синтаксическая конструкция:

```
CREATE SCHEMA <имя_схемы> [AUTHORIZATION имя владельца]
  [DEFAULT CHARACTERSET набор символов по умолчанию]
  [PATH <символьный путь к файлам БД> ]
  [ <дополнительные инструкции>... ]
```

В минимальной нотации достаточно только указания названия будущей БД. При необходимости можно определить путь к файлам создаваемой схемы и задать дополнительные опции, связанные с вопросами авторизации и кодировкой символов (если последняя должна отличаться от назначенной по умолчанию).

```
CREATE SCHEMA 'WAREHOUSE' AUTHORIZATION 'DBAdmin';
```

В большинстве СУБД предусмотрена возможность перестроения схемы. Такая операция может понадобиться при переносе БД, изменении размеров БД, перенастройке параметров сортировки и в ряде других случаев. Обычно в подобном случае следует обращаться к команде `ALTER SCHEMA` или `ALTER DATABASE`.

Возможностей у `ALTER DATABASE` немного, например в MySQL можно изменить кодовую страницу символов (листинг 13.4).

Листинг 13.4. Изменение кодовой страницы для БД в MySQL

```
ALTER SCHEMA 'warehouse' DEFAULT CHARACTER SET cp1251;
```

В СУБД, поддерживающих шифрование данных, `ALTER DATABASE` обычно позволяет изменить правила и ключи шифрования.

Процесс уничтожения схемы особым изыском не отличается. Практически во всех диалектах SQL имеется инструкция:

```
DROP DATABASE <имя_базы_данных>;
```

В свою очередь, стандарт SQL опирается на команду DROP SCHEMA:

```
DROP SCHEMA <имя_схемы> [CASCADE|RESTRICT];
```

По умолчанию (ключевое слово RESTRICT) инструкция полагает, что удалению подлежит БД, не содержащая никаких объектов, и если это не так, то выполнение команды отменяется. Если следует обеспечить удаление схемы с каскадным уничтожением всех принадлежащих ей таблиц, представлений и других объектов, используем RESTRICT.

Домены

Если вы видите, что для корректного описания допустимых значений, передаваемых в то или иное поле таблицы, возможностей типа данных явным образом не хватает, то это признак того, что пора воспользоваться услугами доменов. Если типизация отвечает за физический формат хранения данных, то в дополнение к этому домен позволяет накладывать дополнительные логические ограничения на значения, передаваемые в столбец таблицы. Поэтому механизм доменов существенно обогащает наши возможности по определению ограничений.

Для описания домена SQL предусматривает следующую конструкцию:

```
CREATE DOMAIN <имя_домена> [ AS ] <предопределенный тип данных>
[ <значение по умолчанию> ]
[ <ограничения домена>[,...] ]
[ <правила сравнения> ]
```

При определении доменных правил огромную пользу может принести грамотное применение инструкции CHECK – благодаря ей программист сможет описать ограничения, накладываемые на вводимые пользователем значения. Допустим, что нам следует создать домен, ограничивающий ввод значений диапазоном от 1 до 100 и по умолчанию принимающий значение 1, тогда следует написать строки, предложенные в листинге 13.5.

Листинг 13.5. Домен-диапазон

```
CREATE DOMAIN DOMAIN_SMALLRANGE AS INTEGER
DEFAULT 1
CHECK (VALUE BETWEEN 1 AND 100);
```

Созданный домен сохраняется в системном каталоге БД, поэтому его можно применять при определении полей множества таблиц.

Например, в InterBase благодаря инструкции `DEFAULT` (листинг 13.6) можно научить домен помещать в поле таблицы текущую дату [40].

Листинг 13.6. Текущая дата как значение по умолчанию в InterBase

```
CREATE DOMAIN D_DATE
AS DATE
DEFAULT current_date;
```

Еще одно преимущество домена в том, что при необходимости изменить ограничения во всех таблицах достаточно перенастроить всего один домен с помощью инструкции `ALTER DOMAIN`. Для удаления домена следует воспользоваться инструкцией `DROP DOMAIN`.

Таблицы

После создания базы данных и доменов можно переходить к созданию основного объекта реляционной БД – таблиц. Рассмотрим средства подязыка DDL, нацеленные на решение этой задачи:

```
CREATE [ {GLOBAL | LOCAL} TEMPORARY] TABLE имя_таблицы
({определение поля | [ограничение таблицы]}., ...
[ON COMMIT {DELETE | PRESERVE} ROWS]):
определение поля ::= имя_поля {имя домена | тип данных [размер]}
                                [ограничение поля...]
[DEFAULT значение по умолчанию]
[COLLATE имя сравнения]
```

Минимальная конструкция, позволяющая создать простейшую таблицу, должна включать операторы `CREATE TABLE`, имя создаваемой таблицы и перечень столбцов с их типами и размерами. В результате мы получим пустую таблицу. В следующем примере (листинг 13.7) создается таблица `PEOPLES`, включающая пять столбцов. Первый столбец целочисленный, второй, третий и четвертый столбцы – символьные, и последний столбец предназначен для хранения даты.

Листинг 13.7. Создание простейшей таблицы

```
CREATE TABLE PEOPLES
(IndNum int,
```

```
Surname char(30),
FName char(20),
LName char(20),
BDay date);
```

В таблице должно быть одно или несколько определений столбцов. Определение столбца обязательно должно включать имя, тип данных и при необходимости размер. В различных диалектах SQL могут различаться названия типов столбцов и формат хранения данных (например, даты и времени). По умолчанию атрибуты таблицы допускают помещение в них значения NULL.

В соответствии со стандартом SQL для таблицы можно задать одно или несколько ограничений (табл. 13.1).

Таблица 13.1. Ограничения столбцов таблицы

Ключ	Описание
NOT NULL	Запрет на вставку в столбец неопределенного значения NULL
UNIQUE	Значение столбца должно быть уникальным
PRIMARY KEY	Признак первичного ключа. Значение поля должно быть уникальным, оно не может содержать NULL, в таблице это ограничение может использоваться только один раз
CHECK	Ограничение-проверка на допустимое значение. В скобках за оператором CHECK указывается предикат, проверяющий допустимость значения
FOREIGN KEY и REFERENCES	Оба ограничения весьма похожи, единственное различие между ними в том, что FOREIGN KEY – ограничение внешнего ключа для таблицы, а REFERENCES – ссылка на столбец со значениями подстановки. В случае установки ограничения для таблицы сразу за FOREIGN KEY в скобках необходимо указать перечень столбцов, относящихся к ключу. В остальном синтаксис одинаков. Ограничения внешнего ключа требуют, чтобы все значения, присутствующие во внешнем ключе, соответствовали значениям родительского ключа (обеспечение ссылочной целостности)

В любой реляционной таблице должна существовать возможность уникальной идентификации отдельной строки, для этой цели предназначен первичный ключ. Для включения столбца в состав первичного ключа используется ключевое слово PRIMARY KEY. Продемонстрируем это на примере таблицы поставщиков SUPPLIERS (листинг 13.8).

Листинг 13.8. Создание таблицы поставщиков

```
CREATE TABLE SUPPLIERS (
  SUPPLIER_ID int NOT NULL AUTO_INCREMENT,
  SUPPLIER varchar(100) NOT NULL,
  PRIMARY KEY (SUPPLIER_ID),
  UNIQUE KEY SUPPLIER_UNIQUE (SUPPLIER)
);
```

В представленном примере на роль первичного ключа назначен столбец SUPPLIER_ID.

Внешние ключи и связи между таблицами

В реляционных базах данных возникающее между таблицами отношение создается посредством ограничения внешнего ключа. Данное ограничение обеспечивает правило существования строк в подчиненной таблице, защищая ее от попыток вставить записи, несовместимые с выбранной моделью.

Изучите рис. 13.1, на котором представлены таблицы поставщиков SUPPLIERS и приходных накладных DELIVERYNOTES. Каким образом реализовать поддержку ссылочной целостности для таблицы DELIVERYNOTE на языке SQL?

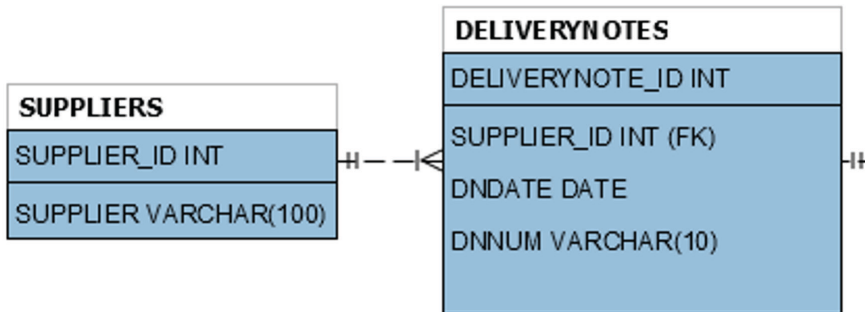


Рис. 13.1. Связь между таблицами

Минимальная конструкция определения внешнего ключа в большинстве диалектов SQL выглядит примерно так:

```
FOREIGN KEY (имя_столбца,...)
REFERENCES имя_таблицы (имя_столбца, ...)
[ON DELETE {CASCADE | SET NULL | NO ACTION | RESTRICT | SET DEFAULT}]
[ON UPDATE {CASCADE | SET NULL | NO ACTION | RESTRICT | SET DEFAULT}]
```

Первое и самое главное условие для построения связи между таблицами заключается в наличии однотипных столбцов первичного ключа в родительской таблице и внешнего ключа в дочерней. Раз в нашем примере в качестве первичного ключа во всех таблицах задействуется целочисленный тип, то столбцы внешнего ключа также должны быть целочисленными. В противном случае связь окажется создать невозможно.

Второе условие – столбцы первичного ключа главной таблицы и соответствующие им столбцы внешнего ключа подчиненной таблицы должны быть проиндексированы. В случае первичного ключа индексация производится автоматически, нам достаточно только поставить признак **PRIMARY KEY**. В случае внешнего ключа индекс создается в момент обработки инструкции **FOREIGN KEY**.

Затем в соответствии с требованиями минимальной нотации к описанию связи сразу после **FOREIGN KEY** следует указать имя столбца (или столбцов) внешнего ключа дочерней таблицы. Имя главной таблицы и название столбца первичного ключа следуют после ключевого слова **REFERENCES** (листинг 13.9).

Листинг 13.9. Создание таблицы с внешним ключом

```
CREATE TABLE DELIVERYNOTES (  
    DELIVERYNOTE_ID int NOT NULL AUTO_INCREMENT,  
    SUPPLIER_ID int NOT NULL,  
    DNDATE date NOT NULL,  
    DNNUM varchar(10) NOT NULL,  
    PRIMARY KEY (DELIVERYNOTE_ID),  
    FOREIGN KEY (SUPPLIER_ID) REFERENCES SUPPLIERS (SUPPLIER_ID)  
);
```

Таким образом, за счет применения **FOREIGN KEY** и **REFERENCES** в инструкции создания таблицы накладных **DELIVERYNOTES** мы реализовали отношение 1:М (один поставщик – много накладных) между таблицами точно так, как предложено в ER-модели.

Расширенный синтаксис позволяет в момент описания связи настроить правила ссылочной целостности, которые определяют поведение дочерней таблицы при изменении (**ON UPDATE**) или удалении (**ON DELETE**) связанных данных в родительской. В момент изменения значения первичного ключа и в момент удаления значения первичного ключа у родительской таблицы сервер осуществляет контроль ссылочной целостности данных. Если при создании внешнего ключа в

инструкции SQL мы никак не конкретизировали поведение механизма контроля целостности, то они по умолчанию устанавливаются в режим RESTRICT, остальные доступные варианты работы триггеров отражены в табл. 13.2.

Таблица 13.2. Поведение дочерней таблицы при изменении/удалении значения РК

Действие	Описание
CASCADE	Изменение значения первичного ключа приводит к автоматическому изменению соответствующих значений внешнего ключа
SET NULL	При изменении значения или удалении первичного ключа все значения в связанных с внешним ключом колонках устанавливаются в NULL. В результате в дочерней таблице появляются «брошенные» строки, утерявшие связь с соответствующей записью из главной таблицы
RESTRICT	Режим по умолчанию. Внешний ключ воспринимает только значения первичного ключа или NULL. Попытки поместить в него какие-либо другие значения будут отвергаться
NO ACTION	То же самое, что и RESTRICT

Ограничения на значения столбцов

Установка значения по умолчанию производится при помощи оператора DEFAULT. Таким образом обеспечивается автоматический ввод в столбец некоторого значения, если его явным образом не укажет пользователь в момент вставки новых данных (листинг 13.10).

Листинг 13.10. Создание таблицы с простейшими ограничениями

```
CREATE TABLE TABLE1
(COLUMN1 INT DEFAULT 0,
 COLUMN2 FLOAT NOT NULL,
 COLUMN3 NUMERIC(9,2), CHECK (COLUMN3>-100 AND COLUMN3<100),
 COLUMN4 FLOAT NOT NULL, CHECK (COLUMN4<COLUMN2));
```

В этих же строках вы найдете оператор CHECK, позволяющий наложить определенные ограничения на вводимые в поле данные. Благодаря CHECK третий столбец принимает только значения из диапазона от –100 до 100, а четвертый столбец таблицы должен содержать число, не превышающее значение, хранимое во втором столбце.

Листинг 13.11 демонстрирует еще один аспект применения значений по умолчанию в InterBase (FireBird).

Листинг 13.11. Столбец со значением по умолчанию

```
CREATE TABLE DELIVERYNOTES (
    DELIVERYNOTE_ID INTEGER PRIMARY KEY,
    ...
    INSERTEDBY VARCHAR(50) DEFAULT CURRENT_USER)
```

В представленном примере таблица DELIVERYNOTES дополнена столбцом INSERTEDBY, в который по умолчанию будет передаваться содержимое контекстной переменной CURRENT_USER, в которой InterBase хранит имя текущего пользователя. Такое решение позволит нам контролировать, кто именно вносил в таблицу данные об очередном заказе.

Внимание!

В большинстве СУБД DEFAULT гарантирует попадание назначенного разработчиком БД значения в заданный столбец только в одном случае – при осуществлении операции вставки новой записи (инструкция INSERT). Это произойдет в ситуации, когда пользователь не предложил взамен неопределенности NULL никакого значения. Если же запись уже существует и подвергается редактированию (инструкция UPDATE), то пользователь запросто сможет отправить в столбец NULL. Это замечание следует учитывать при проектировании таблиц и при необходимости строить еще один эшелон обороны от некорректности данных, например добавляя ограничение NOT NULL или описывая дополнительные правила на уровне триггеров.

Столбец-перечисление

В ряде СУБД в качестве целевого типа данных для столбца может выступать перечисление – тип данных, обозначаемый ключевым словом ENUM. Перечисления очень полезны в ситуации, когда в столбец должно заноситься одно значение из заранее подготовленного перечня (например, день недели (листинг 13.12), название месяца года и т. п.).

Листинг 13.12. Определение столбца-перечисления

```
CREATE TABLE workcalendar (
    ...,
    taskname VARCHAR(40),
```

```
weekday ENUM('Понедельник', 'Вторник', 'Среда', 'Четверг', 'Пятница', 'Суббота',
             'Воскресенье')
);
```

После создания таблицы основанный на перечислении столбец будет контролировать передаваемые в него значения и не допускать попадания в ячейку таблицы некорректного (не входящего в перечень) значения, единственное допустимое исключение отводится неопределенности NULL.

Столбец-множество

Ряд СУБД поддерживает размещение в столбце таблицы множества. Как правило, в состав множества может входить не более 64 элементов.

Пример определения столбца-множества предложен в листинге 13.13, в котором создается таблица с расписанием работы магазинов. На роль множества здесь назначен столбец `workday` – в нем хранятся названия рабочих дней недели.

Листинг 13.13. Определение столбца-множества в MySQL

```
CREATE TABLE shopcalendar (
shopcalendar_id int NOT NULL AUTO_INCREMENT,
shopname VARCHAR(100) NOT NULL,
workday SET('Понедельник', 'Вторник', 'Среда', 'Четверг', 'Пятница', 'Суббота',
            'Воскресенье'),
openat TIME NOT NULL DEFAULT '8:00:00',
closeat TIME NOT NULL DEFAULT '22:00:00',
PRIMARY KEY ('shopcalendar_id')
);
```

Физически значения в столбце-множестве `workday` хранятся не в текстовом виде, а в формате бинарной последовательности. В нашем примере, в котором множество представляет собой дни недели, битовая последовательность включает 7 бит. Если ни один из дней не входит во множество – последовательность будет состоять только из нулей. Если во множество войдет только понедельник, то мы получим 0000001, только вторник – 0000010, только среда – 0000100, понедельник и вторник – 0000011, все дни недели – 1111111 и т. п.

Хотя мы еще не рассматривали инструкцию INSERT для полноты картины работы со столбцом-множеством, приведем листинг 13.14, демонстрирующий вставку новой строки.

Листинг 13.14. Вставка новой строки с данными о множестве

```
INSERT INTO shopcalendar
(shopname, workday, openat, closeat)
VALUES
('Антей', 'Понедельник, Вторник, Среда, Четверг, Пятница', '09:00:00', '20:00:00');
```

Обратите внимание на то, что направляемые в столбец-множество данные дней недели передаются в виде текстовой строки, а значения в ней разделяются запятыми и нет никаких пробелов!

Временные таблицы

Зачастую при обработке данных программисты баз данных нуждаются в создании временных хранилищ данных, в которых будут находиться какие-то промежуточные или сводные результаты. Одним из методов решений таких задач является использование временных таблиц. Временные таблицы существуют только на время текущего сеанса SQL и по окончании его автоматически удаляются.

При создании временной таблицы требуется воспользоваться ключевым словом **TEMPORARY** (листинг 13.15). В зависимости от того, создаем мы глобальные или локальные временные таблицы, можно (но не обязательно) использовать ключевые слова **GLOBAL** или **LOCAL** соответственно.

Листинг 13.15. Создание глобальной временной таблицы

```
CREATE GLOBAL TEMPORARY TABLE TmpTable
(IntValue INT, FloatValue FLOAT);
```

Замечание

Таблицы базы данных должны обладать уникальными именами, однако это требование не актуально для временных таблиц. Дело в том, что временная таблица видима только создавшему её пользователю, поэтому если другой пользователь в рамках своей сессии доступа к БД создаст одноименную временную таблицу, конфликта имен не произойдет.

Замечание

Строго говоря, создаваемая инструкцией **CREATE GLOBAL TEMPORARY TABLE** таблица столь же «временная», как и все остальные таблицы БД. Просто команда **GLOBAL TEMPORARY** конкретизирует, что она специализируется на хранении временных для некоторой сессии данных. Для физического удаления такой временной таблицы точно так же можно воспользоваться выражением **DROP TABLE**.

Модификация таблицы

Единожды созданная таблица не является статичным объектом, в SQL предусмотрена возможность модификации структуры уже существующих таблиц. Для этой цели реализована инструкция `ALTER TABLE`:

```
ALTER TABLE имя_таблицы
{ADD[COLUMN] определение столбца}
| {ALTER [COLUMN] имя изменяющегося столбца}
| {DROP [COLUMN] имя_столбца RESTRICT | CASCADE}
| {ADD определение ограничения для таблицы}
| {DROP CONSTRAINT имя_ограничения RESTRICT | CASCADE}
```

Особенности синтаксической конструкции рассмотрены в табл. 13.3.

Таблица 13.3. Описание операторов команды `ALTER TABLE`

Оператор	Описание
<code>ADD [COLUMN]</code>	Добавляет в таблицу новый столбец. Новый столбец определяется так же, как в операторе <code>CREATE TABLE</code>
<code>ALTER [COLUMN]</code>	Используется для создания или отмены значения по умолчанию для столбца
<code>DROP [COLUMN]</code>	Удаляет из таблицы столбец. При использовании параметра <code>RESTRICT</code> перед удалением столбца СУБД проверит наличие ссылок на него из других таблиц и представлений. Если таковые имеются, то столбец не будет удален. Наоборот, при использовании параметра <code>CASCADE</code> вместе со столбцом будут удалены все объекты, ссылающиеся на него
<code>ADD</code>	Позволяет добавить к таблице новое ограничение
<code>DROP CONSTRAINT</code>	Удаляет уже существующие ограничения. Если в предложении задан параметр <code>RESTRICT</code> , то в этот момент столбец не должен использоваться как родительский ключ для внешнего ключа другой таблицы. Если передается параметр <code>CASCADE</code> , то внешние ключи, имеющие ссылки или ограничения <code>FOREIGN KEY</code> , уничтожаются

Для добавления к таблице еще одного столбца воспользуйтесь следующим примером:

```
ALTER TABLE ORDERS ADD COLUMN ORDERSPRICE INT;
```

Для изменения типа уже существующего столбца:

```
ALTER TABLE ORDERS ALTER COLUMN ORDERSPRICE FLOAT;
```

И наконец, для удаления столбца:

```
ALTER TABLE ORDERS DROP COLUMN ORDERSPRICE;
```

Внимание!

Изменяя структуру таблицы, уже заполненную данными, следует проявлять особую осторожность, в особенности при удалении столбцов или изменении их параметров. В любом случае, дабы избежать случайной потери важных данных, перед обращением к инструкции `ALTER TABLE` следует создать резервную копию всей БД.

Клонирование и копирование таблиц

Практически все современные диалекты SQL позволяют копировать таблицу (с тем же набором столбцов), как и исходную. В СУБД Oracle, PostgreSQL, MySQL для этих целей следует написать команду, предложенную в листинге 13.16.

Листинг 13.16. Копируем структуру таблицы

```
CREATE TABLE NEW_COUNTRYIS SELECT * FROM COUNTRYIS WHERE COUNTRYIS_ID=-1;
```

Благодаря помощи `SELECT` мы получили точно такую таблицу, как и справочная таблица `COUNTRYIS`. Чтобы итоговая таблица оказалась пустой, мы задали заранее невыполнимое условие соответствия первичного ключа значению `-1`.

Положительный эффект повторения структуры таблицы можно многократно усилить, воспользовавшись огромным потенциалом инструкции `SELECT`. Следующий пример кода (листинг 13.17) демонстрирует работу Transact-SQL, создающего временную таблицу на основе объединяющего запроса.

Листинг 13.17. Создание таблицы на основе двух таблиц в SQL Server

```
SELECT SUPPLIERS.SUPPLIER, DELIVERYNOTES.DNDATE, DELIVERYNOTES.DNNUM  
INTO #NEWTABLE  
FROM SUPPLIERS  
INNER JOIN CONTRACTS ON SUPPLIERS.SUPPLIER_ID = DELIVERYNOTES.SUPPLIER_ID  
ORDER BY DELIVERYNOTES.DNDATE, DELIVERYNOTES.DNNUM;
```

В результате мы получаем таблицу, содержащую упорядоченные по дате и номеру накладной данные о поставщиках.

Индексы

Не станем останавливаться на теоретическом аспекте индексирования, он нам уже хорошо знаком по главе 9. Посему сразу перейдем к практической стороне вопроса – изучению возможностей SQL по работе с индексами.

Система индексирования используется во всех СУБД без исключения, но в стандартном SQL синтаксическая конструкция создания индексов отсутствует. Вместе с тем диалекты SQL большинства производителей весьма схожи и придерживаются примерно следующей упрощенной конструкции:

```
CREATE [UNIQUE] [ASCENDING|DESCENDING] INDEX имя_индекса
ON имя_таблицы (имя_столбца [, ... n]);
```

В случае создания индекса с опцией **UNIQUE** БД прекращает допускать ввод повторяющихся значений в столбец (столбцы), включенный в определение индекса (листинг 13.18).

Листинг 13.18. Индекс для одного столбца таблицы с проверкой уникальности

```
CREATE UNIQUE INDEX ind_SUPPLIER ON SUPPLIERS(SUPPLIER);
```

Довольно часто приходится создавать составные индексы, то есть индексы, состоящие из двух и более столбцов таблицы (листинг 13.19).

Листинг 13.19. Индекс для нескольких столбцов таблицы

```
CREATE INDEX ind_VENDOR ON VENDORS (vendor, email);
```

В последнем примере индекс строится по столбцам с именем и электронным адресом производителя товаров.

Отметим одну из интересных особенностей построения индекса для текстовых данных в MySQL. Эта СУБД с целью экономии системных ресурсов умеет индексировать лишь часть строкового значения. Например, в листинге 13.20 предложен пример индексирования лишь первых 20 символов из столбца **EMAIL**.

Листинг 13.20. Индекс для первых 20 символов столбца **EMAIL** в MySQL

```
CREATE INDEX ind_vendors_email ON vendors (EMAIL(20));
```

Ряд СУБД (Oracle, SQL Server) способен создавать особый вид индекса – кластерный (clustered). Кластерный индекс – специальный индекс, заставляющий СУБД хранить данные в таблицах именно в том порядке, который задает этот индекс. Использование кластерного индекса способствует значительному сокращению времени доступа к данным, поэтому в таких СУБД основной индекс таблицы всегда стоит определять CLUSTERED (листинг 13.21).

Листинг 13.21. Кластерный индекс для SQL Server

```
CREATE CLUSTERED INDEX ind_COUNTRY ON COUNTRY(COUNTRY);
```

В таблице может быть назначен только один кластерный индекс. Кластерный индекс вместо хранения простого ключевого значения, ссылающегося на данные в таблице, хранит целую строку.

Замечание

В кластерный индекс включаются именно те столбцы и именно в таком порядке, который соответствует наиболее часто используемому представлению таблицы.

Допускается создавать индексы, осуществляющие сортировку данных в обратной последовательности, например в SQL Server, InterBase, FireBird это делается при помощи ключа DESCENDING (листинг 13.22).

Листинг 13.22. Индекс по убыванию значений для SQL Server

```
CREATE DESCENDING INDEX ind_SUPPLIER_DESC ON SUPPLIERS(SUPPLIER);
```

В MySQL аналогичная задача решается немного по-другому (листинг 13.23).

Листинг 13.23. Индекс по убыванию значений для MySQL

```
CREATE INDEX ind_SUPPLIER_DESC ON SUPPLIERS(SUPPLIER DESC);
```

Изменение индекса

Перенастройка индекса производится инструкцией ALTER INDEX. Особенности команды зависят от эксплуатируемой СУБД, например InterBase, FireBird, SQL Server, Oracle умеют отключать индекс (листинги 13.24 и 13.25).

Листинг 13.24. Отключение и включение индекса в InterBase

```
ALTER INDEX ind_VENDOR INACTIVE;
/* блок операций с данными */
ALTER INDEX ind_VENDOR ACTIVE;
```

Листинг 13.25. Отключение индекса в SQL Server

```
ALTER INDEX ind_VENDOR ON VENDORS DISABLE;
```

Такая операция может пригодиться, когда в короткий срок времени большей части данных, хранящихся в таблице VENDORS, грозит удаление или изменение. Мы временно отключаем индекс, чтобы приостановить процесс индексирования и не перегружать сервер баз данных. Если вы предположили, что для включения индекса в SQL Server потребуется команда «Enable», – вы ошиблись, в Transact-SQL для этой задачи применяется инструкция ALTER INDEX REBUILD. Однако если вы работаете в Oracle, то «Enable» – как раз то, что надо. В InterBase включением/отключением ведают ключевые слова ACTIVE/INACTIVE.

После подключения индекса СУБД самостоятельно реорганизует его и приведет в соответствие с текущими данными проиндексированных столбцов таблицы.

Внимание!

Если вы администрируете большую, активно эксплуатируемую многопользовательскую базу данных, то периодически стоит заставлять СУБД перестроить все индексы. Подобные операции следует проводить в период наименьшего использования БД (в выходные и праздничные дни или в ночное время), предварительно не забывая создать полную копию БД и журналов транзакций.

Удаление индекса

Синтаксис удаления индекса включает в себя ключевые слова DROP INDEX и непосредственно имя таблицы и индекса, разделенные точкой.

```
DROP INDEX имя_индекса;
```

В некоторых диалектах SQL (листинг 13.26), кроме имени индекса, следует указать имя таблицы, которой он принадлежит.

Листинг 13.26. Удаление индекса в MySQL

```
DROP INDEX ind_SUPPLIERS_SUPPLIER ON SUPPLIERS;
```

Представления

Познакомимся с очередным объектом БД – представлением (view), или, как его иногда называют, виртуальной таблицей. Представление показывает пользователю не реально существующие в БД таблицы, а виртуальные отношения, созданные в результате одной или нескольких динамических операций SQL. В обычном состоянии представление «дремлет», но в момент обращения к нему транзакции выполняется заранее подготовленный и хранящийся в БД запрос SQL, основанный на ключевом слове SELECT, и перед пользователем во всей красе возникает виртуальная таблица с данными.

Замечание

Создание представлений с помощью SQL неразрывно связано с применением инструкции выборки данных SELECT. Если эта инструкция пока вам незнакома, то сначала стоит прочитать *главу 12*.

В умелых руках механизм представлений может стать дополнительным элементом защиты БД. Во-первых, благодаря представлениям от отдельных категорий пользователей легко скрыть части таблиц БД, например столбец в таблице. Во-вторых, в подавляющем большинстве случаев любые манипуляции с представлениями не способны навредить реальным данным. Если представление доступно только для чтения, то все попытки несанкционированного редактирования данных обречены на провал.

Вместе с тем основная задача представлений заключается не столько в борьбе за безопасность данных, сколько в наделении разработчика БД дополнительной степенью свободы. Поэтому мы в первую очередь станем рассматривать представление как инструментарий, упрощающий доступ к данным, ускоряющий разработку проекта БД и снижающий трудозатраты при перестроении структуры БД. Допустим, клиентские приложения обращаются к определенному срезу данных, формируя динамические запросы SQL не напрямую к таблицам, а при посредничестве представления. Если вдруг у вас возникнет задача изменить правила выборки данных, то совсем не стоит переписывать клиентские приложения, вместо этого достаточно внести правки только в од-

ном месте – на стороне сервера изменить запрос на выборку SELECT в SQL-коде представления.

Еще одно преимущество представлений – упрощение схемы данных. При желании мы сможем подменить сложный в понимании уровень взаимосвязанных реляционных таблиц (зачастую состоящий из сотен отношений) более наглядным слоем не столь многочисленных представлений.

Никто не запрещает нам создавать новые представления, комбинируя уже существующие представления и базовые таблицы. При необходимости, постепенно укрупняя отношения БД, мы даже сможем создать единое представление всей БД – некую глобальную виртуальную таблицу, отражающую абсолютно все данные.

К сожалению, представления не свободны от недостатков. Основным минусом является невозможность прямого редактирования представления, созданного путем соединения двух и более таблиц. Кроме того, несколько снижается производительность запросов, выполняемых при посредничестве многотабличных представлений.

Для создания представления в большинстве диалектов SQL необходимо обработать следующую синтаксическую конструкцию:

```
CREATE VIEW <имя_представления>  
AS <запрос на основе SELECT> [WITH [CASCADED|LOCAL] CHECK OPTION]
```

Наличие команды RECURSIVE говорит о том, что создается представление, которое будет получать данные из себя самого. Определение представления содержит обычный запрос на базе оператора SELECT, но с некоторыми ограничениями – обычно нельзя использовать операторы ORDER BY и GROUP BY (соответственно упорядочивающие и группирующие набор строк). При объединении двух и более таблиц надо быть внимательным при совпадении имен столбцов таблиц. В таких случаях команда SELECT * ... (выбор всех атрибутов) является недопустимой, и совпадающее имя надо переименовать при помощи оператора AS. Также явный список имен столбцов после оператора SELECT необходимо описать в случае, если атрибуты содержат вычисляемые значения или существуют объединенные атрибуты с разными именами в соответствующих им таблицах.

Собственно создающий представление текст запроса описывается на базе рассмотренной в главе 12 инструкции SELECT. В листинге 13.27 представлен пример создания простейшего представления, объединяющего все данные из таблиц поставщиков и накладных.

Листинг 13.27. Создание представления на основе двух таблиц

```

CREATE VIEW view_suppliers_deliverynotes
AS
SELECT S.SUPPLIER_ID AS SUPPLIER_ID,
       D.DELIVERYNOTE_ID AS DELIVERYNOTE_ID,
       S.SUPPLIER AS SUPPLIER,
       D.DNDATE AS DNDATE,
       D.DNNUM AS DNNUM
FROM SUPPLIERS S
JOIN DELIVERYNOTES D ON D.SUPPLIER_ID = S.SUPPLIER_ID;

```

Внимание!

При описании запроса `SELECT` для определения представления в большинстве СУБД следует соблюдать ряд ограничений, например запрещено применять операторы `UNION`, `EXCEPT`, `INTERSECT`, `ORDER BY`, `GROUP BY`, `HAVING`, `DISTINCT`. Впрочем, из правил всегда есть и исключения, например MySQL допускает группировку данных (`GROUP BY`) в представлении.

В ряде диалектов синтаксис команды создания представления допускает явное присваивание имен столбцам будущей виртуальной таблицы. Новые имена перечисляются в круглых скобках сразу за названием будущего представления (листинг 13.28).

Листинг 13.28. Определение имен столбцов в представлении

```

CREATE VIEW view_vendors_countries (ID, VENDOR_NAME, COUNTRY_NAME)
AS
SELECT
    V.VENDORS_ID, V.VENDOR, C.COUNTRY
FROM VENDORS V
JOIN COUNTRIES C ON C.COUNTRY_ID = V.COUNTRY_ID;

```

Впрочем, приведенный метод назначения имен столбцам опирается на традиционный для стандартного SQL способ переименования столбца с помощью команды `AS`, поэтому при желании читатель может работать «по старинке». Например, в листинге 13.29 мы создаем искусственный столбец `SUMPRICE`, в котором отразится суммарная стоимость товара `GOODS`, которая формируется за счет умножения количества единиц товара на отпускную цену производителя. Имя столбцу назначается при помощи `AS`.

Листинг 13.29. Представление с искусственным столбцом

```
CREATE VIEW view_warehouse_sumprice
AS
SELECT *, (GL.FACTORYPRICE*GL.AMOUNT) AS SUMPRICE FROM GOODSLIST GL
NATURAL JOIN GOODSCCLASS
NATURAL JOIN DELIVERYNOTES;
```

Тем столбцам, имена которым не были присвоены явно, СУБД назначит название самостоятельно. Как правило, новые названия полностью соответствуют именам столбцов в физических таблицах.

К представлению, как и к обычной таблице, можно обратиться, используя инструкцию `SELECT` (листинг 13.30).

Листинг 13.30. Выборка данных из представления

```
SELECT * FROM view_suppliers_deliverynotes;
```

Представления – весьма полезный инструмент. Пользователь воспринимает представление как реальную таблицу и даже не задумывается о том, что это результат выполнения запроса к одной или нескольким таблицам. Благодаря механизму представлений разработчик баз данных получает возможность: во-первых, организовать доступ к информации в наиболее удобном для пользователя виде; во-вторых, скрыть от пользователя ненужные ему данные и, в-третьих, упростить свою работу, создавая представления для базовых отношений между таблицами и обращаясь к ним по мере необходимости вместо построения одних и тех же запросов.

Изменение представления

Модификация представления может быть осуществлена несколькими способами. Во-первых, можно применить традиционный для DDL способ, основанный на синтаксической конструкции `ALTER VIEW`. Во-вторых, если представление в явном виде не вызывается из кода хранимых процедур или триггеров, то его можно просто удалить и создать вновь.

В любом случае, осуществляя модификацию представления, учитывайте, что его услугами могут пользоваться другие представления, запросы, хранимые процедуры, функции и триггеры. Поэтому как минимум разработчику стоит позаботиться об идентичности перечня названий и типов значений, возвращаемых обновленным представлением столбцов.

Удаление представления

Для удаления из БД описания представления предназначена лаконичная инструкция:

DROP VIEW имя_представления;

Перед обращением к **DROP VIEW** следует убедиться в том, что представление не задействовано в других объектах БД – выражениях, триггерах, хранимых процедурах, других представлениях и т. п.

Модифицируемые представления

В проектах БД большинство представлений доступно только для чтения. Такой характеристикой автоматически наделяется любое представление, если оно обладает хотя бы одной из перечисленных ниже черт:

- представление построено на базе двух и более таблиц или представлений;
- при описании представления использовались подзапросы;
- представление содержит выражения;
- используется оператор группировки **GROUP BY**;
- используется любая из агрегирующих функций **SUM()**, **MIN()**, **MAX()**, **COUNT()**, **AVG()**;
- задействуются квантификаторы **DISTINCT**;
- в представлении задействуется спецификатор **UNION** или **UNION ALL**;
- множественные ссылки на любой столбец базовой таблицы.

А каким критериям должно соответствовать представление, безусловно допускающее редактирование данных? Приведем и их:

- представление должно обращаться только к одной таблице и возвращать все столбцы, требующие обязательного ввода значений (допускается не возвращать столбцы, при описании которых разработчик назначил значение по умолчанию, или столбцы, допускающие ввод неопределенного значения **NULL**);
- представление обращается к другому представлению, допускающему модификацию данных;
- в представлении отсутствуют столбцы с одинаковыми именами;
- в представлении нет искусственных столбцов.

Кроме того, в представлении должны отсутствовать все черты немодифицируемого представления.

Некоторые СУБД научились обходить ограничения доступных только для чтения данных в представлениях. Например, в InterBase и FireBird это достигается за счет одной весьма полезной способности – эти СУБД обладают возможностью подключать триггеры не только к таблицам, но и к представлениям. Таким образом, для сотворения «чуда» нам всего навсего достаточно создать три триггера: BEFORE INSERT, BEFORE UPDATE и BEFORE DELETE – и описать логику их работы.

Допустим, в нашей БД имеется представление V_SUPPLIERS_DELIVERYNOTES, построенное на основе объединения таблицы поставщиков и накладных (приложение 1). Разработаем триггер, который хотя и принадлежит к формально не модифицируемому представлению, но способен изменить данные в связанных с ним физических таблицах (листинг 13.31).

Листинг 13.31. Триггер обновления для модифицируемого представления

```
CREATE TRIGGER T_V_SUPPLIERS_DELIVERYNOTES FOR V_SUPPLIERS_DELIVERYNOTES
ACTIVE BEFORE UPDATE AS
begin
    UPDATE SUPPLIERS
    SET SUPPLIER=NEW.SUPPLIER
    WHERE SUPPLIER_ID=OLD.SUPPLIER_ID;

    UPDATE DELIVERYNOTES
    SET DNNUM=NEW.DNNUM, DNDATE=NEW.DNDATE
    WHERE DELIVERYNOTE_ID=OLD.DELIVERYNOTE_ID;
end
```

Так как представление V_SUPPLIERS_DELIVERYNOTES обслуживает две таблицы, в триггере обновления нам приходится вносить правки одновременно в таблицы поставщиков SUPPLIERS и накладных DELIVERYNOTES.

Замечание

В InterBase триггер представления выполняется раньше, чем триггеры таблиц, из которых собрано представление.

Резюме

За физическое создание новой БД и ее объектов отвечает подмножество языка SQL, называемое языком определения данных (Data Definition Language, DDL).

В своей работе DDL опирается всего-навсего на три инструкции: CREATE, ALTER и DROP (создание, модификация и удаление соответственно). С помощью перечисленных инструкций создается большинство объектов БД: домены, таблицы, индексы, представления, а в случае если СУБД поддерживает процедурные расширения SQL, то и хранимые процедуры, функции и триггеры (см. главу 14).

Вопросы для самопроверки

1. Расшифруйте аббревиатуру DDL.
2. Какие задачи решаются с помощью доменов?
3. Какие возможности по модификации таблиц предоставляет DDL, какие, на ваш взгляд, при этом могут возникнуть ограничения?
4. Каким образом при создании таблиц описываются правила обеспечения ссылочной целостности?
5. В каком случае индексы таблиц создаются автоматически?
6. Почему представление называют виртуальным отношением?
7. Каким правилам надо следовать, чтобы представление стало модифицируемым?
8. Разработайте инструкции SQL, создающие (см. приложение 1):
 - а) БД «Склад»;
 - б) домены;
 - в) таблицы БД и связи между ними;
 - г) индексы таблиц;
 - е) представление, объединяющее 2–3 таблицы.

Глава 14

Процедурный SQL

Далеко не секрет то, что на ранних этапах развития структурированного языка запросов ученые и программисты поставили перед собой цель создать непроведурный язык программирования. Дело в том, что новый язык реляционных БД предназначался не только для программистов, но и для обычных пользователей, поэтому в идеале он должен быть декларативным. В соответствии с таким подходом оператору ЭВМ было достаточно лишь поставить СУБД правильно сформулированную задачу, а каким образом СУБД станет решать эту задачу, пользователя не касалось.

В целом задумка удалась. Современный SQL имеет все признаки декларативного языка, для доказательства этого утверждения приведу всего лишь один пример. Допустим, вам следует найти минимальное значение в какой-то произвольной таблице. Какую вы сформируете инструкцию SQL? Примерно такую:

```
SELECT MIN(column1) FROM table1;
```

В переводе на русский это звучит так: «Выбрать минимальное из значений столбца column1 из таблицы table1». Другими словами, вы поставили задачу серверу на естественном для человека языке. Если бы вы решали схожую задачу на классическом процедурном языке программирования, вам бы пришлось объявить ряд переменных и организовать цикл, в котором бы осуществлялся последовательный перебор всех значений столбца и проводились операции сравнения текущего значения с предыдущим.

Однако возможности декларативного языка не безграничны. В тех ситуациях, когда так необходимая вам операция с данными хотя бы на немного отходит от привычной для анализатора SQL канвы, язык перестает справляться. В результате и колесо закрутилось назад – во всех диалектах SQL стали появляться процедурные элементы.

Ярчайшими представителями процедурной идеи выступают хранимые процедуры (stored procedures), функции и триггеры (triggers). Это подпрограммы, заранее подготовленные разработчиком БД. Хранимые процедуры, функции и триггеры компилируются и хранятся как самостоятельный исполняемый код в системном каталоге БД. Ключевое отличие процедур и функций, с одной стороны, и триггеров – с другой, заключается в том, что первые могут вызываться с любого места инструкции SQL с помощью оператора CALL (EXECUTE) или SELECT, а также программы, написанной на высокоуровневом языке, а триггеры запускаются СУБД автоматически по определенному событию без какого-либо вмешательства извне.

При написании кода SQL разработчик БД при прочих равных условиях всегда должен отдавать предпочтение хранимым процедурам и функциям. Тому имеется несколько веских причин:

- 1) повышение производительности. В БД хранимые процедуры хранятся в откомпилированном и оптимизированном виде. Как следствие выполнение хранимой процедуры происходит быстрее, чем запуск аналогичного кода динамического SQL;
- 2) снижение объема передаваемых данных. Для вызова хранимой процедуры достаточно указать ее имя и значения параметров, в любом случае это меньше, чем отправка по сети полного текста SQL-запроса. Как следствие значительно сокращается сетевой трафик, что весьма актуально в обычно перегруженных вычислительных сетях;
- 3) обеспечение безопасности данных. Хранимые процедуры как нельзя лучше вписываются в систему безопасности СУБД, предназначенную для защиты данных от несанкционированного доступа. В отличие от сформированного злоумышленником SQL-запроса, в котором могут содержаться вредоносные команды, разработанные профессиональными программистами и хранящиеся на сервере процедуры не смогут нанести вреда БД. Обладая правами на вызов хранимой процедуры, пользователь может даже и не подозревать, какие таблицы будут задействованы для выполнения поставленной задачи;
- 4) повторное использование кода. За годы разработки БД опытный программист постепенно нарабатывает свою собственную библиотеку отлаженных процедур. Незначительно модифицируя код, программист применяет процедуры в новых БД. Благодаря

такому подходу обеспечивается высокая скорость разработки проекта.

Элементы процедурного SQL

Прежде чем приступить к изучению особенности разработки хранимых процедур и триггеров, познакомимся с опорными программными конструкциями, которые можно применять в коде SQL. Особо подчеркнем, что каждый из диалектов SQL предоставляет разработчику свое собственное видение процедурного программирования, поэтому при общем сходстве концепций (наличие переменных, условных операторов, операторов цикла и т. п.) диалекты разных производителей пусть незначительно, но все-таки отличаются друг от друга. Не имея возможности предложить читателю универсальные синтаксические конструкции, охватывающие хотя бы наиболее популярные версии SQL, в основном мы рассмотрим процедурные возможности диалекта SQL, применяемого в СУБД MySQL.

Переменные

Наиболее распространенный способ хранения данных, используемых в промежуточных вычислениях, реализуется при посредничестве переменных. Переменные позволяют размещать в памяти значения различных типов (целые и вещественные числа, символы, текстовые строки).

Можно выделить три категории переменных:

- пользовательские переменные – предназначены для решения локальных задач с применением интерактивного SQL, они доступны исключительно в рамках отдельной сессии;
- системные переменные – играют глобальную роль и в своей основной массе связаны с работой сервера;
- локальные переменные – используемые в коде хранимых процедур (функций, триггеров).

Переменные, определяемые пользователем

Жизненный путь пользовательской переменной начинается с ее объявления, для этого следует определиться с идентификатором (именем) переменной. Имена переменных, определяемых пользователем, должны начинаться с символа «@».

Несколько примеров объявления переменных вы найдете в листинге 14.1.

Листинг 14.1. Примеры объявления переменных в MySQL

```
SET @A=1;
SET @B:='Hello, SQL!';
SET @C=true, @D=false;
SELECT @E:=@A+2;
```

Как видите, для определения переменной и присваивания значения потребуются ключевые слова SET или SELECT и операторы присваивания «:=» или «: =».

Системные переменные

Основное назначение системных переменных заключается в информировании оператора об особенностях конфигурации и работы БД и для изменения параметров сервера. При этом различают глобальные и сеансовые системные переменные. Первая категория переменных оказывает влияние на все операции сервера, а сеансовые – лишь на операции текущего клиента.

Глобальные переменные в большей степени нужны администратору сервера, чем разработчику БД, вместе с тем ряд системных переменных может оказать весьма полезные услуги, например для организации информирования пользователя о ключевых параметрах сервера.

Один из примеров применения системных переменных с целью информирования пользователя о сервере MySQL предложен в листинге 14.2. Здесь хранимая процедура pr_DBMSINFO() собирает наиболее востребованную информацию, помещает ее во временную таблицу и затем, после заполнения таблицы, возвращает пользователю.

Листинг 14.2. Фрагмент процедуры, использующей системные переменные

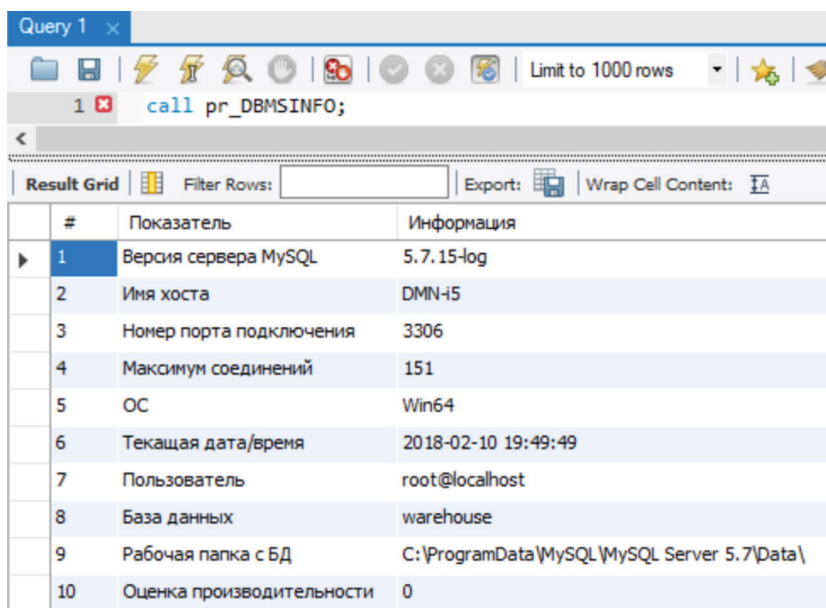
```
CREATE PROCEDURE pr_DBMSINFO()
BEGIN
  DROP TEMPORARY TABLE IF EXISTS dbmsinfo_table;
  CREATE TEMPORARY TABLE dbmsinfo_table
    (num int, caption varchar(25), info varchar(100));
  -----
  INSERT INTO dbmsinfo_table VALUES(1, 'Версия сервера MySQL', @@version);
  -----
  INSERT INTO dbmsinfo_table VALUES(2, 'Имя хоста', @@hostname);
  -----
```

```

INSERT INTO dbmsinfo_table VALUES(3, 'Номер порта подключения', @@port);
-----
/* и т. д. */
SELECT num AS '№', caption AS 'Параметр', info AS 'Описание'
FROM dbmsinfo_table;
END

```

В MySQL переменная @@version содержит версию сервера, @@version_compile_os – версия операционной системы сервера, @@hostname – названия хоста и т. п. Результаты выполнения процедуры, использующей системные переменные, предложены читателю на рис. 14.1.



#	Показатель	Информация
1	Версия сервера MySQL	5.7.15-log
2	Имя хоста	DMN-I5
3	Номер порта подключения	3306
4	Максимум соединений	151
5	ОС	Win64
6	Текащая дата/время	2018-02-10 19:49:49
7	Пользователь	root@localhost
8	База данных	warehouse
9	Рабочая папка с БД	C:\ProgramData\MySQL\MySQL Server 5.7\Data\
10	Оценка производительности	0

Рис. 14.1. Вызов процедуры информирования о состоянии сервера

Внимание!

При проектировании корпоративных программных продуктов разработчик БД не должен менять ключевые параметры сервера без согласования с системным администратором и другими разработчиками.

Переменные в хранимой процедуре

Для объявления локальной переменной в теле хранимой процедуры потребуется помощь ключевого слова DECLARE, затем указывается имя и тип переменной.

```
DECLARE имя_переменной [, имя_переменной, ...] тип_данных  
    [DEFAULT значение_по_умолчанию];
```

Переменные объявляются в первых строках кода хранимой процедуры, сразу за ключевым словом **BEGIN** и будут доступными до **END**, закрывающего тело процедуры.

Составной оператор **BEGIN..END**

Составной оператор **BEGIN..END** предназначен для логического объединения двух и более операторов, которые должны выполняться как единое целое. Задача оператора – указание начала и конца программного блока. Характерным примером подобного блока является создаваемый по умолчанию каркас кода хранимой процедуры, здесь весь исполняемый код заносится вовнутрь **BEGIN..END**. Кроме того, составные операторы активно используются в коде функций и триггеров в условных операторах и при построении циклов.

Внимание!

Выражения, заключенные в рамки составного оператора **BEGIN..END**, рассматриваются компилятором как составной оператор, подлежащий выполнению как единое целое.

Условные операторы

Организуя вычислительный процесс, программист должен обладать возможностью направлять ход вычислений в то или иное русло в зависимости от полученных в ходе выполнения программы промежуточных результатов или введенных пользователем исходных данных. Для этих целей в процедурном SQL предусмотрен набор привычных нам по высокоуровневым языкам программирования операторов.

Оператор **IF..THEN..ELSE**

Для построения едва ли не самой важной для всех процедурных языков программирования конструкции ветвления (если...то...иначе) в MySQL предусмотрен классический оператор условия **IF**. В простейшем случае, когда выполнение выражения осуществляется, если проверяемое условие возвращает истину (**true**), синтаксис оператора таков:

```
IF (проверяемое_условие) THEN  
    выполняемое_выражение;  
END IF;
```

Если же нас интересует и действие, осуществляемое при невыполнении условия, то конструкция дополняется ELSE:

```
IF (проверяемое_условие) THEN
    выполняемое_выражение_1;
ELSE
    выполняемое_выражение_2;
END IF;
```

Замечание

Если при выполнении того или иного условия нам требуется выполнить несколько выражений, то их следует заключать в составной оператор BEGIN . . END.

Оператор-селектор CASE

Практически во многих высокоуровневых языках программирования оператор-селектор CASE введен для повышения удобства работы программиста и предоставления ему возможности создавать читабельные листинги программ. В процедурном SQL решаемая селектором задача схожая – осуществить множественный выбор, но особенностью CASE в БД является то, что здесь оператор обладает двумя синтаксическими формами.

Замечание

При желании селектор CASE может без каких-либо затруднений быть замещен конструкцией IF...THEN...ELSE – все зависит от предпочтений разработчика

Первая форма CASE практически ничем не отличается от аналогичных конструкций в обычных высокоуровневых языках программирования:

```
CASE контрольное_значение
    WHEN значение_1 THEN выражение_1
    [WHEN значение_2 THEN выражение_2] ...
    [ELSE выражение_3]
END CASE
```

Селектор последовательно сравнивает контрольное_значение со значениями значение_1, значение_2 и т. д. и в случае совпадения выполняет соответствующее выражение. Если совпадений не было обнаружено, выполнению подлежит код секции ELSE. Пример использования подобного селектора представлен в листинге 14.3.

Листинг 14.3. Функция преобразования номера дня недели в текст

```

CREATE FUNCTION fn_weekday(weekdaynum tinyint) RETURNS varchar(12)
BEGIN
CASE weekdaynum
WHEN 1 THEN RETURN 'Понедельник';
WHEN 2 THEN RETURN 'Вторник';
...
WHEN 7 THEN RETURN 'Воскресенье';
ELSE RETURN NULL;
END CASE;
END

```

Представленная функция преобразует поступающие на вход номера дней недели в их текстовое название, а если число оказывается в недопустимом диапазоне, функция возвратит неопределенность NULL.

Вторая форма CASE обладает большей гибкостью

```

CASE
WHEN выражение_сравнения THEN выражение_1
[WHEN выражение_сравнения THEN выражение_2] ...
[ELSE выражение_3]
END CASE

```

Здесь вместо простого указания единственного контрольного значения сравнение описывается непосредственно для каждой из конструкции WHEN. В качестве выражения сравнения может приниматься любой набор операторов, при условии что он возвратит истину или ложь. Как только выражение сравнения возвращает истину, выполняется соответствующее выражение.

Допустим, у нас есть некоторые нормативные требования к количеству продукции, хранимой на складе, разработаем функцию, конвертирующую числовое значение в его текстовое описание (листинг 14.4).

Листинг 14.4. Функция преобразования номера дня недели в текст

```

CREATE FUNCTION fn_amount_to_text(Amount decimal(8,4))
RETURNS varchar(25)
BEGIN
CASE
WHEN Amount=0 THEN RETURN 'Продукция закончилась';
WHEN (Amount>0) AND (Amount<10) THEN RETURN 'Низкий запас';

```



```

WHEN (Amount>=10) AND (Amount<20) THEN RETURN 'Нормальный запас';
WHEN (Amount>=20) THEN RETURN 'Избыток продукции';
ELSE RETURN 'Неопределенность';
END CASE;
END

```

А теперь рассмотрим иллюстрацию примера использования только что разработанной функции в реальном запросе к таблице остатков товаров (напомню, что остаток продукции на складе хранится в столбце AMOUNT таблицы TRANSFERS). Представленный на рис. 14.4 экранный снимок предлагает вниманию читателя как текст запроса SELECT, так и результаты его выполнения.

Query 1 x fn_amount_to_text - Routine

```

1 select goodslist.GOODS, goodslist.FACTORYPRICE, transfers.AMOUNT, fn_amount_to_text(transfers.AMOUNT)
2 FROM goodslist
3 join transfers ON transfers.GOODSLIST_ID=goodslist.GOODSLIST_ID

```

GOODS	FACTORYPRICE	AMOUNT	fn_amount_to_text(transfers.AMOUNT)
SM-900	55000.0000	3.00	Низкий запас
SN-901	56000.0000	2.00	Низкий запас
PH-101	15000.0000	15.00	Нормальный запас
GF-0012c	14500.0000	15.00	Нормальный запас

Рис. 14.2. Обращение к функции в тексте запроса SELECT

Циклы

В большинстве диалектов SQL допускается применение двух-трех типов циклов. Наиболее популярны циклы WHILE (цикл с предусловием) и REPEAT (цикл с постусловием). Несколько реже встречается цикл LOOP – петлевой цикл для выхода, из которого обязательно задействуется оператор LEAVE.

Цикл с предусловием WHILE

Цикл с предусловием выглядит следующим образом:

```

[метка:] WHILE условие_цикла DO
операторы_цикла
END WHILE [метка]

```

Операторы цикла выполняются только в том случае, если условие цикла возвращает истину.

Замечание

Все типы циклов могут расширять свои возможности за счет включения необязательных операторов, которые позволяют осуществлять досрочный выход из цикла (оператор `LEAVE` или `BREAK`) и пропуск итерации (`ITERATE` или `CONTINUE`).

Вариант работы цикла с предусловием представлен в листинге 14.5. В предложенной вниманию читателя процедуре рассчитываются координаты гиперболы. Может быть, это и не вполне характерная задача для БД, однако особенности работы с циклами как нельзя лучше раскрывает математика. Итак, гипербола описывается хорошо знакомой нам по учебнику формуле $y = k/x$. Для того чтобы получить значение координаты y на установленном отрезке, в качестве параметров на вход процедуры поступает коэффициент k и границы исследуемого отрезка – аргументы `minX` и `maxX`.

Листинг 14.5. Хранимая процедура для расчета гиперболы $y = k/x$ и цикл `WHILE`

```
CREATE PROCEDURE pr_hyperbola(IN k dec(4,1),
                             IN minX dec(4,1), IN maxX dec(4,1))
BEGIN
  declare x float;
  set x=minX;
  while x<=maxX do -- цикл выполняется, пока x меньше и равно maxX
    select x,k/x;
    set x=x+0.1;
  end while;
END;
```

На рис. 14.3 представлен экранный снимок работы процедуры. Обратите внимание на особенность реакции сервера MySQL на попытку деления k на 0. В данном случае СУБД «хитрит», и, вместо того чтобы сгенерировать исключительную ситуацию как реакцию на недопустимую математическую операцию, сервер возвращает неопределенность `NULL`.

Цикл с постусловием REPEAT

Цикл с постусловием основан на операторе `REPEAT`, например в MySQL синтаксис выглядит следующим образом:

```
[метка:] REPEAT
операторы_цикла
UNTIL условие_цикла
END REPEAT [метка]
```

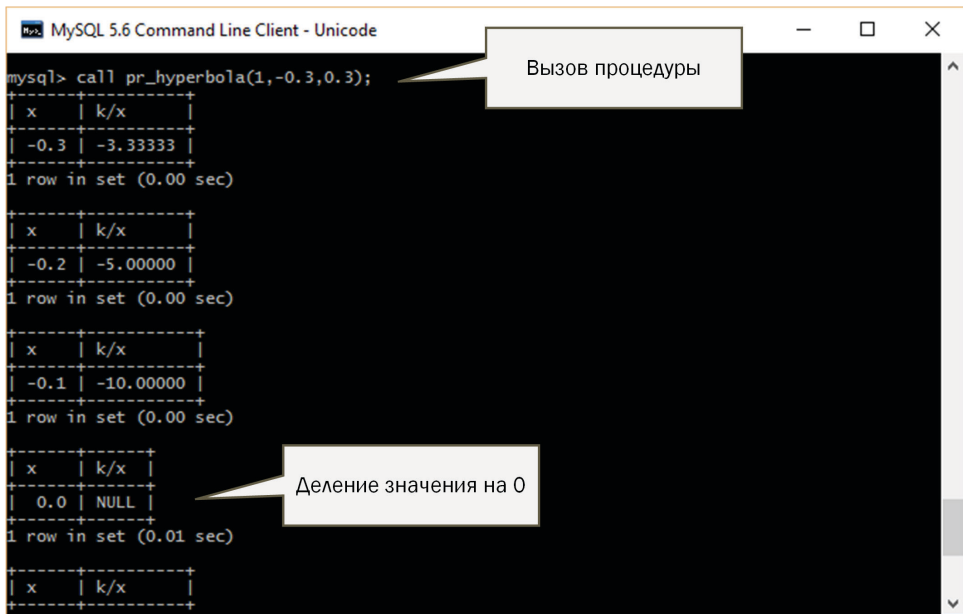


Рис. 14.3. Отклик MySQL на попытку деления на 0

Цикл REPEAT имеет две особенности:

- 1) цикл выполняется, пока его условие ложно;
- 2) вне зависимости от состояния условия цикл выполнит, по крайней мере, одну итерацию, ведь проверяемое условие находится в последней строке его кода.

Для демонстрации работы цикла REPEAT вновь решим математическую, точнее тригонометрическую задачу построения таблицы синусов. Но на этот раз объединим тригонометрию с задачами, характерными для БД, – поместим результаты расчетов во временную таблицу (листинг 14.6).

Листинг 14.6. Построение таблицы синусов в цикле REPEAT

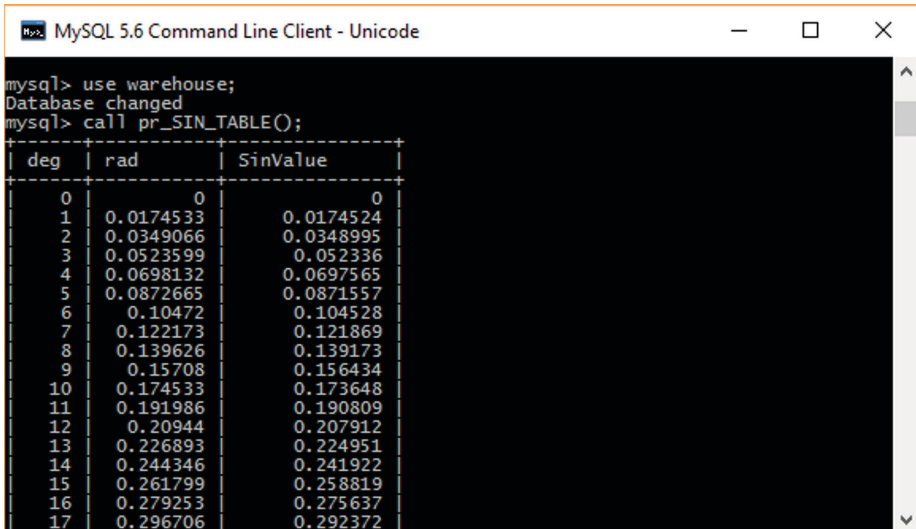
```

CREATE PROCEDURE pr_sin_table()
BEGIN
    DECLARE d int DEFAULT 0; # переменная для хранения градусов
    DECLARE r float DEFAULT 0; # переменная для хранения радиан
    # ---- создаем временную таблицу с 3 столбцами -----
    DROP TEMPORARY TABLE IF EXISTS sin_table;
    CREATE TEMPORARY TABLE sin_table (deg int, rad float, SinValue float);

```

```
# ---- цикл заполнения таблицы данными -----
REPEAT
  INSERT INTO sin_table VALUES (d, r, sin(r));
  SET r=r+pi()/180;
  SET d=d+1;
UNTIL r>=pi()
END REPEAT;
# -----
SELECT * FROM sin_table; # результат - выборка из таблицы синусов
END;
```

В нашем примере цикл продолжает работу до тех пор, пока не будет просмотрен диапазон значений от 0 до π . В теле цикла осуществляется расчет значения синуса (для этого задействуется библиотечная функция `sin()`), и результат заносится в соответствующий столбец временной таблицы. Затем, применив запрос `SELECT`, мы делаем полную выборку из таблицы и удаляем ее. Результат выполнения процедуры отражен на рис. 14.4.



The screenshot shows a MySQL 5.6 Command Line Client window. The user has executed the command `mysql> use warehouse;` and the response is `Database changed`. Then, the user executed `mysql> call pr_SIN_TABLE();`. The result is a table with three columns: `deg`, `rad`, and `SinValue`. The table contains 18 rows of data, starting from 0 degrees and ending at 17 degrees.

deg	rad	SinValue
0	0	0
1	0.0174533	0.0174524
2	0.0349066	0.0348995
3	0.0523599	0.052336
4	0.0698132	0.0697565
5	0.0872665	0.0871557
6	0.10472	0.104528
7	0.122173	0.121869
8	0.139626	0.139173
9	0.15708	0.156434
10	0.174533	0.173648
11	0.191986	0.190809
12	0.20944	0.207912
13	0.226893	0.224951
14	0.244346	0.241922
15	0.261799	0.258819
16	0.279253	0.275637
17	0.296706	0.292372

Рис. 14.4. Вызов хранимой процедуры построения таблицы синусов угла

Петлевой цикл LOOP

Основная особенность петлевого цикла заключается в том, что в нем (в отличие от циклов `WHILE` и `REPEAT`) отсутствует условие выхода. Поэтому программист обязан предусмотреть способ прерывания петли,

иначе мы рискуем получить бесконечный процесс. Синтаксис цикла выглядит следующим образом:

```
[метка:] LOOP
операторы_цикла
END LOOP [метка]
```

Обратите внимание, что на этот раз при работе с циклом целесообразно использовать метки, они помогут организовать выход из цикла. Кроме меток, для выхода из цикла еще понадобятся услуги выражения `LEAVE`.

Рассмотрим практический пример использования в хранимой процедуре петлевого цикла. В листинге 14.7 мы научим MySQL рассчитывать показательную функцию вида $y = a^x$. На вход хранимой процедуры поступают три аргумента: значение a , начало и конец диапазона значений x .

Листинг 14.7. Хранимая процедура расчета показательной функции и цикл `LOOP`

```
CREATE PROCEDURE pr_exponential(IN a float, IN minX dec(4,1), IN maxX dec(4,1))
BEGIN
  DECLARE x dec(4,1);
  DECLARE y float;
  SET x=minX;
  # ----- начало цикла -----
label1: LOOP
  SET y=pow(a,x); -- расчет
  SELECT x, y;    -- вывод результатов расчета
  SET x=x+0.1;    -- приращение x
  IF x>maxX THEN LEAVE label1; -- условие выхода из цикла
  END IF;
END LOOP label1;
# ----- конец цикла -----
END;
```

Хранимые процедуры и функции

На предыдущих страницах главы нам уже несколько раз встречались листинги хранимых процедур. Так что обобщенная синтаксическая конструкция, предназначенная для создания хранимой процедуры, вряд ли удивит читателя:

```

CREATE PROCEDURE <имя_процедуры> ([параметр1[,...]])
BEGIN
<тело_процедуры>
END

```

В минимальной нотации хранимая процедура должна начинаться с команды `CREATE PROCEDURE`, за которой следует уникальное имя, перечень параметров и, как правило, в рамках составного оператора `BEGIN...END`, исходный код процедуры.

В листинге 14.8 предложен пример простейшей хранимой процедуры для MySQL, суммирующей два числа (входные параметры `X` и `Y`) и возвращающей результат через выходной параметр `SUM`.

Листинг 14.8. Хранимая процедура MySQL, суммирующая 2 целых числа

```

CREATE PROCEDURE pr_add (IN X INT, IN Y INT, OUT SUM INT)
BEGIN
  SET SUM=X+Y;
END;

```

Чаще всего хранимые процедуры задействуются для осуществления операций с данными. Например, листинг 14.9 предлагает вариант процедуры, вставляющей новую запись в таблицу производителей товаров.

Листинг 14.9. Вставка новой строки в таблицу в MySQL

```

CREATE PROCEDURE pr_vendors_insert (IN AVENDOR VARCHAR(100),
                                     IN APHONENUM VARCHAR(20),
                                     IN AEMAIL VARCHAR(100),
                                     IN ACOUNTRY_ID INT,
                                     OUT AVENDORS_ID INT)
BEGIN
  INSERT INTO VENDORS
  (VENDOR, PHONENUM, EMAIL, COUNTRY_ID)
  VALUES
  ( TRIM(AVENDOR), TRIM(APHONENUM), TRIM(AEMAIL), ACOUNTRY_ID);

  SET AVENDORS_ID=LAST_INSERT_ID(); --возвратим значение PK
END;

```

К особенностям представленной хранимой процедуры можно отнести наличие выходного параметра, через который возвращается значение первичного ключа только что добавленной к таблице записи.

А как обстоят дела с определением хранимых процедур в других типах СУБД? Сейчас вы убедитесь, что, несмотря на различия в диалектах SQL, схожих моментов гораздо больше. Судите сами. Начнем с примера хранимой процедуры, редактирующей название поставщика в таблице SUPPLERS для СУБД Oracle (листинг 14.10).

Листинг 14.10. Вставка новой строки в таблицу Oracle

```
CREATE PROCEDURE pr_SUPPLIERS_UPDATE
(ID INT, SUPPLIER_NAME CHAR(40))
BEGIN
    UPDATE SUPPLIERS SET SUPPLIER=SUPPLIER_NAME
    WHERE SUPPLIER_ID=ID;
END;
```

Для сравнения синтаксиса рассмотрим процедуру на диалекте Transact-SQL (листинг 14.11), на этот раз мы добавляем новую запись в таблицу и возвращаем ее первичный ключ.

Листинг 14.11. Вставка новой строки в таблицу SQL Server

```
CREATE PROCEDURE pr_SUPPLIERS_INSERT
@SUPPLIER_NAME VARCHAR(40), @ID INT OUT
AS
INSERT INTO SUPPLIERS
    (SUPPLIER)
    VALUES
    (SUPPLIER_NAME)
SET @ID=@@IDENTITY; --возвратим значение PK
GO;
```

Еще одна хранимая процедура, но на этот раз предназначенная для работы в базах данных, InterBase и Firebird (листинг 14.12), возвратит название класса товара по значению первичного ключа.

Листинг 14.12. Получение названия класса товара по его первичному ключу

```
CREATE PROCEDURE PR_GOODSCLASS_SELECTNODE
(AGOODSCLASS_ID INTEGER)
```

RETURNS`(AGOODSCLASS VARCHAR(255))`**AS****BEGIN**

```

SELECT GOODSCLASS FROM GOODSCLASS
WHERE GOODSCLASS_ID=:AGOODSCLASS_ID
INTO :AGOODSCLASS;

```

END;

Как видите, синтаксис хранимых процедур в различных СУБД похож друг на друга, так что, освоив любой из диалектов SQL, читатель при необходимости быстро адаптируется и к особенностям альтернативных СУБД.

Вызов хранимой процедуры

Вызов хранимой процедуры из инструкции SQL осуществляется при посредничестве выражения `CALL`. В MySQL и Oracle синтаксис таков:

```
CALL <имя_процедуры>([параметр1[,...]])
```

Сразу после выражения `CALL` следует имя процедуры и перечень разделенных запятыми аргументов.

Рисунок 14.5 демонстрирует результат вызова хранимой процедуры, суммирующей два числа (листинг 14.8), разработанной нами для MySQL.

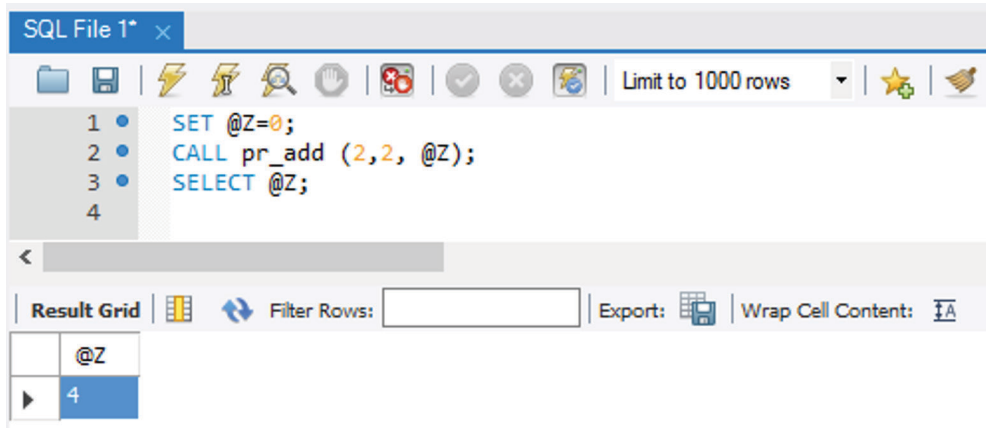


Рис. 14.5. Обращение к хранимой процедуре сложения двух чисел в MySQL

В свою очередь, для вызова хранимой процедуры Oracle (листинг 14.10) потребуется набрать такие строки кода:


```
CALL PROC_SUPPLIERS_UPDATE(112, 'ОАО Аметист')
```

Вызов процедуры из InterBase и Firebird немного отличается. На этот раз вместо команды CALL нам понадобятся услуги EXECUTE PROCEDURE. В минимальной нотации для обращения к хранимой процедуре достаточно указать ее имя.

```
EXECUTE PROCEDURE имя_процедуры
```

Однако в подавляющем большинстве случаев на вход процедуры должны подаваться какие-то аргументы, тогда разделенные запятыми входные значения перечисляют сразу после имени процедуры именно в том порядке, в котором они были объявлены при описании хранимой процедуры. В качестве входных параметров могут применяться как переменные, так и константы.

Сервер SQL Server для обращения к хранимой процедуре работает схожим образом, например для работы с процедурой из листинга 14.11 нам придется воспользоваться следующей строками кода:

```
DECLARE @X INT;
EXEC PR_SUPPLIERS_INSERT 'ОАО Север',@X;
```

Особенности работы с функциями

Кроме хранимых процедур, многие СУБД способны работать с функциями. Отличие между хранимыми процедурами и функциями в большинстве диалектов SQL чисто косметическое – считается, что функция призвана возвращать единственное (скалярное) значение. Хотя есть и исключения, например в SQL Server компании Microsoft выделяют три вида пользовательских функций: скалярные (возвращающие одно значение); внедренные табличные (соответствующие представлениям) и сложные табличные (создающие в программном коде результирующий набор данных).

Как и в ситуации с хранимыми процедурами, синтаксис определения функций разнится от сервера к серверу, так что предложим читателю наиболее обобщенную синтаксическую конструкцию:

```
CREATE FUNCTION <имя_функции> ([параметр1[,...]])
RETURNS <тип_данных>
BEGIN
<тело функции>
END
```

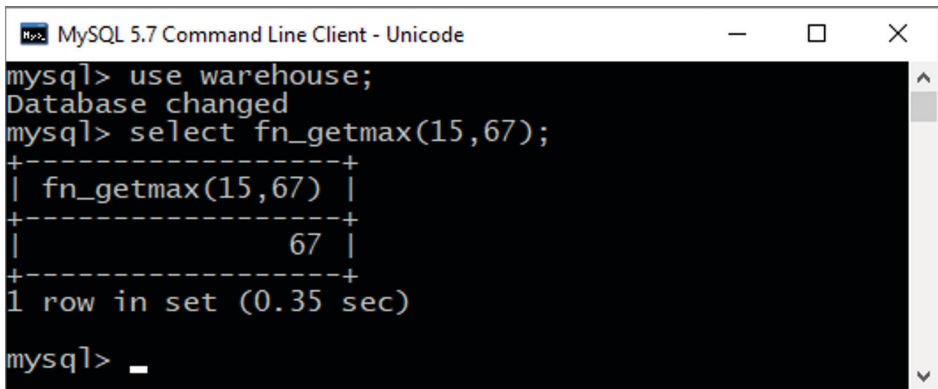
В качестве примера функции рассмотрим листинг 14.13.

Листинг 14.13. Функция MySQL, возвращающая максимальное из 2 целых чисел

```
CREATE FUNCTION fn_getmax(x INT,y INT) RETURNS INT
BEGIN
  IF x>y THEN RETURN x;
  ELSE RETURN y;
  END IF;
END;
```

Все описываемые в круглых скобках параметры являются входными, а тип возвращаемого функцией значения указывается после ключевого слова RETURNS. Для возврата значения в теле функции задействуется хотя бы один оператор RETURN. Названный оператор (или операторы) может находиться в любой строке функции, но надо помнить, что после выполнения любого из RETURN работа функции завершается.

В коде SQL функция может вызываться просто по ее имени, для проверки работоспособности функции можно воспользоваться ключевым словом SELECT (рис. 14.6).



```
MySQL 5.7 Command Line Client - Unicode
mysql> use warehouse;
Database changed
mysql> select fn_getmax(15,67);
+-----+
| fn_getmax(15,67) |
+-----+
|                67 |
+-----+
1 row in set (0.35 sec)

mysql> _
```

Рис. 14.6. Обращение к функции fn_getmax()

Создавая функцию, программист, как правило, предполагает, что, во-первых, его творение решает такую задачу, в которой заинтересованы и другие процедуры, и функции текущей БД, и, во-вторых, код функции с высокой вероятностью окажется востребованным в других проектах БД. Выводом из всего сказанного должно стать то, что программисту следует создавать максимально универсальные конструкции, легко переносимые из одной БД в другую.

Изменение процедур и функций

Для модификации процедур и функций применяют инструкции ALTER:

ALTER PROCEDURE имя_процедуры ...

ALTER FUNCTION имя_функции ...

В остальном синтаксис точно такой же, как и у инструкций CREATE PROCEDURE и CREATE FUNCTION. Это объясняется тем, что на самом деле инструкции ALTER сначала удаляют подлежащую редактированию процедуру (функцию), а затем вновь создают ее.

Удаление процедур и функций

Для удаления ставших ненужными процедур и функций достаточно всего одной строки кода:

DROP {PROCEDURE | FUNCTION} [IF EXISTS] имя_процедуры(функции);

Перед удалением процедуры следует убедиться, что она нигде больше не используется, иначе вы в лучшем случае столкнетесь с отказом БД избавиться от процедуры или в худшем случае нарушите работоспособность всей БД.

Триггеры

Под триггером понимается особая разновидность хранимой процедуры, вызов которой осуществляется автоматически, вне зависимости от пожеланий пользователя БД. Целевое назначение триггеров в первую очередь связано с обеспечением бизнес-правил на уровне таблицы, так как код триггера выступает последней линией обороны по поддержке целостности и непротиворечивости данных. В умелых руках защита таблицы, реализуемая с помощью триггеров, в состоянии стать непреодолимым препятствием для некорректных данных. Эта гарантия обеспечивается правилами построения триггера, которые часто представляют в виде формулы «событие – условие – действие» (event – condition – action rules, ECA rules) [15]:

- триггер гарантированно срабатывает только при наступлении определенного события, обычно связанного с модификацией значений в строке таблицы;

- триггер проверяет условия выполнения операции, вызвавшей его срабатывание;
- если условия верны, то триггер выполняет определенные действия (например, разрешает добавить в таблицу новую строку), а если условия ложны – триггер отвергает операцию.

Кроме того, механизм триггеров позволяет поддерживать корпоративную логику БД в целом, так как на основе их можно строить целую последовательность вызовов (когда срабатывание одного триггера инициирует триггер у другой таблицы), реализуя каскады обновления (удаления) данных в цепочках таблиц.

Замечание

Существенное отличие триггера от хранимой процедуры в том, что за вызов триггера отвечает только сервер SQL, а клиентское приложение никогда не сможет управлять работой триггеров. Благодаря этой особенности триггер попросту незаменим в вопросах поддержания внутренней логики базы данных.

Стандарт SQL рекомендует разработчикам СУБД обеспечить возможность не только описывать триггер, реагирующий на факт свершившегося события (AFTER), но и предоставить возможность вызова триггера перед событием (BEFORE). Таким образом, в идеальной реализации языка SQL должна иметься возможность создания трех пар триггеров:

- события BEFORE INSERT и AFTER INSERT возникают соответственно перед тем, как новая строка попадет в таблицу, и после того, как эта строка будет сохранена;
- события BEFORE UPDATE и AFTER UPDATE генерируются перед началом и после завершения редактирования данных;
- событие BEFORE DELETE предшествует, а AFTER DELETE завершает операцию удаления данных из таблицы.

Многие СУБД позволяют к одной и той же таблице подключать несколько триггеров, явно указывая порядок их срабатывания. Если условие срабатывания триггера BEFORE верно, то действие выполняется и управление передается триггеру AFTER, который, в свою очередь, проверяет свое условие и выполняет свое действие. При невыполнении условия в любом триггере BEFORE осуществляется отмена действия, и данные обрабатываемой строки возвращаются в исходное состояние. Если за вызвавшим исключение триггером BEFORE следует триггер AFTER, то управление ему даже и не передается.

Очень важно запомнить еще одну деталь. Если срабатывание триггера произошло в рамках транзакции, до этого выполнившей какой-то перечень операций с данными, и в результате срабатывания триггер пришел к выводу, что условие ошибочно, то генерация исключительной ситуации приведет к откату целой транзакции.

Вполне естественно, что для создания триггера на различных платформах SQL применяются разные синтаксические конструкции, но вне зависимости от диалекта все они стартуют с команды `CREATE TRIGGER`. В качестве обобщенной синтаксической конструкции предложим следующую:

```
CREATE TRIGGER имя_триггера
{BEFORE | AFTER} (DELETE | UPDATE | INSERT [OF имя_столбца]])
ON имя_таблицы
[FOR EACH {ROW|STATEMENT}]
[WHEN дополнительные условия выполнения триггера]
BEGIN ATOMIC
  <код триггера>
END;
```

Таким образом, создавая триггер, программист как минимум должен определиться с его именем, перечнем событий, на которые обязан реагировать триггер, и именем обслуживаемой таблицы.

Предложение `FOR EACH ROW` укажет, что триггер срабатывает отдельно для каждой из строк таблицы, попавших под воздействие команды SQL. Такие триггеры называют триггерами уровня кортежа (row-level trigger). Например, инструкция `DELETE` удаляет некоторое количество строк из таблицы. Если под воздействие этой команды SQL попадет всего 1 строка – триггер выполнится 1 раз для этой строки, если 2 – то триггер выполнится дважды индивидуально для каждой из строк и т. д. В ситуации, когда вы воспользуетесь услугами `FOR EACH STATEMENT`, будет создан триггер уровня команды (statement level triggers). Этот триггер, вне зависимости от количества строк, попавших под воздействие инструкции SQL, сработает только один раз.

Замечание

Помимо триггеров уровня таблиц, СУБД Oracle позволяет создавать триггеры уровня схемы (срабатывает всегда, когда владелец схемы выполняет операцию, на которую должен отреагировать триггер) и триггеры уровня базы данных (срабатывает, когда любой пользователь БД выполняет команду, на которую обучен отзываться триггер).

Контекстные переменные

Для того чтобы разработчик триггера мог получать доступ как к новым, поступающим в таблицу с очередной строкой значениям, так и к старым (подлежащим модификации) значениям столбцов, в теле триггера в ряде диалектов SQL (Oracle, MySQL, InterBase, Firebird и т. п.) мы имеем право обращаться к двум разновидностям контекстных переменных: NEW и OLD. Для доступа к ним потребуется небольшое уточнение – указание имени столбца таблицы, с которым ассоциируется контекстная переменная, например NEW.SUPPLIER или OLD.VENDOR.

Контекстная переменная NEW применяется в триггерах, вызываемых в момент вставки новой строки или модификации данных в таблице (в случае удаления строки в триггере DELETE такой переменной просто не существует). В этой переменной окажется новое значение, претендующее на попадание в таблицу, но пока не сохраненное в ней.

Обращение к переменной OLD имеет смысл в триггерах модификации и удаления данных – в ней хранится старое значение. Сразу заметим, что все значения с префиксом OLD доступны исключительно только для чтения.

Внимание!

В триггерах BEFORE все контекстные переменные NEW, принадлежащие к столбцам автоинкрементного типа, всегда равны 0, т. к. значение им будет присвоено только после физической записи в таблицу. Не забывайте об этом замечании, особенно в тех случаях, когда ваши первичные ключи построены на счетчиках, – в переменной NEW их не идентифицировать.

Замечание

В SQL Server дела обстоят несколько иначе. Здесь перед удалением строки создается логическая таблица DELETED, в которую автоматически передаются удаляемые строки. Если же мы добавляем новые или редактируем существующие строки, то новые (модифицированные) строки сначала направляются в логическую таблицу INSERTED. В коде триггера мы имеем возможность проанализировать содержимое логической таблицы и, если оно нас по какой-то причине не устраивает, отказаться от изменений (осуществить откат транзакции).

Примеры триггеров

Контролируя передаваемые в таблицу значения, программист сможет построить хорошо эшелонированную линию обороны принятой в БД бизнес-логики. Например, в столбцах WEIGHT и GROSSWEIGHT таблицы

GOODSLIST должны соответственно содержаться величины веса нетто и брутто хранящегося на складе товара. Здравая логика подсказывает, что величина чистого веса нетто не может превышать значение веса товара в упаковке. Исходя из этого утверждения, разработаем триггер для СУБД InterBase (листинг 14.14), стоящий на страже только что изложенного бизнес-правила.

Листинг 14.14. Контроль правильности значений нетто и брутто в InterBase

```
CREATE TRIGGER T_GOODSLIST_BI FOR GOODSLIST
ACTIVE BEFORE INSERT POSITION 1
AS
BEGIN
    if ((NEW.WEIGHT IS NOT NULL)
        AND (NEW.GROSSWEIGHT IS NOT NULL)
        AND (NEW.WEIGHT>NEW.GROSSWEIGHT)) THEN
        /* Вес нетто не может превышать вес брутто! */
        EXCEPTION E_WEIGHT_EXCEPTION;
END
```

Столкнувшись с нарушением условия брутто > нетто, триггер вызывает подготовленное заранее исключение E_WEIGHT_EXCEPTION, которое, в свою очередь, прекратит операцию вставки данных и известит об этом пользователя.

При желании триггер контроля значений нетто и брутто можно переделать и вместо вызова исключения просто поправить содержание одного из столбцов, например приравняв нетто и брутто NEW.WEIGHT=NEW.GROSSWEIGHT. Другими словами, в триггерах BEFORE значение контекстной переменной NEW может быть изменено (в триггерах AFTER это занятие не имеет смысла).

В SQL Server компании Microsoft контекстные переменные NEW и OLD не поддерживаются, вместо этого перед операцией удаления создается логическая таблица DELETED, в которую автоматически передаются удаляемые строки. Если же мы добавляем новые или редактируем существующие строки, то новые (модифицированные) строки сначала направляются в логическую таблицу INSERTED. В коде триггера мы имеем возможность проанализировать содержимое логической таблицы и, если оно нас по какой-то причине не устраивает, отказаться от изменений (осуществить откат транзакции).

Допустим, что нам необходимо гарантировать, чтобы отпускная цена поступающего на склад товара, по крайней мере, превышала нулевую

отметку (столбец FACTORYPRICE таблицы GOODSLIST), в противном случае запись не должна попасть в таблицу GOODSLIST.

Листинг 14.15. Триггер таблицы товаров SQL Server

```
CREATE TRIGGER TR_GOODSLIST_INSERTUPDATE
ON GOODSLIST FOR INSERT, UPDATE
AS
BEGIN
    DECLARE @AFPRICE
    SELECT @AFPRICE =FACTORYPRICE FROM INSERTED

    IF @AFPRICE<=0 -- проверка корректности цены
    BEGIN
        IF @@TRANCOUNT>0 ROLLBACK TRAN -- откат транзакции
        RAISERROR(100106,16,10) -- несоответствие цены!
    END
END
```

Для решения изложенной выше задачи мы опрашиваем логическую таблицу INSERTED с целью узнать стоимость единицы продукции. И если цена не превышает 0, то осуществляем откат транзакции. По правилам хорошего тона программирования при откате транзакции следует сгенерировать сообщение об ошибке – эта задача возложена на функцию RAISERROR().

Рассмотрим еще один пример, на этот раз защиты строки от несанкционированного удаления. В демонстрационной БД (приложение 1) описание складов хранится в таблице WAREHOUSES. Строка с информацией об основном складе заносится в эту таблицу еще на этапе введения БД в эксплуатацию и получает признак склада по умолчанию (DEF=true), на который автоматически поступают все новые товары. Склад по умолчанию удалять нельзя, иначе разрушится бизнес-правило оприходования товаров. Как мы станем защищать эту запись в InterBase? Естественно, с помощью триггера (листинг 14.9).

Листинг 14.16. Запрет на удаление определенной записи

```
CREATE TRIGGER T_WAREHOUSES_BD FOR WAREHOUSE
ACTIVE BEFORE DELETE POSITION 0
AS
BEGIN /*нельзя удалить основной склад DEF=TRUE*/
```



```
if (OLD.DEF=TRUE) THEN EXCEPTION E_DEFAULTWAREHOUSE_EXCEPTION;
END
```

Как видите, попытка удаления склада по умолчанию приведет к генерации исключительной ситуации и откату транзакции.

Замечание

Строго говоря, предложенный в листинге 14.16 пример удаления строки из таблицы складов не завершен. Для полноты картины было бы неплохо в рамках триггера еще проверять, нет ли за складом закрепленных товаров, и запрещать удаление строки при наличии таковых.

В базах данных InterBase триггеры зачастую применяются для определения значений для столбцов первичного ключа. Для этого сначала создаются генераторы значений (листинг 14.17).

Листинг 14.17. Создание и инициализация генератора с помощью SQL

```
CREATE GENERATOR G_WAREHOUSES_ID;
SET GENERATOR G_WAREHOUSES_ID TO 1;
```

В этом примере в первой строчке в базе данных создается генератор с именем G_WAREHOUSES_ID, а затем этому генератору присваивается стартовое значение 1.

Для обращения к генератору предусмотрена функция:

```
GEN_ID(имя_генератора, шаг_приращения);
```

Названная функция обладает двумя параметрами, в которые передаются имя генератора и шаг приращения значения. Для того чтобы подготовленное генератором число попало в столбец первичного ключа таблицы, функция GEN_ID() вызывается в триггере (листинг 14.10).

Листинг 14.18. Генерация значения ключа в триггере таблицы

```
CREATE TRIGGER T_WAREHOUSE_BI_GEN
FOR WAREHOUSES
ACTIVE BEFORE INSERT POSITION 0
AS
BEGIN
if (NEW.WAREHOUSES_ID IS NULL) THEN
    NEW.WAREHOUSES_ID=GEN_ID(G_WAREHOUSES_ID, 1);
END;
```

Еще одна область применения триггеров – ведение аудита изменений в наиболее важных таблицах базы данных. Допустим, что намерены отслеживать все операции, связанные с таблицей контрактов CONTRACTS, в таком случае в СУБД Oracle мы смогли бы создать триггер, представленный в листинге 14.11.

Листинг 14.19. Ведение протокола изменений в таблице

```
CREATE TRIGGER TR_CONTRACT_LOG
  AFTER DELETE OR INSERT OR UPDATE ON CONTRACTS
  FOR EACH ROW
BEGIN
  INSERT INTO CONTRACT_LOG
    (USERNAME, EVENTDATE, OLDCONTRACT, NEWCONTRACT)
  VALUES
    (USER, SYSDATE, :OLD.CONTRACT, :NEW.CONTRACT)
END TR_CONTRACTS_LOG;
```

Помимо названия контракта, в таблице CONTRACTS_LOG мы сохраняем имя пользователя, изменявшего данные, и дату изменений.

Внимание!

Достоинство триггера – способность определения логики работы таблиц, недостаток – скрытость этой логики от конечного пользователя, последний может просто не понять, чего от него хочет таблица. Еще один повод для беспокойства в том, что триггеры способны заметно снизить производительность БД, особенно в случае, когда в таблицах редактируется или удаляется большое количество строк.

Триггер – это очень мощное оружие поддержки целостности и непротиворечивости данных, но далеко не всесильное. Поэтому работа с триггерами связана с рядом ограничений, о которых следует всегда помнить разработчику:

- в теле триггера запрещено вызывать инструкции явного управления транзакцией, другими словами, недопустимы команды START TRANSACTION, COMMIT и ROLLBACK;
- в триггере запрещено создавать (CREATE), изменять (ALTER) и удалять (DROP) элементы БД;
- в ряде СУБД из кода триггера не рекомендуется вызывать хранимые процедуры;

- будьте осторожны, используя внутри триггера инструкции INSERT, UPDATE и DELETE, в особенности если они нацелены на модификацию данных в той же самой таблице, к которой и принадлежит триггер, проблема усугубится, если, например, внутри триггера UPDATE будет вызываться инструкция UPDATE к этой же таблице, – необдуманные решения могут стать причиной вечной работы триггера.

Завершая тему, упомянем инструкции по изменению и удалению триггеров. Для редактирования блока кода, содержащегося в триггерах, следует воспользоваться инструкцией ALTER TRIGGER, а вопросами удаления ведает инструкция DROP TRIGGER.

Курсоры

Обычный, построенный на основе инструкции SELECT запрос сразу возвращает нам некоторый результирующий набор строк. Однако зачастую логика разрабатываемой БД требует дополнительной обработки каждой отдельной строки в результирующем наборе, на которую не способна команда SELECT. Специалисты выделяют следующие специфичные ситуации, при которых стоит обращаться за помощью к курсорам:

- 1) сложная логика запроса (наличие формул, необходимость обращения из запроса к процедурам или функциям, необходимость использования условных операторов и исключений);
- 2) операции по денормализации данных, когда из нескольких отношений следует собрать одно (например, в формате временной таблицы) или вернуть результаты в формате текстовой строки, документа XML и т. п.;
- 3) создание перекрестного запроса, допустим, выполняющего статистические расчеты по двум и более таблицам, которые позднее группируются в виде таблицы;
- 4) движение по иерархическому дереву.

В таких случаях на помощь приходит курсор (cursor), представляющий собой указатель на отдельный кортеж в отношении. С помощью курсора мы сможем последовательно перебирать строки полученного отношения, осуществляя с ними дополнительные операции, например редактирования.

Замечание

Курсоры применяются в программных конструкциях SQL, в которых требуется обеспечить поочередный доступ к строкам отношения. Преимущество строчного доступа в том, что он позволяет программисту работать с данными, опираясь на процедурные расширения SQL.

Для работы с курсорами последовательно применяется четыре команды SQL:

- 1) объявить курсор (DECLARE CURSOR);
- 2) открыть курсор (OPEN);
- 3) считать данные из курсора (FETCH);
- 4) закрыть курсор (CLOSE).

Все операции начинаются с объявления курсора. Процесс объявления весьма схож с организацией запроса данных на основе инструкции SELECT. Синтаксическая конструкция объявления курсора в SQL выглядит следующим образом:

```
DECLARE имя_курсора [SENSITIVE|INSENSITIVE|ASENSITIVE]
                    [[NO] SCROLL] CURSOR
FOR инструкция на основе SELECT
[FOR {READ ONLY | UPDATE [OF имя_столбца, ... ]}]
```

Следующее за именем курсора предложение SQL [SENSITIVE|INSENSITIVE|ASENSITIVE] определяет порядок доступа курсора к данным. Параметр SENSITIVE устанавливает такой режим работы, когда чтение данных курсором осуществляется непосредственно из таблицы (представления), благодаря этому курсор становится чувствительным ко всем вносимым другими пользователями изменениям в результирующем наборе данных. Значение INSENSITIVE, напротив, максимально снизит чувствительность курсора, с этой целью результирующий набор помещается в отдельную временную таблицу, недоступную другим пользователям. Наконец, режим ASENSITIVE указывает на то, что порядок взаимодействия курсора с данными определяется СУБД самостоятельно.

Замечание

Курсоры большинства СУБД по умолчанию работают в режиме INSENSITIVE. В этом случае курсор запоминает состояние БД на момент своего открытия. Для того чтобы увидеть изменения, сделанные другими пользователями, следует закрыть и повторно открыть курсор.

Параметр `SCROLL` говорит о том, что курсор способен извлекать строки в любом направлении (от первой к последней записи или, наоборот, от последней к первой), причем направление перемещения курсора допускается изменить в любой момент времени. Обратный параметр `NO SCROLL` допускает только последовательный доступ к строкам от первой к последней. По возможности разработчику БД стоит отдавать предпочтение однонаправленным курсорам, так как они менее ресурсоемкие.

Как всегда, у различных диалектов SQL имеются свои нюансы по работе с курсорами. Ряд из них нацелен на работу с однонаправленным курсором, пробегающим строки таблиц от первой к последней, ряд способен прокручивать курсоры в обоих направлениях.

Параметр `READ ONLY` запрещает любую модификацию данных. Использование ключевого слова `UPDATE [OF имя_столбца, ...]`, напротив, позволяет явно определять столбцы, которые могут быть обновлены.

Для получения права на считывание данных из курсора его следует открыть. Для этого предназначена команда

```
OPEN имя_курсора;
```

В ответ на эту команду СУБД выполняет принадлежащий курсору запрос, но не возвращает полученное отношение целиком, а только устанавливает курсор на первую строку результирующего набора строк в готовности ее прочитать.

Для извлечения данных из открытого курсора применяется предложение `FETCH`. Синтаксис инструкции:

FETCH

```
[{NEXT | PRIOR | FIRST | LAST {ABSOLUTE n | RELATIVE n}}]
```

FROM

```
имя_курсора
```

```
INTO имя_переменной1[, ...]
```

Если при объявлении курсора не использовалось ключевое слово `SCROLL`, то единственным способом считывания является последовательное прохождение полученного набора данных. Для нормального порядка считывания (от первой строки к последней) применяют ключевое слово `NEXT`. Если же во время декларации курсора вы указали, что считывание может осуществляться в любом порядке, то для обращения к предыдущей записи разрешается применять ключевое слово `PRIOR`. Инструкции `FIRST` и `LAST` соответственно перемещают курсор к самой первой и к самой последней записи в результирующем наборе. Ключевое слово

ABSOLUTE *n* определяет, что считывание начнется с *n*-й строки результирующего набора, **RELATIVE** *n* вызывает считывание *n*-й строки относительно текущей строки. Если аргумент положителен, то осуществляется перемещение вперед, при отрицательном значении – назад. Необязательное ключевое слово **FROM**, стоящее перед именем курсора, делает код более читабельным. За командой **INTO** следуют имена переменных, в которые передаются извлеченные из строки данные. Количество и тип переменных должны четко соответствовать количеству и типу столбцов, описанных в инструкции **SELECT** во время объявления курсора.

Завершив обработку всех строк, курсор закрывается, для этого обращаемся к инструкции

CLOSE имя_курсора;

Закрытие курсора освобождает все связанные с ним ресурсы, а также отключает все ранее установленные блокировки. Чтение из закрытого курсора невозможно.

Внимание!

Практически у всех СУБД при выполнении кода с курсорами несколько снижается производительность, поэтому там, где это возможно, старайтесь не использовать услуги курсоров или писать максимально компактный быстрый код.

Примеры курсоров

Практическую сторону применения курсоров вначале рассмотрим на простом примере для MySQL. Данный пример в дальнейшем можно использовать в качестве шаблона при решении более сложных задач. В хранимой процедуре из листинга 14.20 осуществляется обработка строк одной из таблиц демонстрационной БД. Для чтения данных задействуется петлевой цикл **LOOP**.

Листинг 14.20. Работа с курсором с контролем за ошибкой NOT FOUND

```
CREATE PROCEDURE pr_cursor_template1()
BEGIN
  DECLARE aGOODSLIST_ID INT;
  DECLARE aGOODS VARCHAR(100);
  DECLARE aFACTORYPRICE DECIMAL(10,2);
  DECLARE CUR_STOP INT DEFAULT FALSE; #переменная для контроля чтений

  DECLARE CUR CURSOR FOR #объявляем курсор
```

```

SELECT G.GOODSLIST_ID, G.GOODS, G.FACTORYPRICE FROM goodslist G;
#----- Обработчик ошибки попытки чтения несуществующей строки -----
DECLARE CONTINUE HANDLER FOR NOT FOUND SET CUR_STOP = TRUE;
#-----
OPEN CUR;
#----- Цикл построчного чтения -----
cur_loop: LOOP
  FETCH CUR INTO aGOODSLIST_ID, aGOODS, aFACTORYPRICE;

  IF CUR_STOP THEN
    LEAVE cur_loop; #выход из цикла
  END IF;
  ***** Обработка данных *****
  IF (aFACTORYPRICE>10000) THEN # если цена за единицу более 100 000, тогда
  BEGIN
    #...
    SELECT aGOODSLIST_ID, aGOODS, aFACTORYPRICE;
  END; END IF;
  *****
  END LOOP;
#----- Конец цикла -----
CLOSE CUR; #закрываем курсор

```

Обратите внимание на то, что для отслеживания попытки чтения данных за пределами набора нами введены специальная переменная CUR_STOP и обработчик исключительной ситуации HANDLER. Ключ CONTINUE разрешает продолжить выполнение кода после возникновения ошибки NOT FOUND, иначе хранимая процедура прервется. В момент обработки ИС мы устанавливаем переменную CUR_STOP в состояние true, что, в свою очередь, является условием выхода из петлевого цикла.

При желании программист может вообще не допустить чтений за пределами отношения. Для этого достаточно перед обращением к оператору FETCH узнать количество подлежащих обработке строк и ввести счетчик чтений (листинг 14.21).

Листинг 14.21. Работа с курсором со счетчиком чтений

```

CREATE PROCEDURE pr_cursor_template2()
BEGIN
  DECLARE aGOODSLIST_ID, RECCOUNT, NUM INT DEFAULT 0;

```

```

DECLARE CUR CURSOR FOR SELECT G.GOODSLIST_ID FROM goodslist G;

SELECT COUNT(*) FROM goodslist INTO RECCOUNT; #узнали число строк

OPEN CUR; #открыли курсор
#----- Цикл построчного чтения -----
WHILE (NUM<=RECCOUNT-1) DO -- цикл выполняется, пока NUM меньше и равно RECCOUNT
    FETCH CUR INTO aGOODSLIST_ID; #чтение строки
    IF (...) THEN
        # обработка данных
    END IF;
    SET NUM:=NUM+1; #приращение счетчика
END WHILE;
#-----
CLOSE CUR; #закрыли курсор

```

Сервер MySQL допускает, что в одной процедуре (функции) может совместно работать несколько курсоров. Более того, цикл чтения одного курсора вполне может быть вложен в цикл чтения другого курсора. Подобный пример представлен в листинге 14.22. Хранимая процедура предназначена для обслуживания таблицы GOODSCCLASS с классификатором товара, напомним, что в таблице между строками организованы рекурсивные связи «главный–подчиненный» (приложение 1). Заполняя классификатор, пользователь может размещать в нем узлы в любом порядке, для этой цели в таблицу введен столбец NODEINDEX, содержащий целочисленное значение сортировки. Но если пользователь захочет переупорядочить узлы классификатора по алфавиту, то в этом ему окажет помощь наша процедура – она пересчитает значения NODEINDEX так, чтобы, с одной стороны, обеспечить сортировку по алфавиту, а с другой – не нарушить рекурсивные связи.

Листинг 14.22. Два курсора в хранимой процедуре

```

CREATE PROCEDURE pr_goodsclass_sortbyalpha()
BEGIN
    DECLARE aINDEX, aPARENT_ID, aGOODSCCLASS_ID, bGOODSCCLASS_ID INT DEFAULT 0;
    DECLARE CUR_STOP INT DEFAULT FALSE; #переменная для контроля чтений

    DECLARE GLOBAL_CUR CURSOR FOR # "внешний" курсор
        SELECT GOODSCCLASS_ID, PARENT_ID FROM goodsclass WHERE NODEINDEX IS NULL

```



```
ORDER BY PARENT_ID, goodsclass;

DECLARE LOCAL_CUR CURSOR FOR # "вложенный" курсор
  SELECT GOODSCLASS_ID FROM goodsclass
  WHERE (PARENT_ID=aGOODSCLASS_ID) AND (NODEINDEX IS NULL)
  ORDER BY goodsclass;

#----- Обработчик ошибки попытки чтения несуществующей строки -----
DECLARE CONTINUE HANDLER FOR NOT FOUND SET CUR_STOP = TRUE;
#-----
# очищаем всю предыдущую сортировку
UPDATE goodsclass SET NODEINDEX=NULL WHERE GOODSCLASS_ID>0;

OPEN GLOBAL_CUR;
#----- Внешний курсор -----
global_cur_loop: LOOP
  FETCH GLOBAL_CUR INTO aGOODSCLASS_ID, aPARENT_ID;

  IF CUR_STOP THEN
    LEAVE global_cur_loop; #выход из внешнего цикла
  END IF;
  SET aINDEX=aINDEX+1; #приращение счетчика сортировки
  UPDATE goodsclass SET NODEINDEX=aINDEX WHERE GOODSCLASS_ID=aGOODSCLASS_ID;
#***** Вложенный курсор *****
  OPEN LOCAL_CUR;
  local_cur_loop : LOOP
    FETCH LOCAL_CUR INTO bGOODSCLASS_ID;
    IF CUR_STOP THEN
      BEGIN
        SET CUR_STOP=FALSE;
        LEAVE local_cur_loop; #выход из цикла
      END; END IF;

    SET aINDEX=aINDEX+1;

    UPDATE goodsclass SET NODEINDEX=aINDEX
      WHERE GOODSCLASS_ID=bGOODSCLASS_ID;
  END LOOP;

CLOSE LOCAL_CUR; #закрываем вложенный курсор
```

```

#*****
END LOOP;
#----- Конец внешнего цикла -----
CLOSE GLOBAL_CUR; #закрываем внешний курсор

```

Принцип работы исходного кода хранимой процедуры следующий. Объявляется два курсора: внешний GLOBAL_CUR и внутренний LOCAL_CUR. Внешний курсор станет просматривать все строки таблицы, не имеющие индекса сортировки (NODEINDEX IS NULL), упорядоченные в алфавитном порядке. Внутренний курсор будет просматривать только дочерние строки по отношению к строке, выбранной глобальным курсором (также упорядоченные в алфавитном порядке). С каждой итерацией как внешнего, так и внутреннего циклов мы модифицируем значение столбца NODEINDEX текущей строки, передавая в нее значение переменной-счетчика aINDEX.

Обратите внимание на еще одну особенность примера. Несмотря на то что мы работаем с двумя курсорами, для контроля за выход за пределы отношения достаточно одного обработчика исключительной ситуации, управляющего состоянием переменной CUR_STOP. В зависимости от того, в каком из циклов была осуществлена попытка некорректного чтения, мы обращаемся к нужной метке и иницилируем выход из цикла.

Внимание!

Старайтесь писать такой код, чтобы все операции с курсорами выполнялись в максимально сжатые сроки. Никогда не стоит долго держать курсор в открытом состоянии – в SQL-программировании это признак дурного тона.

Резюме

Все наиболее важные корпоративные ограничения и бизнес-правила БД должны сосредотачиваться не в клиентских приложениях, а исключительно на стороне сервера. Соблюдая это правило, разработчик БД не только поднимет на новый уровень целостность и безопасность данных, но и существенно упростит свой труд. Ведь при таком подходе вместо постоянной переработки кода многочисленных клиентов окажется достаточным направить усилия только на БД.

Для описания логики работы БД в SQL предусмотрен гибкий инструмент – процедурный SQL, позволяющий создавать хранимые процедуры, функции и триггеры. В рамках процедурного SQL разработчик БД

получает возможность не только задействовать базовые инструкции SQL, но и использовать такие расширения языка, как переменные, циклы, условный оператор и события.

Вопросы для самопроверки

1. Что, на ваш взгляд, послужило причиной дополнения языка SQL процедурными возможностями?
2. Какие преимущества дают хранимые процедуры и функции?
3. Раскройте особенности применения условного оператора IF.
4. Какие формы оператора-селектора CASE могут использоваться в SQL?
5. Какие виды циклов поддерживаются в SQL?
6. Существует ли возможность прервать цикл или пропустить итерацию цикла?
7. Приведите синтаксическую конструкцию:
 - а) хранимой процедуры;
 - б) функции;
 - с) триггера.
8. Каким образом осуществляется вызов хранимых процедур и функций?
9. Почему вызов триггеров осуществляется автоматически?
10. Прокомментируйте правило «событие – условие – действие» для триггера.
11. Почему вызов триггера должен осуществляться автоматически?
12. На какие события должен уметь реагировать триггер?
13. Какие задачи возложены на контекстные переменные триггера?
14. Чем отличается триггер уровня кортежа от триггера уровня команд?
15. Для чего предназначены курсоры?
16. Какие инструкции SQL понадобятся для работы с курсором?

Глава 15

Регулярные выражения в запросах

Регулярные выражения – это эффективный способ проверки соответствия текста заданному шаблону. За счет специально разработанных алгоритмов обработка текста осуществляется с максимально возможной производительностью, поэтому регулярные выражения широко используются всеми программистами.

Широкую популярность регулярные выражения приобрели с появлением операционной системы Unix. Разработчики системы (среди них такие знаменитости, как Брайан Керниган, Деннис Ритчи, Кен Томпсон) включили в состав ОС утилиты, эффективно работающие с текстом. Позднее регулярные выражения перекочевали из Unix и в другие системы, в том числе и в некоторые СУБД.

Замечание

К сожалению, далеко не все современные СУБД способны работать с регулярными выражениями, к счастливым относятся такие системы, как Oracle, PostgreSQL и MySQL. В этой главе для изучения регулярных выражений мы воспользуемся услугами MySQL.

Операторы для регулярных выражений

По большому счету, вместо слова «операторы», предполагающего существование нескольких операторов, обслуживающих регулярные выражения в MySQL, можно ограничиться единственным числом – «оператор». Так как по своей сути со всеми задачами справляется оператор REGEXP, его синтаксис выглядит следующим образом:

текст [NOT] REGEXP шаблон

Однако в MySQL имеется еще один оператор `RLIKE` (его название подчеркивает, что он способен заменить ключевое слово `LIKE` в запросе `SELECT`), делающий то же самое и обладающий тем же самым синтаксисом, что и `REGEXP`. Так что с этого момента станем считать, что `RLIKE` и `REGEXP` – просто синонимы.

Регулярное выражение анализирует текст на предмет его соответствия шаблону и возвращает 1 или 0. Как вы уже поняли, 1 является признаком того, что текст соответствует правилу, определенному шаблоном, а 0 – не соответствует. В ситуации, когда на вход регулярному выражению подается `NULL`, результатом станет также `NULL`.

Основы синтаксиса

По своей сути регулярные выражения – это микроязык программирования, построенный на относительно несложных правилах [14, 46]. На основе этих правил программист (а может быть, и подготовленный пользователь) создает шаблон и анализирует целевой текст. Самый простейший шаблон (т. е. регулярное выражение) представляет собой последовательность символов, допустим «ABCDE», и если это выражение использовать для проверки текста, где хотя бы раз встречается «ABCDE», то «механизм» регулярных выражений возвратит положительный результат.

Поиск совпадений на основе простого соответствия текста образца и исходного текста – это самое простое, что можно сделать с помощью рассматриваемого в этой главе механизма. Однако помимо обычных символов в регулярные выражения могут входить специальные метасимволы (представленные в табл. 15.1), которые распознаются в качестве управляющих. С помощью них можно творить если не чудеса, то, по крайней мере, весьма неординарные вещи.

Таблица 15.1. Основные метасимволы для регулярных выражений

Метасимвол	Описание
<code>^</code>	Привязка к началу строки
<code>\$</code>	Привязка к окончанию строки
<code>.</code>	Любой произвольный символ
<code>\\</code>	Рассматривать следующий за «\» метасимвол как обычный символ
<code> </code>	Логическое или
<code>[]</code>	Класс символов
<code>()</code>	Группировка

Метасимвол «^» указывает на то, что поиск соответствия должен осуществляться только в начале строки. Например, регулярное выражение на основе шаблона «^ABC» возвратит истину (точнее говоря, результат, равный 1) при проверке строки «ABCDEF» и останется равнодушным к строке «ZABCDEF».

Метасимвол «\$» изменяет условие с точностью до наоборот, теперь поиск соответствия осуществляется только в конце строки. На этот раз при проверке строки «ABCDEF» единичку мы получим, сформировав шаблон «DEF\$» (рис. 15.1).

```

mysql> SELECT 'ABCDEF' REGEXP '^ABC', 'FEDCBA' REGEXP '^ABC';
+-----+-----+
| 'ABCDEF' REGEXP '^ABC' | 'FEDCBA' REGEXP '^ABC' |
+-----+-----+
| 1 | 0 |
+-----+-----+
1 row in set (0.11 sec)

mysql> SELECT 'ABCDEF' REGEXP 'DEF$', 'FEDCBA' REGEXP 'DEF$';
+-----+-----+
| 'ABCDEF' REGEXP 'DEF$' | 'FEDCBA' REGEXP 'DEF$' |
+-----+-----+
| 1 | 0 |
+-----+-----+
1 row in set (0.00 sec)

mysql> _
  
```

Рис. 15.1. Метасимволы «^» и «\$» в регулярных выражениях

Метасимвол «.» призван замещать любой одиночный символ. Таким образом шаблон «DE.\$» станет подходить к строкам «ABCDEA», «ABCDEB», «ABCDEС» и т. д. Заметьте, что в этом примере мы задействовали два метасимвола «.» и «\$».

При построении выражений особой популярностью пользуется метасимвол обратной наклонной черты «\». У него две роли. Во-первых, метасимвол «\» в сочетании с рядом обычных символов (табл. 15.2) может обозначать знаки табуляции, перевода строки и т. п.

Таблица 15.2. Дополнительные метасимволы

Метасимвол	Описание	Метасимвол	Описание
\t	Символ горизонтальной табуляции	\b	Удаление
\n	Новая строка	\v	Вертикальная табуляция
\x	Перевод каретки	\f	Перевод страницы

Во-вторых, если вы встретите две наклонные черты подряд «\», то это означает, что вслед за ними расположен литерал (обычный символ). Исходя из вышесказанного, для проверки окончания текстовой строки «ABCDEF.», которую завершает обычная точка (а не метасимвол «.»), подойдет шаблон «\\$.» (рис. 15.2).

```

mysql> SELECT 'ABC' REGEXP '\\.$', 'ABC.' REGEXP '\\.$';
+-----+-----+
| 'ABC' REGEXP '\\.$' | 'ABC.' REGEXP '\\.$' |
+-----+-----+
| 0 | 1 |
+-----+-----+
1 row in set (0.00 sec)

mysql>
  
```

Рис. 15.2. Превращение метасимвола «.» в обычную точку в регулярных выражениях

Для осуществления операции дизъюнкции (логического ИЛИ) используется метасимвол «|». Если при проверке начала строки в качестве первых символов нас устраивает как «AB», так и «BC», то воспользуемся шаблоном «^AB|BC». В этом случае регулярное выражение возвратит истину для строк: «AB» и «BC» (рис. 15.3).

```

mysql> SELECT 'AB' REGEXP '^AB|BC', 'BC' REGEXP '^AB|BC', 'CD' REGEXP '^AB|BC';
+-----+-----+-----+
| 'AB' REGEXP '^AB|BC' | 'BC' REGEXP '^AB|BC' | 'CD' REGEXP '^AB|BC' |
+-----+-----+-----+
| 1 | 1 | 0 |
+-----+-----+-----+
1 row in set (0.00 sec)

mysql>
  
```

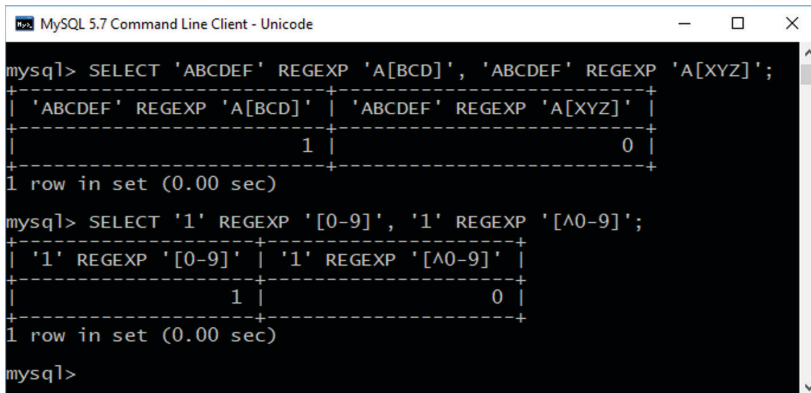
Рис. 15.3. Операция логического «ИЛИ» в регулярных выражениях

Для описания класса символов задействуются квадратные скобки. Классы предназначены для определения некоего набора символов, с которым мы рассчитываем найти совпадения. Внутри класса (читай: внутри квадратных скобок) символы могут определяться:

- по отдельности, например «[A]» означает, что мы ищем в образце «A»;
- перечислением «[ABC]» – означает, что мы ищем в образце «A», или «B», или «C»;
- диапазоном «[A-C]» – аналог «[ABC]».

Еще одна особенность класса – в том, что внутри квадратных скобок метасимволы из табл. 15.1 (за исключением «^») утрачивают свою роль и превращаются в обычные символы. Посему если вы встретите запись «[A\$]», это означает, что мы разыскиваем в тексте символы «A» или «\$».

В определении класса может быть задействован метасимвол «^», но его назначение изменяется. Внутри квадратных скобок метасимвол «^» вместо привязки к началу строки превращается в признак отрицания. Например, выражение «[^0-9]» находит совпадение с любым символом, кроме цифровых (рис. 15.4).



```

mysql> SELECT 'ABCDEF' REGEXP 'A[BCD]', 'ABCDEF' REGEXP 'A[XYZ]';
+-----+-----+
| 'ABCDEF' REGEXP 'A[BCD]' | 'ABCDEF' REGEXP 'A[XYZ]' |
+-----+-----+
| 1 | 0 |
+-----+-----+
1 row in set (0.00 sec)

mysql> SELECT '1' REGEXP '[0-9]', '1' REGEXP '[^0-9]';
+-----+-----+
| '1' REGEXP '[0-9]' | '1' REGEXP '[^0-9]' |
+-----+-----+
| 1 | 0 |
+-----+-----+
1 row in set (0.00 sec)

mysql>

```

Рис. 15.4. Квадратные скобки в регулярных выражениях

Метасимволы могут сопровождаться модификаторами (табл. 15.3), которые определяют, какое число раз должна повторяться часть регулярного выражения.

Таблица 15.3. Модификаторы повторений

Модификатор	Описание
*	0 или больше повторений
+	1 или больше повторений
?	0 или 1 повторение
{n}	точно n повторений
{n,}	не менее n повторений
{n,m}	не меньше n, но и не больше m повторений

Модификатор * указывает на то, что предыдущий символ может быть повторен 0 или неограниченное число раз (позволю себе предположить, что верхний предел повторений устанавливается величиной типа

данных integer). Таким образом, встретив строку «AB*С», алгоритм регулярных выражений будет анализировать образец на предмет наличия следующих вхождений: «АС», «ABC», «ABBC», «ABBBС» и т. д.

Представленные в табл. 15.3 модификаторы могут применяться не только к отдельным символам, но и к более сложным конструкциям, например «A[BCD]+A». Что будет делать анализатор выражения в таком случае? Он станет искать в образце все комбинации: «ABA», «ACA», «ADA», «ABBA», «ACCA», «ADDA», «ABVBA» и т. д.

Поместив в круглые скобки какую-то комбинацию символов и оснастив ее модификатором из табл. 15.3 (допустим, «(AB)+»), мы укажем, что в целевом тексте ищем определенное число повторений данной группы.

Наших знаний уже вполне достаточно, чтобы создать регулярное выражение, проверяющее корректность ввода 10-значного телефонного номера, подобного следующему: (495)-000-0001. Это выражение может выглядеть таким образом:

$$(\backslash([0-9]{3}\backslash)-[0-9]{3}-[0-9]{4})$$

Если разложить выражение «по полочкам» (если говорить по-научному – провести посимвольный анализ), то получится примерно следующее:

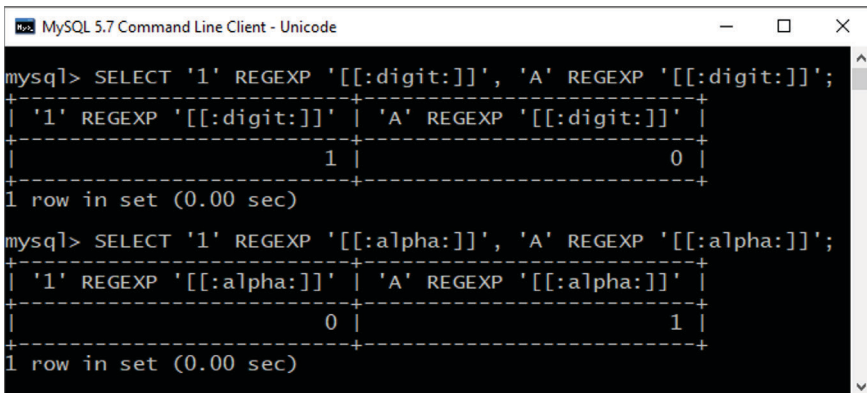
- (– открывает группу;
- \ – экранирует открывающуюся круглую скобку «(» регионального кода телефона;
- [0-9] – допустим только цифровой символ;
- {3} – три повтора;
- \ – экранирует круглую скобку «)», завершающую региональный код телефона;
- - – литеральный дефис;
- [0-9] – допустим только цифровой символ;
- {3} – три повтора;
- - – литеральный дефис;
- [0-9] – допустим только цифровой символ;
- {4} – четыре повтора;
-) – закрывает группу.

Заметим, что это далеко не самое идеальное решение, при желании можно написать еще более сложное выражение, например предполагающее необязательность ввода регионального кода. Но для уяснения возможностей регулярных выражений вполне достаточно.

В регулярных выражениях MySQL поддерживаются классы символов POSIX. Имя, заключенное внутри выражения в скобках `[ :имя_класса: ]`, обозначает список всех символов, принадлежащих данному классу:

- `[ :alnum: ]` – алфавитно-цифровые символы;
- `[ :alpha: ]` – символы алфавита;
- `[ :blank: ]` – символы пробела и табуляции;
- `[ :cntrl: ]` – управляющие символы;
- `[ :digit: ]` – десятичные цифры (0–9);
- `[ :graph: ]` – графические (видимые) символы;
- `[ :lower: ]` – символы алфавита в нижнем регистре;
- `[ :print: ]` – графические или невидимые символы;
- `[ :punct: ]` – знаки препинания;
- `[ :space: ]` – символы пробела, табуляции, новой строки или возврата каретки;
- `[ :upper: ]` – символы алфавита в верхнем регистре;
- `[ :xdigit: ]` – шестнадцатеричные цифры.

Пример работы с группами символов иллюстрирует рис. 15.5. В первом примере с этого экранного символа мы проверяем наличие одиночного цифрового, а во втором – одиночного алфавитного символа.



```

mysql> SELECT '1' REGEXP '[:digit:]', 'A' REGEXP '[:digit:]';
+-----+-----+
| '1' REGEXP '[:digit:]' | 'A' REGEXP '[:digit:]' |
+-----+-----+
| 1 | 0 |
+-----+-----+
1 row in set (0.00 sec)

mysql> SELECT '1' REGEXP '[:alpha:]', 'A' REGEXP '[:alpha:]';
+-----+-----+
| '1' REGEXP '[:alpha:]' | 'A' REGEXP '[:alpha:]' |
+-----+-----+
| 0 | 1 |
+-----+-----+
1 row in set (0.00 sec)
  
```

Рис. 15.5. Группы символов POSIX в регулярных выражениях

Внимание!

Классы POSIX предназначены для использования в классах символов, так что заключайте их в группирующие квадратные скобки, например `[ :alnum: ]`.

Работу фигурных скобок, предназначенных для указания количества повторов символов, может проиллюстрировать экранный снимок, предложенный на рис. 15.6. Например, выражение `AB{1,2}` будет соответствовать строке, в которой за «А» следует от 1 до 2 символов «В». А выражению `A(BC){2,3}` сопоставляются строки, в которых за символом «А» следует от 2 до 3 пар символов «BC».

```
mysql> SELECT 'AC' REGEXP 'AB{1,2}', 'ABC' REGEXP 'AB{1,2}', 'ABBC' REGEXP 'AB{1,2}';
+-----+-----+-----+
| 'AC' REGEXP 'AB{1,2}' | 'ABC' REGEXP 'AB{1,2}' | 'ABBC' REGEXP 'AB{1,2}' |
+-----+-----+-----+
| 0 | 1 | 1 |
+-----+-----+-----+
1 row in set (0.04 sec)

mysql> SELECT 'ABC' REGEXP 'A(BC){2,3}', 'ABCBC' REGEXP 'A(BC){2,3}';
+-----+-----+
| 'ABC' REGEXP 'A(BC){2,3}' | 'ABCBC' REGEXP 'A(BC){2,3}' |
+-----+-----+
| 0 | 1 |
+-----+-----+
1 row in set (0.00 sec)
```

Рис. 15.6. Применение фигурных скобок в регулярных выражениях

При проектировании регулярных выражений существенный интерес представляет пара маркеров: `[[:<:]]` и `[[:>:]]`. С помощью них обозначаются границы слова. Слово может состоять только из алфавитно-цифровых символов и символа подчеркивания. Слово не могут предшествовать никакие алфавитно-цифровые символы, а также после слова не должны следовать никакие алфавитно-цифровые символы. На рис. 15.7 рассмотрен поиск слова «BCD» с помощью регулярного выражения `[[:<:]]BCD[[:>:]]` в трех вариантах текстовых строк.

```
mysql> SELECT 'A BCD E' REGEXP '[[:<:]]BCD[[:>:]]';
+-----+
| 'A BCD E' REGEXP '[[:<:]]BCD[[:>:]]' |
+-----+
| 1 |
+-----+
1 row in set (0.00 sec)

mysql> SELECT 'A-BCD-E' REGEXP '[[:<:]]BCD[[:>:]]';
+-----+
| 'A-BCD-E' REGEXP '[[:<:]]BCD[[:>:]]' |
+-----+
| 1 |
+-----+
1 row in set (0.00 sec)

mysql> SELECT 'A BCDE D' REGEXP '[[:<:]]BCD[[:>:]]';
+-----+
| 'A BCDE D' REGEXP '[[:<:]]BCD[[:>:]]' |
+-----+
| 0 |
+-----+
1 row in set (0.00 sec)
```

Рис. 15.7. Проверка на вхождение в текст отдельного слова с помощью регулярного выражения

Ко всему прочему при посредничестве метасимволов группировки разработчику позволено создавать внутри тела шаблона логические конструкции, простейшие из которых предложены в табл. 15.4. Особо отметим, что возможности круглых скобок гораздо шире, чем удалось рассказать в этой главе, поэтому более подробную информацию следует искать в документации к MySQL (или к Perl).

Таблица 15.4. Логические выражения

Выражение	Описание
(?=шаблон)	Требует, чтобы после этой точки находился фрагмент текста, который соответствует шаблону
(?!шаблон)	После этой точки не должно быть соответствующего шаблону текста
(?<=шаблон)	Перед этой точкой должен находиться соответствующий шаблону текста
(?<!шаблон)	Перед этой точкой не должно быть текста, соответствующего шаблону
(?#текст)	Комментарий
(?:шаблон)	Группировка элементов шаблона

Регулярные выражения в запросах

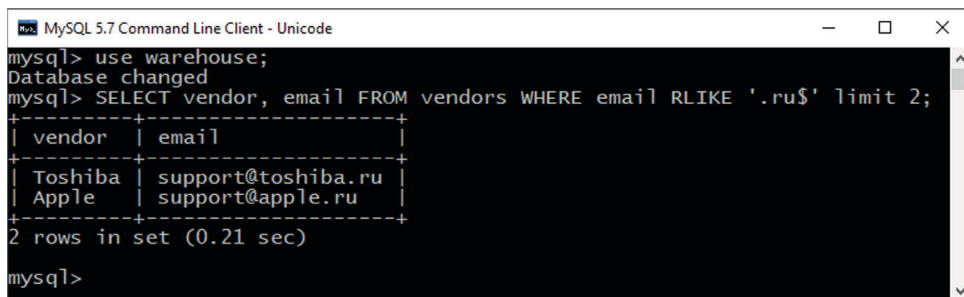
Одному из блестящих ИТ-специалистов и разработчику программных продуктов с открытым исходным кодом Джейми Завински (Jamie Zawinski) приписывают следующую остроумную фразу: «Некоторые люди, столкнувшись с проблемой, думают: “Ну-ка воспользуюсь я регулярными выражениями”. Теперь у них есть две проблемы...» Возможно, это так, но только не в случае, если вы намерены воспользоваться регулярными выражениями в SQL.

Основное место применения регулярных выражений – обработка данных, возвращаемых запросами выборки SELECT. Например, листинг 15.1 демонстрирует порядок выборки строк из таблиц производителей продукции, у которых имеются почтовые ящики в домене .ru.

Листинг 15.1. Выборка почтовых ящиков в домене .ru

```
SELECT vendor, email FROM vendors WHERE email RLIKE '.ru$ ';
```

Результат выполнения запроса представлен на рис. 15.8.



```

mysql> use warehouse;
Database changed
mysql> SELECT vendor, email FROM vendors WHERE email RLIKE '.ru$' limit 2;
+-----+-----+
| vendor | email |
+-----+-----+
| Toshiba | support@toshiba.ru |
| Apple   | support@apple.ru   |
+-----+-----+
2 rows in set (0.21 sec)

mysql>

```

Рис. 15.8. Выборка почтовых адресов из домена .ru

В ситуации, когда необходимо осуществить выбор по наличию слова в текстовой строке, можно воспользоваться примером из листинга 15.2.

Листинг 15.2. Выборка по совпадению слова

```
SELECT * FROM goodslist WHERE GOODS RLIKE '[:<:]]Смартфон[:>:]';
```

В данном случае мы выбираем из перечня продукции все товары, в описании которых встречается слово «смартфон».

Резюме

В программировании регулярные выражения появились в начале 1970-х годов вместе с выходом операционной системы Unix. По своей сути регулярные выражения – это обычные текстовые строки, используемые в качестве образцов при осуществлении операций сравнения. Задача регулярных выражений – упрощение поиска определенных образцов текста.

Механизм регулярных выражений поддерживается в СУБД: Oracle, PostgreSQL и MySQL. Выражения существенно расширяют возможности оператора SELECT, позволяя создавать нетривиальные запросы выборки.

Вопросы для самопроверки

1. Что такое регулярные выражения?
2. Какие метасимволы могут применяться в регулярных выражениях?
3. Как задать соответствие строковым литералам?
4. Как применять логические выражения в регулярных выражениях?

5. Как задать соответствие тексту в начале (конце) строки?
6. Как задать соответствие необязательному символу?
7. Для чего в регулярных выражениях применяются классы?
8. Как задать соответствие цифрам с помощью символьного класса?
9. Предложите регулярное выражение для проверки:
 - a) номера и серии паспорта;
 - b) номера зачетной книжки;
 - c) номера государственной регистрации для автомобиля;
 - d) номера банковской кредитной карты.
10. Как применять регулярные выражения в запросе SELECT?

Глава 16

Управление транзакциями

В большинстве современных СУБД единственным способом доступа к строкам таблиц является организация обращений пользователей к данным только в рамках особой логической единицы работы, именуемой транзакцией (transaction).

Определение

Транзакцией называется действие или набор действий, направленных на получение либо обновление данных и выполняемых как единая процедура. Транзакция не может быть выполнена частично, она либо выполняется полностью, либо полностью отменяется.

Различают три основных типа транзакций.

1. Транзакции извлечения применяются для выборки данных для их последующего отображения или печати.
2. Транзакции обновления работают во время вставки, модификации или удаления кортежей из таблиц.
3. Смешанные транзакции допускают проведение операций чтения и редактирования записей.

После старта транзакции промежуточные результаты всех ее действий обычно не записываются непосредственно в таблицы БД, а заносятся в журнал транзакций. Такой подход позволяет в любой момент времени вернуть БД в исходное состояние (состояние до старта транзакции). Откат транзакции осуществляется в том случае, если хотя бы один из операторов, входящих в состав транзакции, выполняется с ошибкой. Во время отката в обратном порядке отменяются все операции транзакции. Если же все операторы выполняются верно, то результаты транзакции фиксируются – данные из журнала переносятся в таблицы БД.

Для того чтобы СУБД смогла гарантировать корректное завершение или успешный откат транзакции, в журнале транзакций приходится хранить существенный объем данных:

- сведения о старте транзакции и ее идентификатор;
- перечень объектов БД, на которые повлияла транзакция;
- описание каждой из входящих в состав транзакции операций обновления данных (вставка, обновление, удаление);
- значения всех полей, изменившихся в результате выполнения транзакции, причем хранению подлежат как исходное, так и новое значение;
- сведения о завершении транзакции.

Таким образом, по своей сути журнал транзакций представляет собой самостоятельную базу данных, автоматически управляемую СУБД. На ведение журнала затрачивается достаточно большой вычислительный и дисковый ресурс сервера СУБД, но с такими издержками приходится мириться, ведь на другой чаше весов лежат требования по обеспечению надежного хранения основных данных.

Требования к транзакции

Любая транзакция обязана соответствовать ряду строгих требований, большинство специалистов по базам данных рекомендует список из четырех взаимозависимых элементов:

- **атомарность** (atomicity). Вне зависимости от количества вовлекаемых в транзакцию операторов, транзакция должна представлять собой единую и неделимую логическую единицу работы. Транзакция может быть выполнена полностью или не выполнена совсем. Представьте себе, что транзакция снимает некую сумму денег со счета и переводит ее в другой банк. Было бы обидно, если бы транзакция выполнялась частично: только сняла деньги со счета, а потом, не важно, по какой причине, не отправила на другой счет;
- **согласованность** (правильность) (consistency). Транзакция должна переводить базу данных из одного согласованного состояния в другое согласованное состояние. Однако в процессе выполнения транзакции допускается появление несогласованных промежуточных результатов. Говоря о согласованности результатов, мы имеем в виду, что в результате выполнения транзакции данные

будут соответствовать заложенной в базе данных логике и правилам поддержания целостности данных;

- **изолированность** (isolation). Транзакции должны выполняться независимо одна от другой, и ни одна из транзакций не должна быть допущена к промежуточным результатам другой транзакции. Для обеспечения изолированности каждая из запускаемых в СУБД транзакций выполняется в отдельном уникальном контексте, правила поведения которого могут назначаться как автоматически, так и выбираться программистом;
- **устойчивость** (долговечность) (durability). Результаты выполненной транзакции должны немедленно сохраняться в БД, чтобы не быть утерянными в результате вероятных последующих сбоев.

Замечание

Требования к транзакциям легко запомнить, для этого достаточно сохранить в своей памяти аббревиатуру из первых букв английских слов: atomicity, consistency, isolation, durability – ACID. Идея ACID была предложена Джимом Греем в середине 70-х, а аббревиатура появилась в начале 80-х.

В ряде посвященных базам данных работ [38] требование согласованности замещают требованием сериализуемости (serializability). Под сериализуемостью понимается возможность одновременного выполнения нескольких транзакций без их взаимного влияния, что очень важно для многопользовательских систем. Но если еще раз внимательно прочитать требования ACID, вы заметите, что они поглощают в себя и понятие сериализуемости. Справедливости ради отметим, что общепризнанный гурู Крис Дейт также критикует подход, основанный на принципах ACID [19]. Но это только подчеркивает тот факт, что современные информационные технологии находятся в состоянии постоянного совершенствования.

Состояние транзакции

Транзакция может находиться в одном из четырех состояний:

- 1) активном состоянии, т. е. в состоянии, соответствующем транзакции, находящейся в стадии выполнения, но пока не получившей команду на завершение или откат;
- 2) состоянии фиксации (committed) результатов выполнения;

- 3) состоянии отката (rolled back) к исходному состоянию;
- 4) неподтвержденной транзакции (in limbo).

Самое неприятное состояние, в котором может оказаться транзакция, – неподтвержденное. Неподтвержденная транзакция – это незавершенная транзакция, стартовавшая в режиме двухфазного подтверждения (two phase commit, 2PC), т. е. транзакция, чьи интересы затрагивают две (или более) БД. В неподтвержденном состоянии транзакция может находиться очень и очень долго, но одновременно нельзя исключить вероятность, что транзакция в самый неожиданный момент времени «проснется» и запросит фиксацию или откат. Так как СУБД далеко не всегда в состоянии однозначно определить факт «зависания» неподтвержденной транзакции, можно говорить, что для таких транзакций нет четкого механизма автоматического завершения или отката.

Проблемы совместного доступа к данным

При проектировании многопользовательской БД настоящей головной болью разработчика может стать организация совместного доступа двух и более пользователей к одним и тем же данным. В этом случае СУБД должна не только уметь корректно восстанавливать состояние БД при сбоях в транзакциях, но и обеспечивать бесконфликтную параллельную работу пользователей с общим ресурсом.

Для иллюстрации сложности организации многопользовательского доступа перечислим ключевые проблемы, с которыми могут столкнуться пользователи при совместном обращении к БД. К ним относятся:

- проблема **потерянных обновлений** (lost updates). Это тот случай, когда одна транзакция переписывает изменения, осуществленные другой транзакцией, в результате одно из изменений будет утеряно;
- проблема **неактуальных чтений** (зависимости от незафиксированных результатов). Имеет место, когда незафиксированные изменения, осуществленные одной транзакцией, читаются (или обновляются) другой. В случае перезаписи этих промежуточных значений или отката первой транзакции незафиксированные изменения могут быть отменены, а прочитавшая их транзакция с этого момента станет работать с неверными данными. Из-за этого проблему неактуальных чтений метко называли «грязным чтением» (dirty read) или «чтением мусора»;

- проблема **неповторяемого чтения** (non-repeatable read), или, как ее еще называют, проблема несогласованной обработки. Возникает тогда, когда транзакция считывает из базы значение, после чего вторая транзакция обновляет это значение. Если в этот момент времени первая транзакция продолжает выполняться, то имеющиеся в ее распоряжении данные становятся неактуальными;
- **чтение строк-фантомов** (row-phantom). Возникает в том случае, когда одна (продолжительная по времени) транзакция извлекает множество строк, а другая транзакция в это же время модифицирует ранее извлеченные строки. В результате получится, что в выборке первой транзакции окажутся модифицированные или удаленные строки.

Как устранить перечисленные проблемы и сделать так, чтобы транзакции не вредили друг другу? Ответ таков: надо добиться того, чтобы строки таблиц, необходимые для выполнения транзакции, были недоступны для других транзакций до тех пор, пока текущая транзакция не будет зафиксирована или отменена. При этом идеальным случаем было бы то, чтобы у транзакций создавалось впечатление, что они выполняются абсолютно независимо от других.

Управление параллельными транзакциями

Появление проблем многопользовательского доступа к данным объясняется одной-единственной причиной – неверным управлением параллельным доступом. Последствия некорректного доступа к данным могут быть самыми негативными. Так, потерянные обновления и неактуальные чтения переводят БД в несогласованное состояние, а неповторяемые чтения и чтения фантомов грозят выдачей пользователю некорректных результатов.

Определение

Параллельность – это такое качество СУБД, которое позволяет одновременно обращаться к той же базе данных нескольким транзакциям.

Поищем выход из сложившейся ситуации. На первый взгляд может показаться, что решение лежит на поверхности. Достаточно сделать так, чтобы в каждый момент времени выполнялась только одна транзакция, а

команда на запуск очередной транзакции выдавалась лишь после фиксации изменений, сделанных предыдущей транзакцией. Но переход к строго последовательному выполнению транзакций практически исключает реальный многопользовательский доступ к БД, что нас не устраивает.

На сегодняшний день наиболее рациональным решением, позволяющим минимизировать последствия от конфликтов многопользовательского доступа, считается разработка специального графика, упорядочивающего выполнение транзакций. Этот процесс называется планированием транзакций, им руководит планировщик транзакций (см. рис. 2.1). Целью планирования является отыскание такой упорядоченной последовательности выполнения операторов транзакций, чтобы транзакции могли работать совместно без оказания друг на друга взаимного влияния.

Внимание!

Основное правило совместного выполнения транзакций заключается в том, что в ходе обращения к БД транзакция должна видеть только согласованные данные, доступ к промежуточным или используемым другими данным должен быть заблокирован!

Особенности планирования параллельных транзакций во многом зависят по степени оптимизма разработчика БД. Пессимисты не исключают вероятности столкновения интересов транзакций и поэтому стремятся настолько, насколько это возможно, исключить конфликты между ними. Оптимисты исходят из предположения, что вероятность столкновения не столь высока, и не применяют превентивных мер по предотвращению столкновений до тех пор, пока эта неприятность все-таки не произойдет.

Вне зависимости от степени оптимизма надо учитывать, что если транзакции обслуживают разные части базы данных или только читают данные из одного и того же ресурса, то в организации их параллельного выполнения нет ничего сложного. Но если две транзакции обращаются к одним и тем же данным и хотя бы одна из них намерена их изменить, то это и есть потенциальный источник конфликта. На сегодня существует несколько подходов, позволяющих предотвратить споры транзакций за один и тот же ресурс, рассмотрим наиболее показательные из них.

Пессимистический подход

Существует несколько базовых пессимистических методов управления параллельными транзакциями, но практически все они в том или

ином виде опираются на идею блокировок строк таблиц на время выполнения транзакции.

Метод блокировок

Метод блокировок (lock) предполагает, что вероятность одновременного обращения нескольких транзакций к одному и тому же ресурсу относительно высока. Метод откладывает выполнение транзакций, потенциально готовых войти в конфликт с транзакциями, выполняющимися в текущий момент времени. Идея механизма блокировок прозрачна – после доступа к определенному ресурсу базы данных некой транзакции попытка обратиться к занятому ресурсу другими транзакциями отвергается, и опоздавшие транзакции переходят в режим ожидания освобождения ресурса.

Различают два вида блокировок:

- блокировка для операции записи. Ее называют **исключающей** (exclusive) **блокировкой**. Осуществившая такую блокировку транзакция может как читать, так и редактировать захваченный ей ресурс. Все остальные транзакции лишаются всех прав доступа к этим данным – в этом и суть исключения;
- блокировка для операции чтения, или **разделяемая** (shared) **блокировка**. Заблокировавшая ресурс транзакция намерена только прочесть данные, поэтому к ресурсу могут получить доступ и другие транзакции (ресурс разделяется между ними), при условии что они не станут его модифицировать.

Замечание

Блокировка для операции чтения допускает совместную работу с заблокированным ресурсом всех других транзакций с одним условием – никто не имеет право обновлять заблокированные данные. Блокировка для операции записи забирает ресурс в эксклюзивное пользование этой транзакцией.

Метод блокировок обладает одним существенным недостатком – он может стать причиной тупиковой ситуации, когда две (или более) транзакции попадают в состояние взаимного ожидания освобождения блокировок, удерживаемых каждой из них. Допустим, что две транзакции работают с одной и той же таблицей. Транзакция 1 обрабатывает записи таблицы в порядке их физического хранения в таблице (от первой к последней), а транзакция 2 – в обратном. Каждая из транзакций последовательно блокирует обслуживаемые строки и снимает блоки-

ровку, когда необходимость в ней отпадает. Где-то в середине таблицы транзакции встречаются друг с другом. Транзакция 1 удерживает блокировку на строке R_N и ожидает освобождения строки R_{N+1} . У транзакции 2 дела обстоят ровным счетом наоборот – она не в состоянии продолжить работу, пока не получит доступ к строке R_N , удерживаемой первой транзакцией, но при этом не может снять блокировку со строки R_{N+1} , так нужной транзакции 1. Если транзакции были бы настоящими джентльменами, то они моментально уступили бы друг другу дорогу. Но хорошим манерам их не учили, поэтому обе транзакции попадают в состояние **взаимной блокировки** (deadlock) и не могут из него выйти.

Для того чтобы нарушить состояние взаимной блокировки, СУБД приходится идти на радикальную меру – отменять выполнение одной или нескольких транзакций, чтобы хотя бы одна из транзакций завершилась успешно.

Метод временных меток

Метод временных меток представляет собой усовершенствованный протокол управления параллельностью, позволяющий устанавливать очередность выполнения транзакций, при которой более старые транзакции (транзакции с более поздним значением временной отметки) имеют более высокий приоритет при разрешении возникающих конфликтов. Для этих целей каждой транзакции в момент старта присваивается уникальная временная метка (time stamp).

Совсем не обязательно воспринимать термин «временная метка» дословно. В простейшем случае в качестве такой метки выступает обычный целочисленный идентификатор. Значение для идентификатора в момент запуска новой транзакции генерирует глобальный счетчик. Числовой идентификатор транзакции, основанный на идее автоинкрементного приращения, позволит не только именовать транзакцию, но и решит еще одну задачу. Сравнивая значения идентификаторов двух транзакций, СУБД легко узнает, какая из них является более старой, – это важный показатель, учитываемый при разрешении взаимных конфликтов. В этих конфликтах преимущество получит транзакция с более низким значением идентификатора.

Замечание

Метод временных меток не исключает взаимных блокировок, но значительно упрощает процесс принятия решения по выходу из патового положения. Теперь при столкновении интересов транзакций преимущество получает наиболее «древняя»,

а молодые транзакции откатываются в исходное состояние и последовательно запускаются СУБД после того, как конфликтная ситуация будет исчерпана.

Замечание

Существуют и другие подходы, применяемые при выводе транзакций из тупика. Например, в СУБД MySQL критерием такого решения выступает количество строк, которые должна обработать транзакция [36]. Чем больше строк должна модифицировать (удалить, добавить) транзакция, тем больше вероятность того, что она будет продолжена. А откату подлежит более легковесная транзакция. Сразу после отката все блокировки отмененной транзакции снимаются, что позволяет оставшейся транзакции продолжить свое выполнение.

Метод двухфазной блокировки

Для того чтобы снизить вероятность взаимных блокировок и, соответственно, исключить необходимость отката транзакций, программистами разработан **метод двухфазной блокировки** (two-phase locking).

Транзакция, работающая по алгоритму двухфазной блокировки, последовательно проходит два этапа: фазу роста и фазу сжатия. На этапе роста транзакция запрашивает у системы управления транзакциями СУБД все необходимые для ее выполнения блокировки, и до тех пор, пока эти блокировки не будут ей предоставлены, транзакция не осуществляет никаких операций с данными. Только получив все необходимое для работы, транзакция переходит к фазе сжатия – обрабатывает данные в таблицах и постепенно освобождает заблокированные элементы данных.

Благодаря тому что транзакция не начинает обработки данных до тех пор, пока они не будут освобождены другими транзакциями, попадание в тупик практически исключено. Правда, нахождение транзакции в первой фазе может оказаться достаточно долгим, но эту проблему должны решать такие компоненты СУБД, как планировщик и система управления транзакциями.

Оптимистический подход

Оптимистический подход к параллельному выполнению транзакций основан на предположении, что возможность конфликтов между выполняемыми в БД операциями ничтожно низка. Поэтому транзакцию можно выполнять до самого конца без каких-либо ограничений и медленных блокировок.

Предвидим законный вопрос: «А что, если случится невероятное и две транзакции, коварно нарушив расчеты оптимистов, все-таки набросятся на общий ресурс?» Сторонники оптимистического подхода к параллельному выполнению транзакций – просто оптимисты, но ни в коем случае не простаки.

В самом общем виде управление транзакцией в оптимистичном духе сводится к следующим этапам:

- **этап чтения.** Транзакция получает из таблиц интересующие ее данные и сохраняет их в кеш. Затем с копией данных осуществляются все необходимые операции;
- **этап проверки.** Система проводит проверку модифицированных данных на предмет целостности и непротиворечивости. При выявлении сбоев и ошибок транзакция отменяется, если же изменения осуществлены корректно – транзакция переходит к заключительному этапу;
- **этап сохранения.** Производится перенос данных из кеша в БД.

Оптимистические методы сериализации имеют одно весьма важное преимущество над их коллегами-пессимистами – это минимальное время удержания данных в заблокированном состоянии. Своими скоростными качествами оптимистическая блокировка обязана тому факту, что большую часть времени инструкции транзакции работают с данными, размещенными в локальном буфере. Как следствие операции с копией данных никоим образом не влияют на остальных пользователей БД. Необходимость в блокировке данных возникает только на заключительной стадии работы транзакции – во время переноса обработанных данных из кеша в БД.

Многоверсионная архитектура

В качестве модели, отвечающей за управление параллельно выполняющихся транзакций, разработчики СУБД InterBase (и однотипной системы FireBird) выбрали многоверсионную архитектуру (Multi-Generational Architecture, MGA), в целом соответствующую оптимистическому направлению [35]. Термин «многоверсионность» означает то, что у одной и той же записи таблицы одновременно может существовать несколько версий – по числу транзакций, пытающихся изменить строку таблицы. Работает это примерно так.

- Допустим, что к некоторой строке R с целью ее чтения обращается транзакция T_N с идентификатором N . Так как чтение не может

разрушить данные, то планировщик запросов просто отмечает данную строку признаком того, что она читается транзакцией T_N . Однако если транзакция намерена изменить кортеж, то вместо оригинальной строки R ей предоставляется ее копия – иначе говоря, версия R_N .

- Если во время работы транзакции T_N запустится транзакция T_{N+1} и она также захочет внести в запись R изменения, то немедленно создается очередная версия строки R_{N+1} , в которой и найдут свое отражение все действия T_{N+1} . Новая версия строки R_{N+1} размещается в памяти рядом с предыдущей R_N и помечается как строка, принадлежащая транзакции T_{N+1} .

Таким образом, при попытке изменить данные каждая из транзакций получает в свое распоряжение не оригинальную строку таблицы, а лишь ее копию – версию, с которой можно делать все, что угодно, не боясь вступить в конфликт с другими транзакциями. Если транзакция завершается успешно, то во время ее фиксации исходная (оригинальная) строка таблицы отмечается как устаревшая, и на ее место встает строка, которая за секунду до этого была лишь версией. Если же транзакция откатывается, то версия так и остается лишь версией, а оригинальная строка продолжает свой путь в прежнем качестве.

Для того чтобы еще лучше уяснить саму идею MGA, акцентируем внимание читателя на ряде особенностей многоверсионной архитектуры.

Во-первых, одновременно может сосуществовать целый пакет версий одной и той же строки R . Благодаря тому что все версии записи отмечены идентификаторами соответствующих транзакций, легко определяется видимость этих версий, например транзакции T_N будут доступны все версии записи с номером, равным и меньшим N . Это замечание особенно важно для определения уровня изоляции транзакций.

Во-вторых, несмотря на то что в MGA допускается совместное существование нескольких вариантов строк, среди этих вариантов только одна строка может быть отмечена как подтвержденная (зафиксированная).

В-третьих, вне зависимости от числа транзакций, «атаковавших» одну и ту же строку, каждая из транзакций, с некоторой долей оптимизма, полагает, что именно принадлежащая ее контексту версия записи будет зафиксирована в БД. Эта надежда теплится до тех пор, пока новая версия строки не начинает пересылаться в БД. Если во время пересылки возникает конфликт, то инициировавший транзакцию-неудачницу клиент получает сообщение об исключении, и эта транзакция откаты-

вается в исходное состояние. При удачном стечении обстоятельств все инициированные транзакцией изменения фиксируются, и версия становится оригиналом, а старый оригинал помечается как устаревший.

Дабы исключить переполнение БД многочисленными устаревшими версиями данных, в InterBase предусмотрен остроумный механизм очистки от ненужных версий. Задача борьбы с мусором возлагается не на транзакции, наплодившие новые версии данных, а на ни в чем не повинную транзакцию, стартовавшую намного позднее и обратившуюся к этим же данным. Прежде чем приступить к выполнению своих непосредственных обязанностей, новая транзакция осуществляет анализ версий строк, к которым она обратилась. И если оказывается, что хранящиеся в памяти версии были созданы уже прекратившими свою работу транзакциями, то они отмечаются как мусорные и удаляются физически.

Детализация уровня блокировок

В наших примерах мы опирались на модель, когда система способна блокировать отдельную строку таблицы, однако это не единственно возможный подход. У современных СУБД имеются и альтернативные подходы к **детализации блокировок** (lock granularity). Блокировке могут быть подвергнуты:

- 1) вся база данных;
- 2) отдельная таблица БД;
- 3) отдельная страница данных с частью таблицы;
- 4) отдельная строка в таблице;
- 5) отдельное поле в строке.

Степень детализации блокировки оказывает существенное влияние на производительность СУБД и на возможности работы протокола управления параллельными транзакциями, причем производительность и параллельность состоят в явной конфронтации друг с другом. Судите сами, вряд ли можно назвать многопользовательской СУБД, у которой блокируется вся БД. С другой стороны, при блокировке на уровне отдельных полей системе приходится обрабатывать огромный пласт служебной информации, что замедляет ее работу, но действительно облегчает параллельную работу транзакций.

Обычно за наложение блокировок отвечает система, создаваемые ею блокировки называются внутренними (implicit locks). Многие СУБД позволяют вмешиваться в процесс и программисту, тогда ему разреша-

ется вручную накладывать свои собственные блокировки на элементы данных, такие блокировки называются явными (explicit locks).

Требования стандарта SQL

Обсудим требования современного стандарта SQL к независимому выполнению параллельных транзакций. Для этого введен специальный термин – **уровни изоляции SQL-транзакций** (Isolation levels of SQL-transactions). Стандарт рекомендует разработчикам СУБД обеспечивать четыре уровня изоляции, упорядочим их по степени надежности:

- READ UNCOMMITTED (незафиксированное, или, как его иногда называют, «грязное» чтение) – наименее защищенный уровень изоляции, при котором транзакции способны читать незафиксированные изменения, сделанные другими транзакциями (табл. 16.1);
- READ COMMITTED (неповторяемое чтение) – исключается «грязное» чтение, транзакция увидит только изменения, зафиксированные другими транзакциями;
- REPEATABLE READ (повторяемое чтение) – накладывает блокировки на обрабатываемые транзакцией строки и не допускает их изменения другими транзакциями. В результате транзакция видит только те строки, которые были зафиксированы на момент ее запуска. Основной недостаток повторяемого чтения – высокая вероятность появления строк-фантомов;
- SERIALIZABLE (сериализуемость) – самый надежный уровень изоляции, полностью исключающий взаимное влияние транзакций.

Таблица 16.1. Влияние уровня изоляции на проблемы параллельного доступа

Уровень изоляции	Проблемы параллельного доступа			
	Потерянные обновления	Грязное чтение	Неповторяемое чтение	Фантомные строки
READ UNCOMMITTED	Исключено	Допускается	Допускается	Допускается
READ COMMITTED	Исключено	Исключено	Допускается	Допускается
REPEATABLE READ	Исключено	Исключено	Исключено	Допускается
SERIALIZABLE	Исключено	Исключено	Исключено	Исключено

Казалось бы, раз существует высший уровень изоляции транзакций, то всем транзакциям стоит всегда работать именно с этой степенью изоляции. Однако это было бы ошибочным решением. Степень изоляции, соответствующая уровню `SERIALIZABLE` в большинстве СУБД, так надежно защищает транзакцию от стороннего влияния, что практически приостанавливает параллельную обработку данных, заставляя остальные транзакции простаивать в очереди. Поэтому в современных СУБД предусмотрена возможность гибкой настройки уровня изоляции для транзакций.

Внимание!

Основное правило совместного выполнения транзакций заключается в том, что в ходе обращения к БД транзакция должна видеть только согласованные данные, доступ к промежуточным или используемым другими данным должен быть заблокирован!

Явное управление транзакцией

По умолчанию сервер баз данных берет на себя полную ответственность за управление транзакций. Как только пользователь отправляет СУБД инструкцию `SQL`, сервер самостоятельно стартует транзакцию, осуществляет фиксацию изменений, если в ходе выполнения транзакции не произошло исключительных ситуаций, или осуществляет автоматический откат операций, осуществленных в транзакции при наличии в них хотя бы одной ошибки. Такое поведение сервера называют **неявным управлением транзакциями**.

Вместе с тем в `SQL` предусматривается инструментальный набор по **явному управлению транзакциями**. Он складывается из трех инструкций:

- 1) `START TRANSACTION` (или `BEGIN`) – используется для запуска новой транзакции;
- 2) `COMMIT` – фиксация изменений, осуществленных транзакцией;
- 3) `ROLLBACK` – откат транзакции в исходное состояние.

В простейшем случае программисту можно не указывать характеристики транзакции, ограничившись `START TRANSACTION` или `BEGIN [WORK]`, в этом случае будет запущена транзакция с параметрами по умолчанию.

После явного старта транзакции в адрес СУБД передается набор инструкций на языке `SQL`, на рис. 16.1 они представлены как заранее опре-

деленный блок команд. Это может быть одна или несколько хранимых процедур, функции или динамически сгенерированные инструкции SQL. Корректность выполнения всего блока команд обязательно проверяется системой, как минимум на предмет поддержки целостности и непротиворечивости данных. Кроме того, программист также имеет возможность внедрить в блок команд дополнительные управляющие конструкции, в которых он реализует еще один уровень проверки.

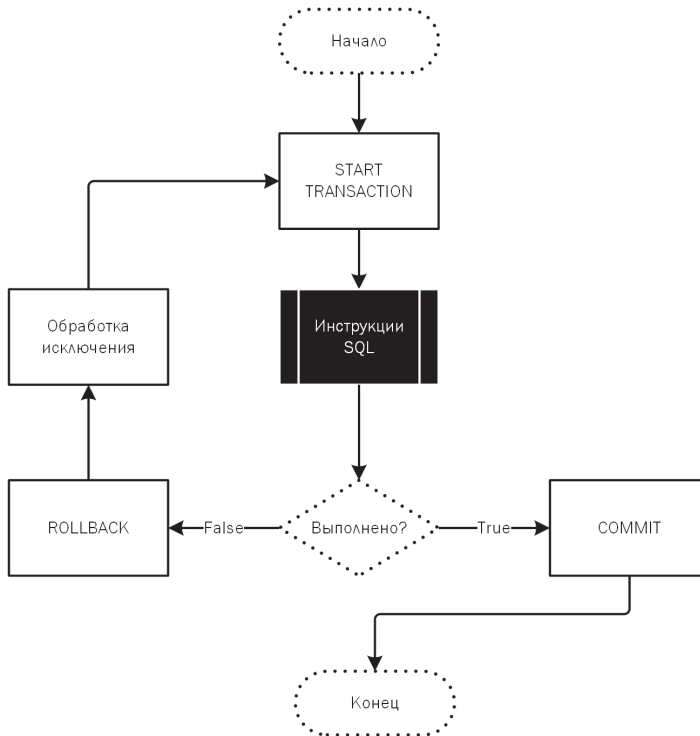


Рис. 16.1. Блок-схема управления транзакцией

В различных диалектах SQL код конструкций варьируется, так, в MySQL для подготовки транзакции надо воспользоваться примерно таким синтаксисом:

```

SET [GLOBAL | SESSION] TRANSACTION
[ISOLATION LEVEL уровень_изоляции]
[READ WRITE | READ ONLY]
  
```

По умолчанию система полагает, что вы настраиваете уровень изоляции транзакций для текущей сессии (необязательное ключевое слово

SESSION). Если это не так и вы намерены определять уровень изоляции для всех новых соединений, примените ключевое слово GLOBAL.

В состоянии по умолчанию сервер MySQL настроен работать с транзакциями с уровнем изоляции REPEATABLE READ (повторяемое чтение), возвратившись к табл. 16.1, вы увидите, что это наиболее оптимальный уровень изоляции с точки зрения стандарта SQL. Если вы намерены явно определить другой уровень, то в вашем распоряжении имеется четыре уже знакомые константы:

READ UNCOMMITTED | READ COMMITTED | REPEATABLE READ | SERIALIZABLE

Например, для запуска транзакции с заданным уровнем изоляции воспользуйтесь примером из листинга 16.1.

Листинг 16.1. Запуск транзакции с указанием уровня изоляции в MySQL

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

Специалисты советуют не злоупотреблять уровнем SERIALIZABLE, обеспечивающим наивысшую степень изоляции транзакций, так как обратной стороной медали является снижение производительности сервера при обработке параллельных транзакций.

Если корректность выполнения всего блока SQL не вызывала никакого сомнения, то мы должны подать команду

```
COMMIT [WORK] [AND [NO] CHAIN] [[NO] RELEASE]
```

Инструкция явным образом завершает транзакцию и фиксирует все изменения в БД.

Выражение AND CHAIN указывает системе, что с очередной транзакцией следует работать так, как будто она является следующим звеном предыдущей. В результате вторая транзакция станет использовать общие настройки первой транзакции (например, уровень изоляции). Инструкция COMMIT AND NO CHAIN (или просто COMMIT) укажет системе использовать настройки по умолчанию.

Ключевое слово RELEASE требует, чтобы после завершения транзакции клиент завершил текущую сессию и отключился от сервера. По умолчанию действует режим NO RELEASE – сессия не прерывается.

Если в ходе выполнения блока команд транзакции выявляются нарушения, то следует воспользоваться инструкцией отката БД к исходному состоянию.

```
ROLLBACK [WORK] [AND [NO] CHAIN] [[NO] RELEASE]
```

Получив команду на откат, система отказывается от переноса данных в таблицы, таким образом, БД остается в том же состоянии, что в момент запуска этой транзакции.

Внимание!

В составе транзакции категорически запрещается применять операторы, которые прямо или косвенно могут изменить структуру БД (таблиц, индексов, хранимых процедур, просмотров, триггеров и т. п.) или повлиять на права пользователей. Кроме того, внутри транзакции не стоит выполнять другие транзакции.

Точки сохранения

В ситуации, когда транзакция продолжительна и включает большое количество операций с данными, ее можно разбить на несколько логических частей за счет **точек сохранения** (savepoints). Точка сохранения выступает своеобразным маркером, говорящим, что предшествующая ей группа операций была осуществлена успешно. Если при дальнейшем выполнении транзакции произойдет исключение, то необходимость отката к ее самому первому оператору отпадает, вместо этого достаточно возвратиться к последней удачной точке сохранения и попытаться вновь возобновить транзакцию с этого места.

Замечание

Если вы осуществите контролируемый откат транзакции к какой-то из точек сохранения (допустим, с порядковым номером N), то это действие приведет к автоматической очистке всех последующих точек сохранения ($N + 1$, $N + 2$ и т. д.).

Резюме

Транзакция – это очень сильное оружие, неумелое применение которого может существенно снизить производительность БД. Чтобы этого не произошло, при проектировании транзакций следует соблюдать ряд правил, вот основные из них:

- пишите быстрые транзакции, такие транзакции должны затрагивать минимально возможный объем данных, ведь чем больше транзакция заблокирует строк – тем больше вероятность столкновения с другими транзакциями;
- не применяйте в транзакциях инструкции DDL, создающие, модифицирующие и удаляющие объекты БД, хотя в некоторых СУБД

транзакция даже не станет возражать против обращения к операторам CREATE, ALTER, DROP и т. п., но она или не обеспечит возможность их отката, или будет завершена неявным образом;

- остерегайтесь вложенных транзакций, в особенности если не можете предсказать, как поведет себя внешняя транзакция при откате вложенных;
- там, где это возможно, не применяйте высокие уровни изоляции транзакций.

Соблюдая перечисленные правила, вы превратите транзакции в своих самых надежных помощников.

Вопросы для самопроверки

1. Что такое транзакция?
2. Какие требования предъявляются к транзакциям?
3. Для чего ведется журнал транзакций?
4. В каких состояниях может находиться транзакция?
5. В чем заключается принципиальное отличие между оптимистическим и пессимистическим подходами управления транзакциями?
6. Как работает метод блокировок?
7. Как можно выйти из ситуации взаимной блокировки транзакций?
8. Какие фазы содержит метод двухфазной блокировки?
9. Разъясните принцип многоверсионного (MGA) подхода к управлению транзакциями.
10. Какие уровни детализации транзакций вам известны?
11. Что такое «явное» и «неявное» управление транзакцией?
12. Какие требования предъявляются стандартом SQL к изоляции транзакций?

Глава 17

Определение прав пользователей

В доступе к хранящейся в БД информации могут быть заинтересованы не только добросовестные пользователи. К сожалению, современный цифровой мир наполнен множеством лиц, чей интерес к конфиденциальной информации продиктован далеко не самыми благими намерениями. Для противодействия несанкционированному доступу в большинстве современных СУБД реализована многоуровневая система обеспечения безопасности, включающая три процедуры:

- **идентификацию** – сущность процедуры состоит в назначении пользователю (процессу) уникального имени;
- **аутентификацию** – это процедура проверки подлинности пользователя, представившего свой идентификатор. Обычно пользователь подтверждает то, что он является именно тем, за кого он себя выдает, путем ввода в систему уникальной (неизвестной другим) информации о себе. Наиболее распространенный способ подтверждения – ввод символьного пароля;
- если пользователь успешно прошел процедуру аутентификации, то сервер осуществляет его **авторизацию** – процедуру предоставления пользователю определенных ресурсов и прав на их использование.

Диапазон полномочий, которые может получить пользователь, достаточно широк – от ограниченного доступа к отдельному столбцу до всеобъемлющих прав относительно всей базы данных. Набор прав (привилегий) назначается администратором БД. Обычно пользователь вводится в группу с заранее предопределенными ролями (администратор данных, администратор БД, пользователь БД, гость).

Определение

Привилегия (privilege) – разрешение на использование определенной услуги управления данными для доступа к объекту данных, предоставляемое идентифицированному пользователю [6].

Все дальнейшее взаимодействие пользователя с объектами БД строго регламентируется в соответствии с назначенной ролью. Пользователи могут обладать разными правами на один и тот же объект, одни могут лишь просматривать данные, другие – только добавлять, третьи – осуществлять чтение, вставку и запись.

Идентификатор авторизации

Стандарт SQL не предусматривает никаких команд, предназначенных для создания учетной записи пользователя, с которой он будет входить в систему. Поэтому практически во всех СУБД для этих целей придуманы свои собственные методы. Обычно пользователя можно создать интерактивно, воспользовавшись консолью сервера или задействовав системные хранимые процедуры конкретного диалекта SQL. Например, в распоряжении администратора MySQL имеется инструкция `CREATE USER`, которая позволит создать учетную запись (листинг 17.1) для сервера, развернутого на локальном хосте.

Листинг 17.1. Создаем учетную запись с паролем для БД MySQL

```
CREATE USER 'db_user1'@'localhost' IDENTIFIED BY 'db_user1_password';
```

Схожее поведение InterBase и Firebird (листинг 17.2).

Листинг 17.2. Создаем учетную запись с паролем для БД InterBase

```
CREATE USER db_user2 SET PASSWORD 'db_user2_password';
```

Есть и другие решения. Разворачиваемая на базе операционной системы Microsoft Windows Server СУБД SQL Server в состоянии оперировать учетными записями пользователей, зарегистрированных на контроллере домена, таким образом, эта СУБД встраивается в целостную политику безопасности домена. Примерно так же поступают СУБД Ingres, Informix и DB2, они умеют извлекать учетную запись пользователя из недр операционной системы.

Пользователю, успешно прошедшему процедуры идентификации, аутентификации и авторизации, системой присваивается **идентификатор авторизации** (authorization identifier). И вновь стандарт никак не уточняет правила присвоения идентификатора. Поэтому разработчики СУБД решили, что весь процесс должен осуществляться в рамках старта новой SQL-сессии. И все это осуществляется таким образом, чтобы сразу после запуска сессия знала, какими полномочиями обладает владеец идентификатора авторизации, и не давала ему выйти за их рамки.

Замечание

В большинстве диалектов SQL реализована функция (реже переменная), благодаря которой можно узнать пользовательский идентификатор текущей сессии. Как правило, функция называется `SESSION_USER`.

Объекты защиты

Вполне естественно, что к БД предприятия может предоставляться доступ сотням или даже тысячам пользователей. Но согласитесь, что было бы неразумным всем им (от руководителя до технического служащего) предоставлять равноправный доступ к данным. Вместо этого гораздо логичнее выделить в распоряжение пользователей только те данные, которые им нужны по роду деятельности, и контролировать их доступ.

Стандарт SQL рекомендует разработчикам СУБД обеспечить контроль доступа к следующим объектам БД:

- таблицам;
- столбцам таблиц;
- представлениям;
- доменам;
- хранимым процедурам и функциям;
- пользовательским типам данных;
- наборам символов (речь идет о локализации БД);
- правилам сравнения и сортировки символов;
- трансляции символов.

Не станем уточнять, насколько полно эти рекомендации реализованы в той или иной СУБД, но в целом можно говорить, что большинство серверов БД как минимум поддерживает первую половину предложенного списка.

Управление наборами привилегий

Системы безопасности подавляющего большинства современных СУБД для определения прав пользователей используют заранее сформированные списки привилегий. Администратор системы обладает самым широким перечнем полномочий, разработчик БД – меньшим, полномочия обычного пользователя еще более сужены. В терминах SQL перечень заранее сформированных полномочий называют ролью (role).

Определение

Роль – это именованная совокупность привилегий, которые могут быть предоставлены пользователям или другим ролям.

Функциональная нагрузка, возлагаемая на роли СУБД, во многом схожа с задачами групп пользователей в Windows. Роль значительно упрощает процесс администрирования подсистемы авторизации СУБД – вновь появившийся пользователь просто наделяется правами из той или иной роли. Это гораздо удобнее, чем явное описание перечня полномочий для каждого пользователя в отдельности (рис. 17.1).

Прямое определение полномочий

Рольевой доступ

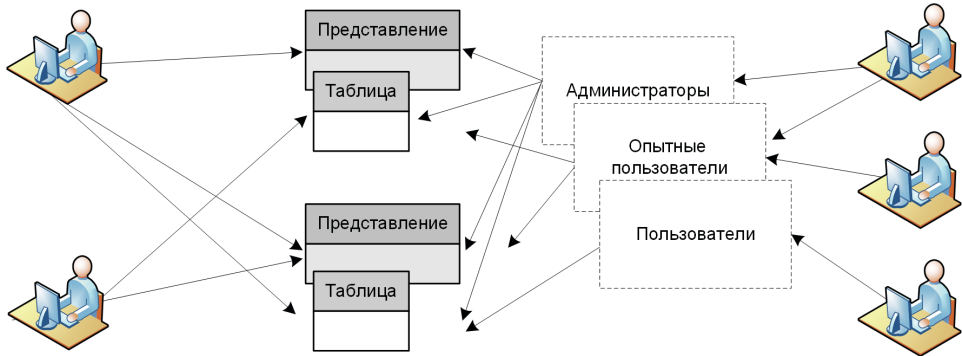


Рис. 17.1. Определение полномочий пользователя

Замечание

Каждой создаваемой в БД роли (так же, как и пользователю) назначается уникальный идентификатор.

Например, администратор БД может создать роль «Пользователи», которой будет позволено лишь просматривать некоторые таблицы,

а роль «Администраторы», наоборот, получит абсолютные права над БД. Впрочем, наличие механизма ролей не запрещает администратору определять права индивидуально для каждого из пользователей. Все зависит от конкретных условий обстановки.

Стандарт требует от создателей СУБД, чтобы в их диалектах SQL присутствовала инструкция `CREATE ROLE`, позволяющая создавать именованный набор полномочий, который позднее было бы можно передавать пользователям БД. Синтаксис такой инструкции выглядит следующим образом:

```
CREATE ROLE имя роли [WITH ADMIN {CURRENT_USER | CURRENT_ROLE}]
```

В простейшем случае команда на создание новой роли выглядит так, как предложено в листинге 17.3.

Листинг 17.3. Создание роли

```
CREATE ROLE DB_OWNER;
```

По умолчанию роль присваивается активному в данный момент пользователю, таким образом, продолжать инструкцию словами `WITH ADMIN CURRENT_USER` не обязательно. Если мы дополним инструкцию строкой `WITH ADMIN CURRENT_ROLE`, то роль назначается текущей роли.

Определение роли точно в соответствии с пожеланиями стандарта не реализовано ни в одной из упоминаемых в книге СУБД. Но очень похожие решения имеются в InterBase, Oracle (надо отметить, что здесь конструкция значительно усовершенствована) и в SQL Server. В PostgreSQL аналогом считается команда `CREATE GROUP` [45]. В ряде других систем эта задача реализуется косвенно, например через вызов системных процедур, либо не поддерживается вообще.

Удаление роли осуществляется командой `REVOKE ROLE`, редактирование – `ALTER ROLE`.

Предоставление привилегий

Предоставление прав на защищаемый объект санкционированному пользователю или роли осуществляется при посредничестве инструкции `GRANT`. Использование указанной команды гарантирует, что пользователь или роль имеет право выполнить определенные операции с объектом. Так как в СУБД полными правами на объект всегда владеет его создатель, то и право вызова `GRANT` обычно предоставляется владельцу объекта.

Команда GRANT вызывается при задании привилегий на объект базы данных или для определения заранее описанной роли. Синтаксис команды следующий:

```
GRANT {привилегия на объект [, ...] | имя роли [, ...]}
ON имя объекта
TO {получатель привилегии [, ...]}
[WITH GRANT OPTION] [WITH HIERARHY OPTION] [WITH ADMIN OPTION]
[FROM {CURRENT_USER | CURRENT_ROLE}]
```

Если мы планируем воспользоваться инструкцией для определения прав пользователя на объект, то сразу после ключевого слова GRANT следует указать возможные значения привилегий (табл. 17.1).

Таблица 17.1. Привилегии инструкции GRANT

Привилегия	Описание	Применимо к объектам
ALL PRIVILEGES	Назначить все привилегии	Ко всем объектам
SELECT INSERT UPDATE DELETE	Право на просмотр, вставку, редактирование и удаление данных в таблице (столбце)	Таблицы, столбцы, представления (только SELECT)
REFERENCES	Право управления ограничением внешнего ключа (FOREIGN KEY), право использовать столбцы в любом ограничении	Таблицы и столбцы
USAGE	Дает право использовать данный объект для определения другого объекта	Домены, пользовательские типы данных, наборы символов, порядки сравнения и сортировки, трансляции
UNDER	Право на создание подтипов или объектных таблиц	Структурные типы
TRIGGER	Право на создание триггера	Таблицы
EXECUTE	Запуск на выполнение	Хранимые процедуры и функции

Допустим, что мы намерены разрешить просмотр, вставку, редактирование и удаление данных из таблицы производителей для роли DB_USER. Тогда стоит написать представленную в листинге 17.4 строку кода.

Листинг 17.4. Предоставление полномочий роли DB_USER

```
GRANT ALL ON TABLE VENDORS TO DB_USER;
```

Привилегия может распространяться и на отдельный столбец или несколько столбцов таблицы, в таком случае имена столбцов перечисляются в круглых скобках сразу за именем таблицы (листинг 17.5).

Листинг 17.5. Полномочия на модификацию столбца для роли DB_USER

```
GRANT UPDATE
ON SUPPLIERS (SUPPLIER)
TO USER;
```

Так как предоставление привилегий для работы с таблицей является наиболее распространенным вариантом обращения к команде GRANT, то нам достаточно только указать имя таблицы. Если же следует обратиться к объекту другого типа, то стандарт требует уточнить тип объекта (табл. 17.2).

Таблица 17.2. Тип объекта инструкции GRANT

Объект	Описание
[TABLE] имя объекта	Таблица
DOMAIN имя объекта	Домен
COLLATION имя объекта	Сравнение
CHARACTER SET имя объекта	Набор данных
TRANSLATION имя объекта	Трансляция
SPECIFIC ROUTINE имя объекта	Подпрограмма

Строки кода из листинга 17.6 дают право на запуск процедуры PR_DELIVERYNOTE_DELETE для пользователей, входящих в роли DB_USER и DB_ADMIN.

Листинг 17.6. Предоставление полномочий роли DB_USER на запуск процедуры

```
GRANT EXECUTE
ON PROCEDURE PR_DELIVERYNOTE_DELETE
TO DB_USER, DB_ADMIN;
```

Для повышения читабельности кода в инструкцию SQL можно добавлять предложение `FROM CURRENT_USER` или `FROM CURRENT_ROLE`. В первом случае мы говорим, что привилегия была предоставлена от имени текущего пользователя, а во втором – от имени текущей роли. Если вы по каким-то причинам не стали завершать инструкцию `GRANT` этим предложением, то система станет полагать, что привилегия предоставлена от лица пользователя.

В SQL принято, что полноправным владельцем объекта всегда выступает его создатель. Что делать, если вы намерены передать свои полномочия другому лицу? Оператор `GRANT` позволяет наследовать привилегии одного пользователя другим. Право передачи привилегий определяет предложение `WITH GRANT OPTION` (листинг 17.7).

Листинг 17.7. Передача полномочий пользователю

```
GRANT USAGE ON DOMAIN MY_DOMAIN TO USER1 WITH GRANT OPTION;
```

Второй вариант опций `WITH HIERARCHY OPTION` позволит осуществлять операции выбора данных (инструкция `SELECT`) не только над объектом, но и над всеми подчиненными ему объектами.

Привилегии могут присваиваться как пользователям, так и ролям (листинг 17.8).

Листинг 17.8. Передача полномочий роли

```
GRANT ALL PRIVILEGES ON SUPPLIERS, CONTRACTS TO ADMINDB_ROLE;
```

Допускается передавать привилегии от одной роли к другой (листинг 17.9).

Листинг 17.9. Передача полномочий от одной роли другой роли

```
GRANT ADMINDB_ROLE TO USERDB_ROLE WITH ADMIN OPTION;
```

Обратите внимание, что при присвоении ролей нам потребовалась помощь опции `WITH ADMIN OPTION`, именно она позволяет присваивать роли.

Лишение привилегий

Наряду с предоставлением прав пользователю или роли стандартом SQL предусмотрена и обратная возможность – лишение привилегий. Синтаксис инструкции выглядит следующим образом:


```
REVOKE {[опции] | привилегия на объект [, ...] | имя роли [, ...]}  
ON объект  
FROM имя получателя [, ...]  
[GRANTED BY {CURRENT_USER | CURRENT_ROLE}]  
{CASCADE | RESTRICT}
```

Инструкция REVOKE также позволяет лишить пользователя или роль определенных прав доступа к объекту БД. Порядок применения команды очень схож с порядком работы с инструкцией GRANT. Если вы больше не хотите, чтобы пользователь BADUSER мог работать с таблицей заказов ORDERS, то можно выполнить инструкцию, предложенную в листинге 17.10.

Листинг 17.10. Лишение пользователя всех полномочий

```
REVOKE ALL PRIVILEGES  
ON ORDERS  
FROM BADUSER CASCADE;
```

Параметры CASCADE и RESTRICT уточняют, отменяется только определенная привилегия (RESTRICT) или же последует каскадное удаление всех связанных привилегий (CASCADE).

Внимание!

Существует вероятность, что одна и та же привилегия будет предоставлена пользователю от двух других независимых владельцев объекта. Допустим, что спустя некоторое время один из предоставивших привилегию передумает и решит отобрать данные права. При таком развитии событий SQL рекомендует, чтобы у попавшего в немилость пользователя по-прежнему сохранялись права на объект, до тех пор, пока инструкцию REVOKE не вызовет второй владелец объекта.

Если следует одновременно отобрать права у нескольких пользователей, то знайте, что инструкция REVOKE (это утверждение относится и к инструкции GRANT) допускает просто перечислить именно пользователей (или ролей) через запятую (листинг 17.11).

Листинг 17.11. Лишение полномочий нескольких пользователей

```
REVOKE DELETE  
ON WRITERS  
TO BADUSER1, BADUSER2 CASCADE;
```

Резюме

Основа защиты данных в современных БД основана на привилегиях, предоставляемых имеющим идентификаторы авторизации пользователям (ролям) на конкретные объекты БД. Линия обороны создается средствами языка SQL, для этих целей реализовано две инструкции:

- инструкция GRANT – используется для предоставления привилегий;
- инструкция REVOKE – применяется для отмены привилегий.

Однако возможностей этих инструкций вполне достаточно для построения мощной линии защиты от злоумышленников.

Никогда не повторяйте ошибок программистов, оставляющих решение задачи ограничения прав доступа пользователей к разрабатываемой БД на самый последний этап. Такой подход может привести к появлению брешей в системе защиты данных. Важно помнить, что формирование политики безопасности должно идти параллельно с проектированием самой БД.

Вопросы для самопроверки

1. Что понимается под выражением «многоуровневая система обеспечения конфиденциальности» в СУБД?
2. В каком случае пользователю присваивается идентификатор авторизации?
3. Какие объекты БД нуждаются в защите?
4. Дайте определение понятию «привилегия».
5. Какие объекты БД нуждаются в защите?
6. Что такое «роль», и как она участвует в процедуре наделения пользователя полномочиями?
7. Какими возможностями обладает инструкция GRANT?
8. Какими возможностями обладает инструкция REVOKE?

Глава 18

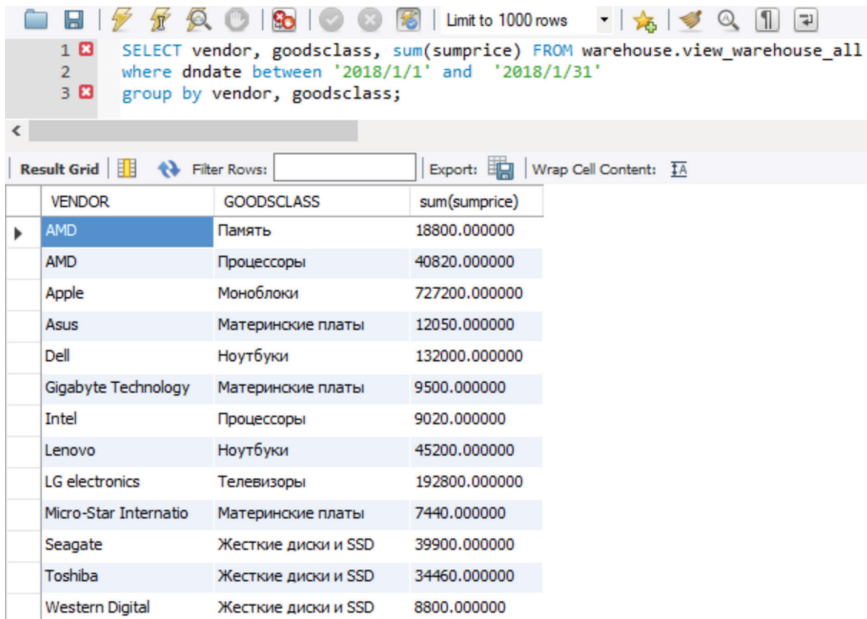
Интерактивная аналитическая обработка OLAP

Аббревиатура OLAP (англ. Online Analytical Processing) расшифровывается как **интерактивная аналитическая обработка**, термин введен Эдгаром Коддом. Под OLAP понимается технология обработки данных, заключающаяся в подготовке агрегированной (суммарной) информации на основе массивов данных, структурированных по многомерному принципу.

Для чего вдруг понадобился многомерный анализ? В используемой в качестве примера демонстрационной БД (см. *приложение 1*) учитываются товары, на протяжении определенного времени поступающие на склад от разных производителей. Вполне естественно, что руководству предприятия хочется иметь инструментарий для анализа этих данных. Однако классическое табличное представление совокупной информации вряд ли окажется столь удобным. Например, на рис. 18.1 предложен срез данных о поступлениях продукции на склад за один месяц, полученный обычным запросом группировки.

А что, если аналитику необходимо одновременно увидеть данные о поступлениях не за один месяц, а, например, за последние три года, сгруппированные по производителям и классификатору продукции? Написать три запроса и отобразить на экране три таблички? А что делать, если потребуется агрегировать еще один показатель? Сколько тогда придется одновременно вывести таблиц?

Как видите, привычное одномерное представление сведений далеко не вполне пригодно для решения задач, связанных с аналитической обработкой данных. Руководителю предприятия необходим абсолютно новый инструмент, предоставляющий ему расширенные возможности по анализу данных.



The screenshot shows a BI tool interface. At the top, there is a toolbar with various icons and a text box that says "Limit to 1000 rows". Below the toolbar, a SQL query is displayed in a text editor:

```
1 SELECT vendor, goodsclass, sum(sumprice) FROM warehouse.view_warehouse_all
2 where dndate between '2018/1/1' and '2018/1/31'
3 group by vendor, goodsclass;
```

Below the query editor, there is a "Result Grid" section. It includes a "Filter Rows:" input field, an "Export:" button, and a "Wrap Cell Content:" checkbox. The main part of the interface is a table with the following data:

VENDOR	GOODSCLASS	sum(sumprice)
AMD	Память	18800.000000
AMD	Процессоры	40820.000000
Apple	Моноблоки	727200.000000
Asus	Материнские платы	12050.000000
Dell	Ноутбуки	132000.000000
Gigabyte Technology	Материнские платы	9500.000000
Intel	Процессоры	9020.000000
Lenovo	Ноутбуки	45200.000000
LG electronics	Телевизоры	192800.000000
Micro-Star Internatio	Материнские платы	7440.000000
Seagate	Жесткие диски и SSD	39900.000000
Toshiba	Жесткие диски и SSD	34460.000000
Western Digital	Жесткие диски и SSD	8800.000000

Рис. 18.1. Табличное представление сведений о производителях – товарах

Требования к OLAP-инструментам

В 1993 году Э. Ф. Кодд сформулировал правила, которым, на его взгляд, должны удовлетворять OLAP-инструменты:

- 1) **многомерное концептуальное представление данных** (multi-dimensional conceptual view), соответствующее представлениям пользователя об организации;
- 2) **транспарентность** (transparency) – OLAP и вся сопутствующая ей архитектура должны быть прозрачны для пользователя;
- 3) **доступность** (availability) – OLAP обеспечивает доступ ко всем требуемым для анализа данным;
- 4) **неизменная производительность подготовки отчетов** (consistent reporting performance) – при проведении анализа не должна снижаться производительность как OLAP-инструментария, так и БД с оперативной информацией;
- 5) **архитектура клиент-сервер** (client-server architecture), позволяющая разделять функции по хранению, обработке и представлению данных между сервером и рабочими станциями;

- 6) **универсальность измерений** (generic dimensionality) – все измерения данных должны быть равноправными (эквивалентны по структуре и функциональным возможностям);
- 7) **динамическое управление разреженностью матриц** (dynamic sparse matrix handling). Инструмент OLAP должен уметь адаптироваться к конкретной аналитической модели и обеспечивать оптимальную обработку разреженных матриц;
- 8) **многопользовательская поддержка** (multi-user support), предполагающая одновременную работу с инструментарием OLAP нескольких пользователей;
- 9) **неограниченные перекрестные операции между размерностями** (unrestricted cross-dimensional operations). OLAP-система автоматически распознает иерархии размерностей и осуществляет перекрестные обобщающие вычисления внутри и среди размерностей;
- 10) **поддержка интуитивно понятного манипулирования данными** (intuitive data manipulation) – все операции с данными должны выполняться с помощью простейших действий пользователя;
- 11) **гибкость средств формирования отчетов** (flexible reporting);
- 12) **неограниченное число измерений и уровней обобщения** (unlimited dimensions and aggregation levels). Количество размерностей аналитической модели должно соответствовать требованиям бизнес-модели.

Позднее, в 1995 году, разработчикам OLAP-систем был рекомендован тест FASMI (Fast Analysis of Shared Multidimensional Information – быстрый анализ разделяемой многомерной информации), включающий следующие требования к программным продуктам, осуществляющим многомерный анализ:

- Fast – предоставление пользователю результатов анализа за минимальное время;
- Analysis – возможность осуществления любого логического и статистического анализа, характерного для данного приложения, и его сохранения в удобном для пользователя виде;
- Shared – разделяемый безопасный доступ к данным с поддержкой соответствующих механизмов блокировок и средств авторизованного доступа;

- Multidimensional – многомерное концептуальное представление данных, включая полную поддержку для иерархий и множественных иерархий;
- Information – возможность обращаться к требуемой информации независимо от ее объема и места хранения.

Заметим, что тест FASMI просто определяет концепцию OLAP, не конкретизируя, каким образом должна быть реализована требуемая OLAP-функциональность, эта область полностью находится в компетенции разработчиков ПО.

Существует три устоявшиеся модели хранения данных в OLAP-системах:

- 1) реляционная ROLAP (Relational OLAP). Основными составляющими архитектуры баз данных являются таблица фактов (fact table) и таблицы измерений (dimension tables);
- 2) многомерная MOLAP (Multidimensional OLAP) – является самым распространенным видом OLAP, здесь детальные и агрегированные данные содержатся в многомерной базе. Такой подход позволяет рассматривать данные как многомерный массив, благодаря чему скорость вычисления агрегатных значений одинакова для любого из измерений;
- 3) гибридная HOLAP (Hybrid OLAP) – использует реляционные таблицы для хранения базовых данных и многомерные таблицы для агрегатов.

Отметим, что в отношениях ROLAP не возбраняется частичный или даже полный отказ от соблюдения правил нормализации. Такое решение продиктовано тем, что быстрое объединение нормализованных таблиц (соответственно разбитых на десятки или даже сотни отношений) – весьма трудоемкий процесс, отнимающий много времени.

Хранилище данных

Для того чтобы получить возможность осуществлять многомерный анализ операционных данных, собранных в двухмерных реляционных таблицах (и/или в других источниках данных), данные должны быть обработаны соответствующим образом и отправлены в хранилища данных (data warehouse). Хранилище данных извлекает данные из БД и из других источников, тем самым образуя фонд специальным образом си-

стематизированных данных, удобных для осуществления оперативного анализа (рис. 18.2).

Замечание

Во время извлечения данных из источников обычно осуществляется их дополнительная обработка: соединение, классификация, фильтрация, агрегирование и т. д.

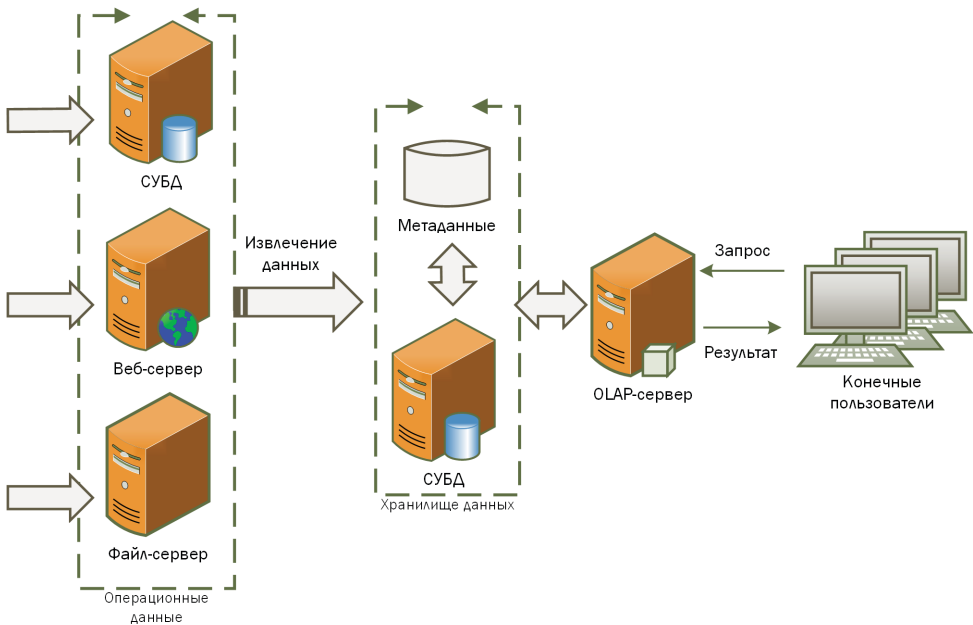


Рис. 18.2. Архитектура OLAP-системы

Автор концепции хранилищ данных Билл Инмон (Bill Inmon) утверждает, что хранилище должно обладать, по крайней мере, следующими признаками.

1. Предметная ориентированность – хранилище ориентировано на предмет (субъект) организации, а не на ее область деятельности. Другими словами, хранилище нацелено на работу с товаром (клиентом, поставщиком и т. п.), а не на учет поступления товара (формирование выписки счета клиента, возврат остатков поставщику).
2. Интегрированность – для формирования в интересах аналитика целостного представления разобщенных данных (которые

обычно поступают в систему из разных источников) необходимо согласовать исходные данные путем объединения их в единый источник.

3. Привязка по времени (нестационарность) – хранилище способно возвратить корректную аналитическую информацию только в ситуации, когда данные в хранилище привязаны к конкретному моменту или интервалу времени. Другими словами, по мере обновления данных должен осуществляться пересчет всех зависящих от времени показателей.
4. Неизменяемость – данные в хранилище не подлежат постоянно-му обновлению, а только регулярно пополняются из оперативных данных. При этом вновь поступившие данные не изменяют предыдущие, вместо этого осуществляется накопление данных.

Теоретически инструментарий OLAP может обойтись и без хранилища данных, собирая сведения непосредственно из БД с оперативной информацией, но специалисты так никогда не поступят по двум причинам.

Во-первых, хранилище данных централизует хранение всей информации организации, полученной из разных, зачастую гетерогенных источников. Таким образом OLAP-серверу гораздо удобнее обратиться к единому, чем к нескольким источникам данных.

Во-вторых, запросы OLAP требуют чрезвычайно высоких временных ресурсов на их выполнение. А если они осуществляются не к хранилищу данных, а к обычным БД с их системами обработки транзакций, иницируются транзакции, способные заблокировать БД на длительный период, что абсолютно неприемлемо.

OLAP-куб

Наиболее яркой иллюстрацией многомерного анализа выступает OLAP-куб. Куб содержит показатели, используемые предприятием для анализа и принятия управленческих решений.

Физически куб представляет собой многомерный (обычно разреженный) массив данных. На каждой оси куба может быть показано одно или несколько измерений (dimensions), которые представляют собой атрибуты из источника данных. Хранимые кубом значения обычно называют мерами (measures). В их качестве могут выступать прибыль, рентабельность, активы и т. п.

В настоящее время благодаря снижению стоимости оперативной памяти, магнитных и твердотельных накопителей расходы на хранение больших объемов данных вполне приемлемы. Вычислительные затраты на регулярный пересчет агрегированных величин также могут оказаться не критичными, если периодичность пересчета (находящаяся в прямой зависимости от изменений в хранилище данных) выставлена разумно.

Таким образом, если в хранилище данных имеется информация о поставках на склад с 2017 по 2019 год, то ось дат куба получит три измерения (2017, 2018 и 2019 год). Если поставки идут от трех компаний, то соответствующая ось получит 3 точки (рис. 18.2).

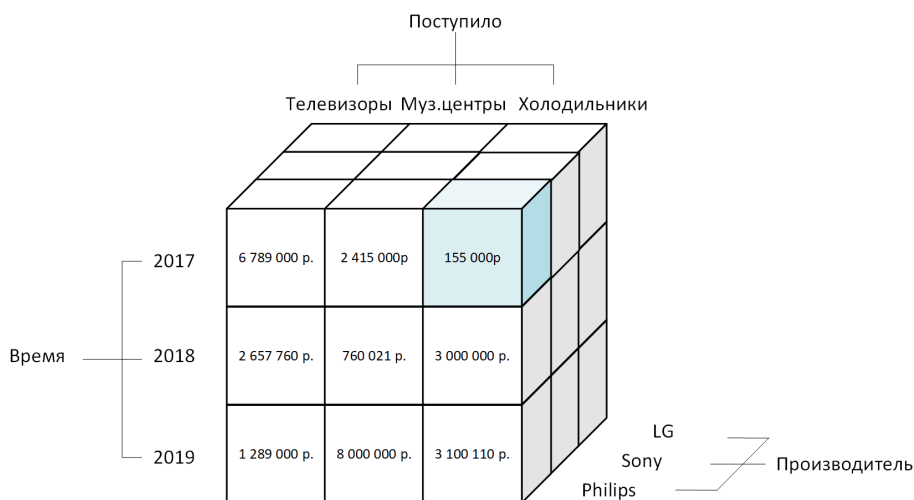


Рис. 18.3. OLAP-куб «Производитель – товар – время»

Инструментарий OLAP позволяет группировать данные в кубе. Например, «Время» может быть разложено по более мелким деталям (год, квартал, месяц и т. п.). Подобные действия с OLAP-кубом называют рассечением (dicing).

Кроме того, аналитику предоставляется возможность сужать выборку, например запрашивая только определенные ценовые рамки. Это действие называют срезом, или расслоением (slicing).

Язык многомерных выражений MDX

Специально для работы с многомерными данными в стенах корпорации Microsoft был разработан язык многомерных выражений (Multi-Dimensional eXpressions, MDX) и внедрен в Analysis Services 7.0 в 1998 году.

Язык превосходно разбирается с принятой в OLAP моделью устройства данных и обеспечивает навигацию по многомерному пространству.

Синтаксис MDX сильно напоминает SQL с той лишь разницей, что здесь вместо реляционной модели мы работаем с многомерными структурами со всеми присущими им особенностями (измерения, уровни иерархий, меры).

Для запроса SELECT в MDX используется следующая конструкция:

```
[WITH <выражение> [, <выражение> ...]]
SELECT [<ссылка_на_измерение>, [<ссылка_на_измерение>...]]
FROM [<обращение_к_кубу>]
[WHERE [<выражение_среза>]]
```

После ключевого слова SELECT следует ссылка на осевые измерения куба, представляющие те данные, которые вы желаете извлечь. При построении запроса разрешено указать до 128 осей, причем для первой пятерки осей заготовлены псевдонимы:

- 1) COLUMNS (столбцы);
- 2) ROWS (строки);
- 3) PAGES (страницы);
- 4) SECTIONS (разделы);
- 5) CHAPTERS (главы).

Последующие оси указываются с помощью слова Axis, за которым следует номер оси.

Предложение FROM определяет куб, данные из которого подлежат извлечению. Вновь просматривается некоторое сходство с SQL, где после FROM следовали имена таблиц, однако подчеркнем, что запрос MDX работает только с одним кубом (или подразделом куба).

Предложение WHERE назначает срез данных, извлекаемых из куба, ограничивая набор результатов запроса с помощью некоторого критерия.

Предложение WITH необязательно, оно понадобится нам, если аналитическая задача требует использования вычислений, которые должны быть сформулированы внутри какого-то запроса.

Язык MDX оснащен операторами, позволяющими осуществлять операции добавления, вычитания, умножения и деления, а также логическими операторами AND и OR.

В фигурные скобки {} заключается кортеж или несколько кортежей, чтобы сформировать набор MDX. В том случае, когда используется один кортеж, фигурные скобки не обязательны.

Листинг 18.1 демонстрирует простейший пример выборки данных из куба с помощью MDX.

Листинг 18.1. Пример выборки с помощью MDX

```
SELECT
{[Производитель].[Philips]} ON COLUMNS,
{[2017].[Январь]} ON ROWS
FROM [наш_куб]
WHERE [Поступило].[Холодильники]
```

Благодаря относительной простоте MDX с особенностями применения языка под силу разобраться не только программистам, но и обычным пользователям аналитических приложений.

Резюме

Появившийся в середине 1990-х годов механизм OLAP и сегодня является одним из самых популярных методов анализа многомерных данных. Информация, предоставляемая OLAP-системами аналитикам и руководству предприятий, выступает серьезным подспорьем при организации долгосрочного планирования и управления.

Для работы с данными, представленными в форме OLAP-куба, корпорацией Microsoft разработан специальный язык многомерных выражений MDX. Благодаря тому что у MDX сохранено внешнее сходство с запросами SQL, язык может быть легко освоен не только программистами, но и обычными пользователями.

Понимая важность и востребованность инструментария многомерного анализа, очень многие разработчики СУБД реализовали свои OLAP-системы, в их числе программные продукты известных производителей: SQL Server Analysis Services компании Microsoft, OLAP-сервер Hyperion Essbase компании Oracle, IBM Cognos TM1. Помимо коммерческих решений, существуют и OLAP-системы с открытым кодом. В частности, к таким относится Mondrian OLAP Server компании Pentaho. Сервер Mondrian написан на языке Java и является кроссплатформенным продуктом.

Вопросы для самопроверки

1. В каких случаях следует обращаться к OLAP-анализу?
2. Какие модели хранения данных применяются в OLAP-системах?

3. Почему для обеспечения работы OLAP обычно применяются хранилища данных?
4. Какие требования предъявляются к хранилищам данных?
5. Прокомментируйте требования теста FASMI к OLAP-системам.
6. Как устроен OLAP-куб?
7. Какой язык предназначен для работы с многомерными данными?

Глава 19

Расширяемый язык разметки XML

Расширяемый язык разметки (eXtensible Markup Language, XML) был разработан специально для создания структурированных документов – он как никто другой позволяет представлять и манипулировать элементами данных. Основное достоинство XML в том, что он дает возможность пользователям создавать свой собственный набор тегов. Благодаря такой возможности XML и назван **расширяемым**.

Замечание

Стандарт XML появился на свет в 1998 году, именно тогда он был опубликован консорциумом W3C (World Wide Web Consortium). Для получения последней информации о расширяемом языке разметки следует обращаться на сайт консорциума W3C, в частности по вопросам, связанным с XML, на страницу <http://www.w3.org/standards/xml/>.

Любой документ XML может рассматриваться как символьная строка, состоящая из символов разметки и данных. Благодаря тому что двоичные данные в XML отсутствуют, документ на основе расширяемого языка разметки может быть открыт и отредактирован в любом текстовом редакторе. Однако полноценная работа с документом XML невозможна без помощи специализированных программ-анализаторов (parsers).

Анализаторы несут ответственность за проверку правильности XML-документа и, только убедившись в этом, предоставляют всю необходимую информацию приложению, ориентированному на работу с данными XML. Зачастую XML-документ использует дополнительное описание, благодаря которому разработчик может наложить более строгие ограничения на хранимые в документе данные.

Корректность документа XML

При разборе документа анализатор оценивает его корректность по двум критериям [48]:

- 1) правильность оформления (well-formed) документа;
- 2) допустимость (valid) документа.

Проверка правильности относительно проста и включает лишь контроль синтаксиса и нескольких простых правил определения элементов XML, о которых мы поговорим немного позднее. Правильно оформленный документ станет доступным приложению сразу после осуществления проверки документа анализатором.

Проверка допустимости проводится значительно строже. Во-первых, допустимый документ должен быть правильным. Во-вторых, допустимый документ обязан соответствовать своему определению (мета-описанию), обычно представленному в формате DTD- или XML-схемы. Второй уровень проверки не является обязательным для всех XML-документов, более того, далеко не каждый документ обладает определением в виде DTD- или XML-схемы. Но с точки зрения приложений баз данных это желательная черта XML-документа.

Построение документа XML

Правила создания рядового документа XML столь просты, что для написания демонстрационного проекта в формате XML уйдет не более минуты (листинг 19.1).

Листинг 19.1. Простейший XML-документ

```
<?xml version="1.0" encoding="WINDOWS-1251"?>
<!--Это комментарий-->
<example>
  <line>Пример работы с языком XML</line>
</example>
```

Первая строка выступает визитной карточкой XML-документа. Здесь указано, что мы работаем с документом XML первой версии и документ предназначен для работы в кодировке Windows-1251, что автоматически разрешает использование символов кириллицы.

Еще одна характерная черта первой строки документа – применение в ней особенных символов разметки – `<? ... ?>`. Эти символы обрамляют

инструкции, которые подскажут анализатору XML порядок работы с содержащимися в документе данными.

Вторая строка содержит комментарий создателя документа. Текст комментариев заключается в символы `<!-- ... -->`, при разборе документа комментарий игнорируется анализатором.

Судя по примеру, едва ли не самыми популярными символами в нашем документе выступают левая (`<`) и правая (`>`) угловые скобки. Это зарезервированные символы, предназначенные для описания тегов, информирующих о начале и завершении элемента документа.

Элементы документа

Каждому открывающему тегу должен сопоставляться закрывающий тег (за исключением первой строки документа). В нашем вводном примере (листинг 19.1) существует связь между тегами `<example>` и `</example>`, `<line>` и `</line>`. Признаком того, что тег является закрывающим, служит наклонная черта `«/»`.

Совокупность открывающего, закрывающего тегов и данных внутри них

```
<line>данные</line>
```

называется элементом документа.

Внимание!

Язык XML чувствителен к регистру символов! В первую очередь про этот факт следует помнить при определении названий тегов, например в XML тег `<LINE>` не эквивалентен `<line>`.

Язык XML заслужил звание расширяемого языка разметки потому, что он позволяет нам определять новые теги. При остром желании в названиях тегов допустимо использовать символы национальных алфавитов.

Еще одним обязательным условием построения корректного XML-документа является наличие корневого элемента, в нашем случае это `<example>`. Внутри корневого элемента располагаются дочерние элементы, они, в свою очередь, могут быть родительскими для других элементов. Поэтому XML-документ как нельзя лучше подходит для хранения иерархических данных.

Настал черед свести воедино базовые правила определения элементов XML:

- каждому открывающему тегу должен соответствовать закрывающий тег;
- документ XML позволяет использовать вложенные друг в друга теги, при условии что теги не перекрываются (открывающему тегу должен соответствовать одноименный закрывающий тег);
- в документе XML должен быть всего один корневой элемент;
- имена тегов должны начинаться с буквы, имя не должно содержать символы пробела и двоеточия «:», кроме того, в именах запрещено применять аббревиатуры «XML», «xml», «Xml» и т. п.;
- имена чувствительны к регистру символов;
- в стандарте XML пробелы, символ табуляции и символ новой строки могут быть полноценными элементами данных.

Как бы это не показалось удивительным, но столь простые правила позволяют нам обмениваться данными с помощью XML.

Атрибуты

Предположим, что мы через интернет скачиваем прайс-лист одного из своих поставщиков. В нем (листинг 19.2) содержатся данные – систематизированные по разделам классификатора компьютерные товары, а внутри элементов классификатора `<classifier>..</classifier>` хранятся элементы с описанием товара `<goods>..</goods>`.

Листинг 19.2. XML-документ с атрибутами

```
<?xml version="1.0" encoding="WINDOWS-1251"?>
<pricelist>
<classifier>Моноблоки
  <goods VENDOR="APPLE" PRICE="100000.00">iMac MNE02RU/A</goods>
  <goods VENDOR="ACER" PRICE="30750.00">Aspire Z20-730</goods>
  <goods VENDOR="HP" PRICE="35550.00">HP 24-e041ur</goods>
</classifier>
<classifier>Ноутбуки
  <goods VENDOR="ACER" PRICE="46350.00">Aspire E5-576G-51UH</goods>
  <goods VENDOR="LENOVO" PRICE="21100.00">IdeaPad 320-15IAP</goods>
</classifier>
</pricelist>
```

Самое важное в новом примере – то, что открывающий тег описания компьютеров содержит пока не встречавшиеся нам блоки – атрибуты. Атрибут состоит из названия и значения. Порядок именования атри-

бутов ничем не отличается от правил задания имен для элементов документа. Значения атрибутов заключаются в двойные или одинарные кавычки, в нашем примере в них содержится дополнительная информация, описывающая ту или иную единицу товара.

Благодаря атрибутам в документах XML начинают просматриваться некоторые родовые черты классических баз данных, ведь атрибуты XML можно расценивать как столбцы реляционных таблиц. Схожесть возрастет еще больше после того, как мы обсудим далее возможности наложения ограничений на передаваемые через атрибуты данные (табл. 19.2).

Пространство имен

До тех пор, пока создаваемые нами XML-документы не находят массового применения и используются только в домашнем компьютере или в пределах небольшой компании, можно особенно не задумываться над выбором названий для элементов документа. Если мы полагаем, что для определения товара наиболее подходит имя элемента «goods», а для классификатора «classifier», то нам ничто не мешает использовать именно эти названия. Однако как только разработчик сталкивается с XML-документами, спроектированными для профессионального программного обеспечения, эксплуатируемого значительным числом пользователей, ситуация с подбором названий несколько изменяется. Ведь нельзя исключить вероятность, что используемые нами имена уже задействованы сторонними разработчиками в других целях.

На сегодняшний день существует значительное число стандартизированных или ставших стандартом де-факто названий элементов. Вы их встретите в масштабируемых векторных рисунках формата SVG (Scalable Vector Graphics), в словаре для описания математических выражений MathML, в языке управления химической информацией (Chemical Markup Language, CML) и во многих других производных от XML языках. Кто поручится за то, что применяемые вами названия не станут конфликтовать с именами других разработчиков? Выход из положения заключается в использовании в наших документах имен, гарантированно отличающихся от имен сторонних производителей. Конечно же, можно проявить немного выдумки и в наших документах выбирать такие названия элементов и атрибутов, которые вряд ли встретятся у других разработчиков. Для этого достаточно дополнить свои имена вроде глобальных универсальных идентификаторов GUID. Но в результате подобного подхода наш документ утратит одно из важнейших достоинств

XML – наглядность. Поэтому для снятия проблемы совпадения имен стандарт XML предусматривает более рациональное решение – введение пространства имен XML (XML Namespaces).

Пространство имен XML – это соглашение об именовании, в котором элементы XML документа, помимо собственно имени, снабжены особым префиксом, создаваемым по тем же правилам, по которым назначаются универсальные идентификаторы ресурса URI (Uniform Resource Identifier). Разработчики стандарта XML учли, что благодаря URI в интернете обеспечивается однозначность доменных имен, а эта особенность как нельзя лучше подходит и для обеспечения уникальности названий элементов в документах XML.

Переработаем прайс-лист поставщика таким образом, чтобы имена элементов стали соответствовать рекомендациям пространства имен XML и в результате приобрели право называться квалифицированными именами. В таких целях нам следует вставить в документ строку примерно следующего содержания:

```
<pl:pricelist xmlns:pl="http://www.myurl.ru/pricelist">
```

В начале строки мы объявляем префикс `pl` и связываем его с элементом `pricelist`. Заметьте, что имя префикса не стандартизировано и нам разрешается заменить его любой приемлемой аббревиатурой. Далее мы информируем анализатор о том, что имя соответствует пространству имен XML. Для этого мы применяем атрибут `xmlns:pl` (`xmlns` – сокращение от XML namespace). В заключение мы воспользуемся любым вымышленным URI, допустим <http://www.myurl.ru/pricelist>. Особо отметим, что для создания квалифицированного имени нет необходимости указывать реальную страничку в глобальной паутине, просто рекомендация по пространству имен XML требует пользоваться соглашением URI для префиксной части названий элементов.

Замечание

Имя элемента, построенного в соответствии с рекомендациями пространства имен XML, называется квалифицированным именем (qualified names).

После создания префикса мы получаем право использовать его при именовании всех дочерних по отношению к элементу `<pricelist>` элементов и атрибутов. С этой целью мы добавляем префикс к начальному и конечному тегам элементов и перед названиями атрибутов – так, как предложено в очередном примере прайс-листа (листинг 19.3).

Листинг 19.3. Документ XML с поддержкой уникальности имен

```
<?xml version="1.0" encoding="WINDOWS-1251"?>
<pl:pricelist xmlns:pl="http://www.myurl.ru/pricelist">
<pl:pricelist>
  <pl:classifier>Моноблоки
    <pl:goods VENDOR="APPLE" PRICE="100000.00">iMac MNE02RU/A</pl:goods>
    <pl:goods VENDOR="ACER" PRICE="30750.00">Aspire Z20-730</pl:goods>
    <!-- и т. д. -->
  </pl:classifier>
  <!-- и т. д. -->
</pl:pricelist>
```

При обработке подобного документа анализатор автоматически заменит префиксную часть имени на соответствующий префиксу универсальный идентификатор ресурса URI, что гарантирует уникальность имен элементов и атрибутов нашего документа.

Внимание!

Тот факт, что порядок именования префикса пространства имен XML опирается на правила URI, ни в коем случае не означает, что в глобальной сети вы обязательно обнаружите интернет-ресурс с указанным в XML-документе адресом.

Если в плане создания квалифицированных имен для элементов документа все понятно, то с именованием атрибутов в спецификации XML имеется некоторая неоднозначность. По сути, атрибуты принадлежат своим элементам, поэтому большинству анализаторов XML будет безразлично, вставляем мы перед именами атрибутов префиксы или обделяем атрибуты своим вниманием. В результате анализатор будет полагать, что строка

```
<pl:goods VENDOR="ACER" PRICE="30750.00">Aspire Z20-730</pl:goods>
```

идентична строке

```
<pl:goods pl:VENDOR="ACER" pl:PRICE="30750.00">Aspire Z20-730</pl:goods>
```

в которой префиксы расставлены перед каждым атрибутом. Поэтому порядок именования атрибутов отдается на откуп разработчику документа.

Подытожим рассказ о пространстве имен XML. Введение пространства имен в XML необходимо для решения двух задач:

- 1) для предоставления возможности программному обеспечению отличать одноименный элемент разных XML-документов;
- 2) для удобства группировки всех связанных элементов и атрибутов одного XML-приложения, что облегчает программному обеспечению их распознавание.

Определение типа документа DTD

Нельзя исключить вероятность того, что заложенная разработчиком смысловая нагрузка в элементы документа окажется не столь очевидной как для пользователя, так и для анализатора XML. Примеры неоднозначностей в толковании разметки документа можно приводить долго. Допустим, создавая элемент `<PHONENUM>`, мы с вами рассчитываем, что в нем будет храниться 11-значный номер телефона, но другой читатель предположит, что достаточно ограничиться сокращенным 7-значным городским номером абонента. Или иной пример: создавая элемент `<INN>`, разработчик надеется, что в нем обязательно окажется уникальный индивидуальный номер налогоплательщика, но пользователь может посчитать, что это поле вообще необязательно для заполнения... Для решения подобных проблем в базах данных разработан весьма эффективный аппарат поддержки целостности и непротиворечивости данных, а как обстоят дела в XML?

В ситуации неоднозначности толкования разметки стоит воспользоваться еще одним козырем XML – способностью подключения к документу дополнительного описания, называемого определением типа документа (Document Type Definition, DTD). DTD конкретизирует, что именно должно содержаться в элементах и атрибутах документа. Не нравится DTD, тогда воспользуйтесь альтернативным решением – XML-схемой.

В DTD и XML в первую очередь содержатся дополнительные пояснения о составе элементов и атрибутов документа, признаки обязательности для заполнения и описание применяемых типов данных.

Несмотря на то что описание документа XML на основе DTD несколько утрачивает популярность (сегодня более часто встречается описание в формате XML-схемы), DTD по-прежнему часто встречается во многих XML-документах, так как хорошо справляется с метаописанием простых структур.

Описание DTD позволяет решить двудединую задачу. Во-первых, документ XML, помимо данных, содержащий и специальное описание, позволяет анализатору провести более глубокую проверку корректнос-

ти содержимого файла. Во-вторых, благодаря DTD документ становится понятным специалисту, в нашем случае разработчику базы данных, поддерживающей XML.

Описание структуры документа XML с помощью DTD начинается сразу после обязательного заголовка документа и включает слово DOCTYPE. Признаком завершения секции описания выступает закрывающая квадратная скобка и закрывающий тег]>. Таким образом, простейшее описание будет выглядеть следующим образом:

```
<!DOCTYPE имя_элемента [ ]>
```

Внутри секции содержится некоторый набор тегов <! и >, в которые заключаются имена элементов структуры или списки атрибутов. Восклицательный знак указывает на то, что это элемент объявления, сразу за восклицательным знаком (без пробела) следует слово DOCTYPE и имя элемента структуры.

При желании определение DTD может быть включено в документ XML или находиться во внешнем файле. В первом случае несколько изменяется первая строка файла с данными.

```
<?xml version="1.0" encoding="WINDOWS-1251" standalone="yes"?>
```

Если документ содержит в себе описание DTD, то в строку включается признак автономности standalone со значением «yes». Если же определение документа располагается во внешнем файле, то объявление автономности отменяется, об этом анализатор уведомляется значением «no».

В простейшем случае описание DTD заносится непосредственно в документ XML, сразу после заголовка с версией (листинг 19.4).

Листинг 19.4. Определение DTD в составе документа XML

```
<?xml version="1.0" encoding="WINDOWS-1251" standalone="yes"?>
<!DOCTYPE pricelist [
    <!ELEMENT pricelist (#PCDATA)>
    <!ELEMENT goods (#PCDATA)>
    <!ATTLIST goods
        VENDOR CDATA      #IMPLIED
        PRICE CDATA       #REQUIRED>]>
<pricelist>
    <classifier>Моноблоки
    <goods VENDOR="APPLE" PRICE="100000.00">iMac MNE02RU/A</goods>
```

```
<!-- и т. д. -->
</classifier>
</pricelist>
```

В представленном примере мы создаем блок определения типа документа, информирующего нас о том, что:

- каждый информационный блок документа `pricelist` состоит из двух элементов (`classifier` и `goods`);
- оба элемента представляют собой разработанные символьные данные (Parsed Character Data, PCDATA), или, проще говоря, элементы хранят обычный текст;
- элемент `goods` содержит список атрибутов (тег `<!ATTLIST>`). Из них один атрибут обязателен для заполнения (`#REQUIRED`) и один необязательный (`#IMPLIED`).

Столкнувшись с объявлением DTD, анализатор сохраняет его в памяти. Затем, в процессе чтения блоков данных XML, анализатор проверяет их на соответствие требованиям DTD. Если выявляются отсутствующие (или, наоборот, лишние) элементы и атрибуты, или элемент данных находится не в ожидаемом месте, или содержимое элементов и атрибутов не соответствует объявлению DTD, то анализатор делает вывод о том, что документ некорректен.

В нашем примере элемент `goods` содержит объявления атрибутов. Список атрибутов начинается с тега `<!ATTLIST>`, далее следует имя элемента, которому принадлежит список, затем описание атрибутов и закрывающая угловая скобка. Ключевое достоинство атрибута в том, что тип хранимых в нем данных не ограничивается банальным `CDATA` (табл. 19.1). Конкретизируя тип атрибута, мы явным образом указываем анализатору порядок обработки символьных данных.

Таблица 19.1. Типы атрибутов DTD

Тип	Описание
CDATA	В качестве значения атрибута могут использоваться символьные данные
ID	Признак, что значение атрибута должно быть уникальным. Весьма полезное качество при работе с базами данных, так как позволяет однозначно идентифицировать элемент, которому принадлежит уникальный атрибут
IDREF	В атрибуте хранится ссылка на идентификатор (ID) элемента

Тип	Описание
IDREFS	В атрибуте находится разделенная пробелами последовательность значений IDREF
ENTITY	Ссылка на внешний объект (например, файл мультимедиа). При проверке корректности документа подобный объект пропускается анализатором
ENTITIES	Разделенные пробелами ссылки на внешние объекты
NMTOKEN	Строка символьных данных, которая будет рассматриваться анализатором как корректное значение
NMTOKENS	Строка, содержащая несколько разделенных пробелами NMTOKEN
Перечислимый атрибут	Атрибут, содержащий список допустимых значений. Например: <!ATTLIST video format (MPEG1 MPEG2 MPEG4) "MPEG4"> В скобках мы перечислили допустимые для атрибута значения видеформата, по умолчанию предлагается применять MPEG4

Описание DTD позволяет формировать достаточно сложные модели XML-данных. Допустим, нам необходимо построить иерархическую структуру описания состава системного блока персонального компьютера (листинг 19.5).

Листинг 19.5. Определение DTD системного блока персонального компьютера

```
<!DOCTYPE COMPUTERS [
  <!ELEMENT COMPUTER (CPU, HDD+, VIDEO, RAM, OTHER*)>
  <!ELEMENT CPU (#PCDATA)>
  <!ELEMENT HDD (#PCDATA)>
  <!ELEMENT VIDEO (#PCDATA)>
  <!ELEMENT RAM (#PCDATA)>
  <!ELEMENT OTHER (#PCDATA)>
]>
```

Строки примера указывают на то, что элемент «COMPUTER» должен содержать информацию о процессоре «CPU», жестком диске «HDD», видеокарте «VIDEO», памяти «RAM» и другом оборудовании «OTHER». С этой целью названия вложенных элементов перечисляются в круглых скобках после слова «COMPUTER». Подобное перечисление требует, чтобы при вводе XML-данных строго соблюдался порядок следования элементов, на первом месте процессор, на втором – винчестер и т. д. Если обозначенный порядок по каким-то причинам нарушится, то анализатор документа немедленно известит нас об ошибке в документе.

Обратите внимание на еще одну особенность описания DTD для примера с компьютером. Во второй строке листинга после названий элементов «HDD» и «OTHER» появились ранее не встречавшиеся нам символы «+» и «*». Благодаря им мы приобретаем дополнительные возможности (табл. 19.2) по введению ограничений на количество элементов данных и установке признака обязательности и необязательности элемента.

Таблица 19.2. Ограничение количества элементов

Определитель	Описание
?	Признак необязательного элемента, он может вообще отсутствовать в XML-данных либо появляться один раз
+	Элемент может появиться один или несколько раз
*	Элемент может не появляться или появляться несколько раз
Пустой определитель	Значение по умолчанию, элемент обязателен и может появляться только один раз

С помощью символов «+» и «*» мы сумели создать определение DTD для XML-документа с переменным числом элементов. Теперь при описании состава компьютера нам позволено подключать один или несколько жестких дисков и неограниченное число дополнительного оборудования «OTHER», при этом документ XML остается корректным.

Зачастую описание DTD хранят отдельно от документа XML. В этом случае содержание файла описания структуры документа для системного блока компьютера будет выглядеть следующим образом (листинг 19.6).

Листинг 19.6. Определение DTD в отдельном файле

```
<!ELEMENT COMPUTER (CPU, HDD+, VIDEO, RAM, OTHER*)>
<!ELEMENT CPU (#PCDATA)>
<!ELEMENT HDD (#PCDATA)>
<!ELEMENT VIDEO (#PCDATA)>
<!ELEMENT RAM (#PCDATA)>
<!ELEMENT OTHER (#PCDATA)>
```

Файл описания сохраняется с расширением «dtd», например «computers.dtd». Для того чтобы анализатор XML понял, что документ XML нуждается в подключении файла «computers.dtd», следует включить дополнительную строку с указанием имени файла описания. В простей-

шем случае, когда файлы описания и документ находятся в одном и том же каталоге локального компьютера, достаточно указать лишь имя файла (см. вторую строку листинга 19.7).

Листинг 19.7. Документ XML, основанный на определении computers.dtd

```
<?xml version="1.0" encoding="WINDOWS-1251" standalone="no"?>
<!DOCTYPE COMPUTERS SYSTEM "computers.dtd">
<COMPUTERS>
  <COMPUTER>
    <CPU>Intel Core i3 7100</CPU>
    <HDD>6TB</HDD>
    <VIDEO>GeForce GTX 1050</VIDEO>
    <RAM>DDR4 8Gb</RAM>
  </COMPUTER>
  <COMPUTER>
    <CPU>Intel Core i5 7200U</CPU>
    <HDD>128Gb(SSD)</HDD>
    <HDD>2Tb</HDD>
    <VIDEO>Intel HD Graphics 620</VIDEO>
    <RAM>DDR4 8Gb</RAM>
    <OTHER>Сканер CANON Canoscan LiDE 120</OTHER>
    <OTHER>Принтер CANON PIXMA iP2840</OTHER>
  </COMPUTER>
</COMPUTERS>
```

Файл описания разрешается располагать в отдельном каталоге на компьютере или даже в интернете. Например:

```
<!DOCTYPE computers SYSTEM "file:///c:/dtdcash/computers.dtd">
<!DOCTYPE computers SYSTEM "http://www.DMK.ru/examples/computers.dtd">
```

В первом случае мы уведомляем анализатор, что файл описания можно обнаружить в папке `c:/dtdcash/`, из второго примера анализатор поймет, что доступ к файлу осуществляется через интернет с помощью URI: [http:// www.mysite.ru/examples/computers.dtd](http://www.mysite.ru/examples/computers.dtd).

Замечание

Для более глубокого изучения DTD обратитесь в интернет по адресу <http://www.w3.org/TR/REC-xml#dt-doctype>.

XML Schemas

Описание DTD вполне справляется с определением структуры XML-документов небольшой степени сложности, поэтому оно до сих пор находит применение во многих проектах. Вместе с тем информационные технологии никогда не стоят на месте, и очень скоро после выхода стандарта XML консорциум W3C пришел к выводу, что настал черед разработки более высокоуровневого формата описания XML-документов, тем более что к тому моменту времени у консорциума имелся ряд интересных наработок. В результате в 2001 году W3C представил разработчикам новую хорошо продуманную технологию XML Schemas (схема XML), позволяющую создавать метаописание XML-документа с более высокой степенью детализации.

Технология XML Schemas освободилась от недостатков DTD и приобрела ряд преимуществ, вот только ключевые из них:

- поддержка стандартных типов данных;
- возможность создания пользовательских типов данных, включая сложные структуры данных;
- схема предоставляет возможность анализаторам проверять не только структуру, но и содержимое данных.

Еще одно преимущество XML Schemas – в том, что, в отличие от описания DTD, целиком и полностью построенного на SGML, XML-схема основана на XML. Благодаря опоре на доступный синтаксис определение структуры документа на базе XML-схемы начало постепенно вытеснять DTD. Самое лучшее тому подтверждение – небольшой пример. Вспомните наш вводный пример документа XML (листинг 19.1).

Как будет выглядеть соответствующая представленному на листинге 19.1 документу схема? Примерно следующим образом (листинг 19.8).

Листинг 19.8. Схема для документа «Hello, World!»

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="demotext" type="xs:string"/>
</xs:schema>
```

Первая строка схемы ничем не отличается от строк, применяемых при создании обычного документа XML. Вторая строка открывает описание самого главного элемента схемы <schema> с одновременным

определением пространства имен. Третья строка указывает на то, что в результирующем документе должен существовать единственный элемент «demotext» строкового типа. Четвертая строка завершает описание схемы, закрывая тег <schema>. На первый взгляд, все не так уж и сложно, не правда ли?

Замечание

Файл XML-схемы должен иметь расширение имени .xsd, это первые буквы слов XML Schema Definition – определение схемы XML.

Несмотря на свою простоту, наш пример схемы способен научить нас первым трем правилам построения XML-схем:

- корневым узлом схемы всегда должен выступать элемент <schema>;
- схема должна ссылаться на пространство имен <http://www.w3.org/2001/XMLSchema>;
- в минимальной нотации определение элементов схемы должно содержать имя и тип данных элемента.

Основная причина успеха описания документов на базе XML-схем не простота, а более серьезные возможности по определению данных в сравнении с DTD, что позволяет широко применять XML-документы в базах данных. На рис. 19.1 предложен результат преобразования рассмотренного ранее описания «computers.dtd» (листинг 19.6) к формату файла xsd. Здесь вы найдете значительно больше родовых черт XML-схем, чем в листинге 19.8. По аналогии со всеми рассмотренными ранее XML-документами схема начинается с указания версии XML и конкретизации используемой в ней кодировки символов. Вторая строка схемы открывается обязательным корневым элементом <schema>. Сразу за ним следует объявление пространства имен XML Schema и целевого пространства имен.

По умолчанию определение любой схемы всегда должно ссылаться на пространство имен <http://www.w3.org/2001/XMLSchema>. Это говорит о том, что элемент <schema> является частью пространства имен XML Schema. Обозначенный URI рекомендован стандартом, посему его следует включать во все проекты, при этом не забывая строго соблюдать регистр символов.

При формировании XML-схемы разработчик имеет возможность на основе стандартизированных типов данных описывать свои собственные глобальные типы данных, которые позднее смогут использоваться

для определения локальных элементов схем. Так как создаваемые нами пользовательские типы данных не принадлежат пространству имен XML Schema, мы вводим еще одно альтернативное пространство – целевое пространство имен. Для этого предназначен атрибут `targetNamespace`, в нашем примере он указывает на то, что данная схема составлена для целевого пространства имен <http://tempuri.org/computers>. В результате мы достигаем задуманного эффекта – наши объявления элементов и атрибутов становятся квалифицированными.

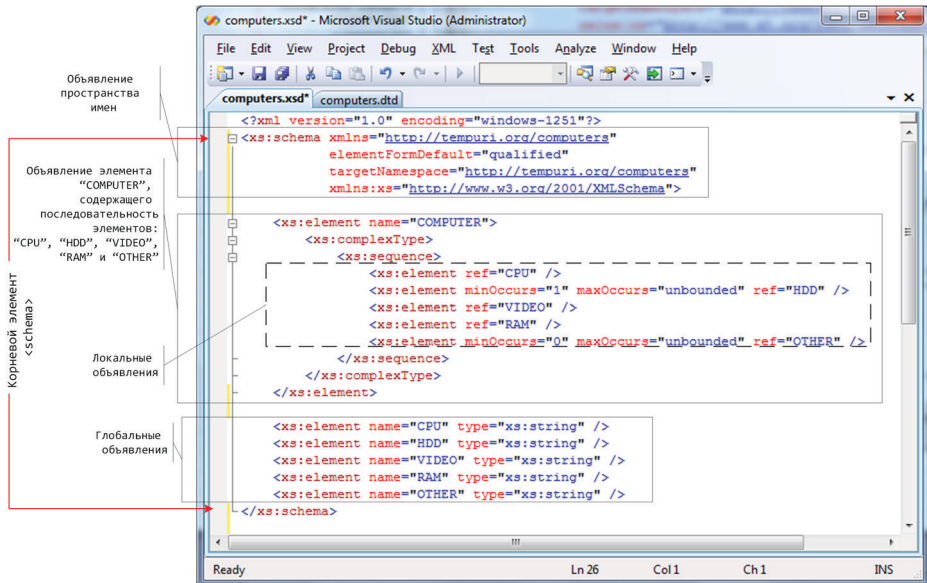


Рис. 19.1. Файл описания computers.xsd

В некоторых случаях от гарантированной уникальности имен допустимо и отказаться. Для этого в XML-объявлении предусмотрен еще один атрибут – `elementFormDefault`. Примененное в нашем примере значение `qualified` укажет анализатору, что все элементы документа квалифицированы. Но, выбрав обратное значение – `unqualified`, мы уведомляем анализатор, что какая-то часть элементов схемы может быть и неквалифицирована.

Если за включение (отключение) квалифицированных имен элементов отвечает атрибут `elementFormDefault`, то за аналогичное действие, но в этот раз с именами атрибутов, несет ответственность `attributeFormDefault`. Для атрибутов схемы требование по квалификации имен не столь строго, посему по умолчанию значение атрибута `attributeFormDefault` устанавливается в `unqualified`.

В качестве префикса пространства имен в соответствии с пожеланиями консорциума W3C следует применять символы `xs`, однако если префикс вам не по душе, то можете подобрать любой другой.

Элементы схемы

Корневому элементу `<schema>` принадлежит иерархия дочерних элементов схемы. Описание этих элементов начинается с тега `<element>`. Синтаксическая конструкция объявления элемента выглядит следующим образом:

```
<element
  name="имя элемента"
  type="тип данных"
  ref="ссылка на глобальное объявление типа элемента"
  form="квалифицирован (qualified) или не квалифицирован (unqualified)"
  minOccurs="минимальное количество элементов"
  maxOccurs="максимальное количество элементов или без ограничений (unbounded)"
  default="значение по умолчанию"
  fixed="фиксированное значение">
```

При определении элемента схемы разработчик обязан сделать две вещи: назначить имя элементу и описать ограничения на принимаемые элементом данные. В минимальной нотации достаточно задать имя и назначить тип данных элемента, например:

```
<element name="HDD" type="string"/>
```

Строки, подобные этой, вы обнаружите в области глобальных объявлений нашего примера с рис. 19.1. При определении типа данных допускается использовать не только стандартные, но и реализованные на их основе производные типы данных. Это очень полезное качество XML-схем позволяет вводить ограничения, сходные с применяемыми в базах данных доменными ограничениями.

Типы данных XML-схемы

Консорциум W3C стандартизировал типы данных, применяемых в XML-схемах. Согласно спецификации, эти типы данных подразделяются на два вида: встроенные первичные типы (*built-in primitive data types*) и вторичные типы (*derived data types*). Перечень первичных и вторичных типов данных вы обнаружите на рис. 19.2. Представленные типы в том или ином виде встречаются в большинстве языков программирования,

в частности в рассмотренном ранее структурированном языке запросов SQL, поэтому мы уклонимся от их углубленного изучения.

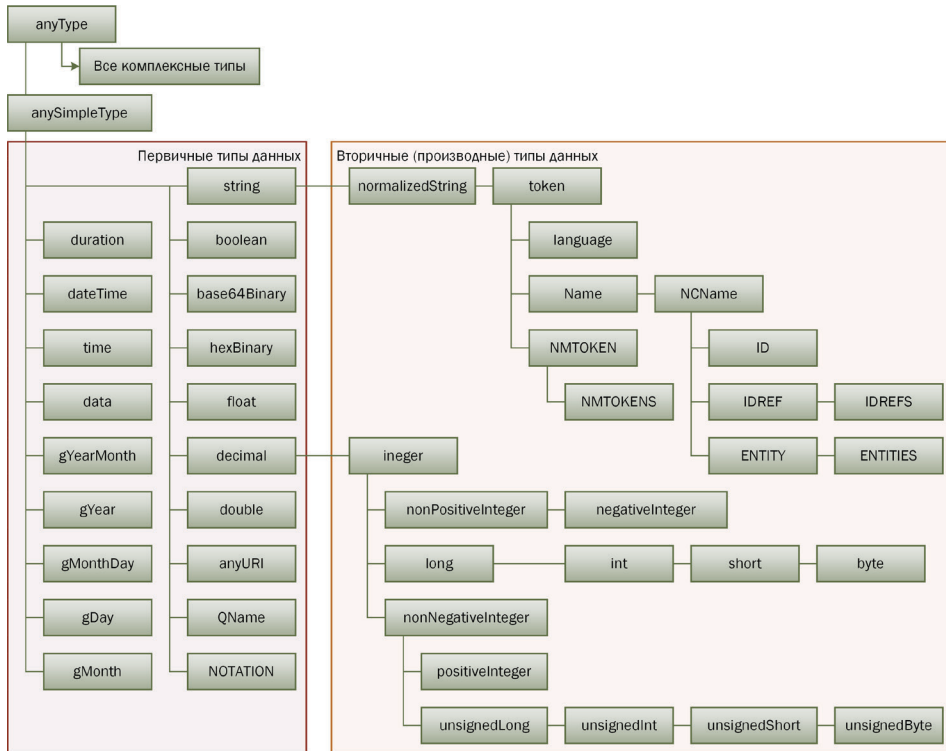


Рис. 19.2. Классификация типов данных XML

Еще одним важным замечанием, касающимся типов данных XML, станет то, что на данные того или иного типа могут накладываться дополнительные ограничения, например по допустимому диапазону значений или на количество символов, что для описания ограничений необходимо воспользоваться услугами тега `<restriction>` (табл. 19.3).

Главным помощником в создании производных типов данных станет тег `<simpleType>`. Несколько упрощенная синтаксическая конструкция создания производного типа представлена ниже:

```

<simpleType name = "имя"
  final = (list или union или restriction))
  {другие атрибуты, не принадлежащие пространству имен}...>
  
```

В рамках производных типов нам разрешается накладывать ограничения, создавать перечисления и даже объединения типов. Наши требо-

вания к подлежащим хранению данным задаются с помощью атрибута `final`, он способен принимать одно из трех значений:

- `<restriction>` – ограничение на значения;
- `<union>` – объединение типов;
- `<list>` – структура специализируется на хранении перечисления.

Ограничения на допустимые значения, передаваемые в поля документа XML, назначаются с помощью определения `<restriction>` и при активном посредничестве предусмотренных в стандарте 12 атрибутов (табл. 19.3).

Таблица 19.3. Атрибуты определения ограничений `<restriction>`

Фасет	Описание
<code>minExclusive</code>	Определяет минимальное значение диапазона, исключая указанное значение
<code>minInclusive</code>	Определяет минимальное значение диапазона, включая указанное значение
<code>maxExclusive</code>	Определяет максимальное значение диапазона, исключая указанное значение
<code>maxInclusive</code>	Определяет минимальное значение диапазона, исключая указанное значение
<code>totalDigits</code>	Назначает количество значащих цифр в цифровых типах данных
<code>fractionDigits</code>	Назначает количество знаков после запятой
<code>Length</code>	Указывает число элементов в списках или количество символов в текстовых типах данных
<code>minLength</code>	Минимальное число элементов в списках или в текстовых типах данных
<code>maxLength</code>	Максимальное число элементов в списках или в текстовых типах данных
<code>enumeration</code>	Позволяет определять допустимые значения в перечислениях
<code>whiteSpace</code>	Определяет реакцию на пробелы во вводимых значениях. Может принимать три значения: <code>preserve</code> – документ не реагирует на пробелы; <code>replace</code> – несколько повторяющихся пробелов сводятся в один пробел; <code>collapse</code> – несколько повторяющихся пробелов трансформируются в один пробел, и удаляются все пробелы перед значением и после него
<code>pattern</code>	Шаблоны и ограничения на текстовые типы

Атрибут `<restriction>` может находиться не только в структуре `<simpleType>`, но и в элементах `<simpleContent>` и `<complexContent>`.

Продemonстрируем порядок определения ограничений. Допустим, что нам необходимо создать тип данных, допускающий хранение целочисленных значений в диапазоне от 1 до 10 000, в этом случае можно воспользоваться примером, предложенным в листинге 19.9.

Листинг 19.9. Ограничение диапазона целых чисел

```
<xs:simpleType name="HDDSize">
  <xs:restriction base="xs:integer">
    <xs:minInclusive value="1"/>
    <xs:maxInclusive value="10000"/>
  </xs:restriction>
</xs:simpleType>
```

Определение опорного типа данных – обязательное условие при работе с ограничениями, нам надо лишь подобрать наиболее подходящий на эту роль (рис. 19.2). В приведенном выше примере мы остановили свой выбор на типе данных `integer`, об этом мы уведомляем анализатор, написав следующую строку: `<xs:restriction base="xs:integer">`. Затем мы устанавливаем требуемые ограничения, заполнив атрибуты `minInclusive` и `maxInclusive`.

Еще один пример (листинг 19.10) демонстрирует применение атрибута `length`. Названный атрибут определяет число символов, которые могут быть переданы в элемент или атрибут. Подобное ограничение может пригодиться при определении типа данных для хранения 11-значного телефонного номера.

Листинг 19.10. Тип данных для хранения телефонного номера

```
<xs:simpleType name="PhoneNum">
  <xs:restriction base="xs:string">
    <xs:length value="11" fixed="true"/>
  </xs:restriction>
</xs:simpleType>
```

Если количество символов в строке может варьироваться в заданных пределах, то вместо `length` применяем атрибуты `minLength` и `maxLength`.

Листинг 19.11 предлагает пример работы с ограничением `restriction` и демонстрирует возможность создания перечислений. Допустим, что

мы намерены сформировать некоторый перечень значений, в рамках которых пользователь будет назначать классификационный признак для товара.

Листинг 19.11. Перечень допустимых значений

```
<xs:simpleType name="goodsKind">
  <xs:restriction base="xs:string">
    <xs:enumeration value="Бытовая техника"/>
    <xs:enumeration value="Телевизоры"/>
    <xs:enumeration value="Музыкальные центры"/>
    <xs:enumeration value="Смартфоны"/>
    <xs:enumeration value="Компьютеры"/>
  </xs:restriction>
</xs:simpleType>
```

Представленный выше простой тип данных `goodsKind` позволяет элементу использовать только одно из перечисленных в нем значений. Но это не догма, небольшая помощь объявления `<list>`, и мы приобретем возможность передавать в элемент несколько значений (листинг 19.12).

Листинг 19.12. Получение данных из перечисления

```
<xs:simpleType name="goodsKindList">
  <xs:list itemType="xs:goodsKind">
</xs:simpleType>
```

Теперь элемент типа `goodsKindList` позволит принимать одновременно несколько разделенных пробелами значений из списка `goodsKind`.

Объявление `<union>` проводит комбинирование различных типов данных. Допустим, что мы хотим допустить хранение столь необходимого в базах данных неопределенного значения `NULL` в элементе, специализирующемся на обслуживании целых чисел. С этой целью мы создадим пользовательский тип данных `NULLString`, который способен содержать лишь один текстовый элемент `NULL` (листинг 19.13).

Листинг 19.13. Объявление пользовательского типа данных

```
<xs:simpleType name="NULLString">
  <xs:restriction base="xs:string">
    <xs:enumeration value="NULL">
```

```
</xs:restriction>
</xs:simpleType>
```

А теперь настал черед объявления `<union>`. В листинге 19.14 мы описываем еще один пользовательский тип `NULLorInteger` и с помощью `<union>` указываем, что он специализируется на хранении целых чисел или типа данных `NULLString`.

Листинг 19.14. Объявление комбинированного пользовательского типа данных

```
<xs:simpleType name="NULLorInteger">
  <xs:union memberTypes="integer xs:NULLString">
</xs:simpleType>
```

С этого момента мы имеем возможность объявлять элементы, способные содержать как целые числа, так и текстовое значение «NULL» (листинг 19.15).

Листинг 19.15. Объявление элемента заданного типа

```
<element name="intKey" type="xs:NULLorInteger"/>
```

Квалифицирование элемента

Атрибут `form` определяет порядок квалификации отдельного элемента. Если атрибут не задействован, то отношение текущего элемента к пространству имен зависит от настроек атрибута `elementFormDefault` (с которым мы познакомились при обсуждении заголовка XML-схемы). Если же мы намерены явным образом квалифицировать элемент, то присваиваем атрибуту `form` значение `qualified` или `unqualified`.

Ограничения на количество элементов

Напомним читателю, что для определения числа повторов элементов в документе XML с помощью определения DTD мы использовали символы «*», «+» и «?» (табл. 19.3). Подобный подход неудобен хотя бы по той причине, что из-за расплывчатых указаний DTD мы не в состоянии задать четкие ограничения на число элементов в результирующем документе.

В схеме XML предложено более элегантное решение, опирающееся на атрибуты `minOccurs` и `maxOccurs`. Первый из атрибутов определяет минимальное количество элементов, которые должны встретиться в

корректном документе XML, второй – максимальное число элементов. Например:

```
<xs:element name="HDD" minOccurs="1" maxOccurs="8"/>
```

Здесь говорится, что как минимум в системном блоке компьютера должен находиться один жесткий диск, а верхний предел не должен превышать 8. Если число дисков не ограничено, то вместо 8 в атрибут `maxOccurs` следует поместить значение `unbounded`.

Значение по умолчанию и фиксированное значение

Для определения значения по умолчанию предназначен атрибут `default`. Само значение передается после знака присвоения в двойных кавычках.

```
<xs:element name="CPU" type="xs:string" default="Intel Core i7-8709G"/>
```

С этого момента при встрече неопределенного элемента, отвечающего за описание процессора компьютера, анализатор вставит в пустое место значение по умолчанию.

Несколько реже требуется создавать элементы, значение которых никогда не изменяется. Для определения фиксированного значения в описание элемента следует включить атрибут `fixed`.

```
<xs:element name="DefaultLanguage" type="xs:string" fixed="Russian"/>
```

С этого момента любое отклонение от значения «Russian» при определении языка по умолчанию в содержимом XML-документа будет расцениваться как некорректное.

Создание сложных структур

При построении XML-схемы нам предоставлена возможность формирования сложных пользовательских структур на базе стандартизированных первичных и вторичных типов данных. Для определения структуры стандартом XML предусмотрен элемент `<complexType>`.

Объявление `<complexType>` можно сравнивать со структурой из C++. Комплексная структура специализируется на хранении множества разнотипных элементов и атрибутов из набора стандартизированных в XML типов данных (рис. 19.2). Если возможностей стандартных типов данных окажется недостаточно, то следует вспомнить о существовании элемента `<simpleType>`. Он позволяет накладывать на данные допол-

нительные ограничения, например определять диапазон допустимых значений или даже конструировать составные типы данных, способные комбинировать различные типы данных в рамках одного элемента или атрибута.

Созданная структура может быть объявлена как глобально (в качестве прямого наследника `<schema>`), так и локально в более глубоких уровнях вложения схемы. В первом случае структура обязательно должна быть именованной, чтобы позволить разработчику ссылаться на них при определении локальных элементов.

Предложенная на рис. 19.1 XML-схема уже содержит до сих пор неза-служенно обойденное нашим вниманием объявление структуры комплексного типа `<complexType>`. Это указание анализатору XML на то, что создается сложная структура, в составе которой имеется некоторый перечень элементов.

Продемонстрируем это на примере комплексной структуры для хранения домашнего адреса (листинг 19.16).

Листинг 19.16. Комплексная структура Address

```
<xs:complexType name="Address">
  <xs:sequence>
    <xs:element name="zip" type="xs:decimal"/>
    <xs:element name="city" type="xs:string"/>
    <xs:element name="street" type="xs:string"/>
    <xs:element name="home" type="xs:string"/>
    <xs:element name="apartment" type="xs:decimal"/>
  </xs:sequence>
</xs:complexType>
```

Структура Address содержит 5 элементов, отвечающих за хранение почтового индекса, название города, улицы, номер дома и квартиры. Комплексная структура может содержать одно из объявлений, уточняющее состав структуры (табл. 19.5).

В состав сложной структуры могут входить группы элементов, заданные с помощью объявления `<group>`. Такое решение весьма полезно, когда одну и ту же модель содержимого следует представить в нескольких элементах. Допустим, что у нас имеется три разнотипных элемента с именами `item1`, `item2` и `item3`. Все перечисленные элементы объявлены глобально (листинг 19.17), т. е. являются прямыми потомками `<schema>`.

Листинг 19.17. Группа элементов

```
<xs:element name="item1" type="xs:string"/>
<xs:element name="item2" type="xs:double"/>
<xs:element name="item3" type="xs:decimal"/>
```

Тогда на основе трех глобальных элементов мы способны создать глобальное объявление группы (листинг 19.18).

Листинг 19.18. Глобальное объявление группы

```
<xs:group name="itemGroup">
  <xs:sequence>
    <xs:element ref="item1"/>
    <xs:element ref="item2"/>
    <xs:element ref="item3"/>
  </xs:sequence>
</xs:group>
```

Глобальное определение обязательно должно быть именованным. Для того чтобы наш дочерний элемент смог воспользоваться услугами объявленной глобально группы, нам потребуется примерно такая строка: `<group ref="itemGroup">`.

В табл. 19.4 вы найдете несколько объявлений, позволяющих осуществить тонкую настройку структуры `<complexType>`.

Таблица 19.4. Спецификация состава комплексной структуры `<complexType>`

Объявление	Описание
<code><simpleContent></code>	Результирующая структура может включать только атрибуты и текстовые данные, элементы или смешанное содержимое внутри объявления <code><simpleContent></code> находиться не могут
<code><complexContent></code>	Результирующая структура может иметь атрибуты, элементы или смешанное содержимое и даже быть пустой
<code><group></code>	Позволяет создавать ссылки на группу элементов, определенную в глобальном объявлении схемы: <pre><xs:complexType name="demo"> <xs:group ref="itemGroup"/> <!-- другие элементы структуры --> </xs:complexType></pre> При указании ссылки требуется использовать префикс имени и имя группы

Объявление	Описание
<code><sequence></code>	Элементы структуры должны передаваться строго в заданной последовательности
<code><choice></code>	Указание на то, что в экземпляре XML-документа может содержаться только одно из объявлений элементов, перечисленных после объявления <code><choice></code> . Например, объявление <pre> <choice> <element name="weight" type="float"> <element name="size" type="float"> </choice> </pre> говорит, что в документ с данными может быть включен элемент «weight» или «size». Одновременное включение обоих элементов запрещено
<code><all></code>	Антипод объявления <code><sequence></code> . При работе с XML-документами объявление <code><all></code> разрешает нам заполнять элементы структуры в любом порядке

Некоторые из перечисленных в табл. 19.5 объявлений могут включать в свой состав вложенные объявления. Например, последовательность `<sequence>` вправе включать внутренние элементы `<sequence>`, `<choice>` и ссылки на `<group>`.

Атрибуты схемы

Если вы хорошо разобрались с особенностями объявления элементов схемы, то изучение основ декларирования атрибутов не должно вызывать особых затруднений. Тем более что синтаксическая конструкция определения атрибута очень схожа с конструкцией работы с элементом.

```

<attribute
  name="имя атрибута"
  type="тип атрибута"
  ref="ссылка на глобальное объявление"
  form="квалифицирован (qualified) или не квалифицирован (unqualified)"
  use="опциональный (optional), или запрещенный (prohibited),
        или обязательный (required)"
  default="значение по умолчанию"
  fixed="фиксированное значение">

```

Рассмотрим пример работы с атрибутами. В листинге 19.19 мы вновь вернемся к системному блоку компьютера и усовершенствуем описание процессора.

Листинг 19.19. Применение атрибутов для описания процессора

```

<xs:complexType name="CPU">
  <xs:attribute name="Manufacturer" type="xs:string" use="required"/>
  <xs:attribute name="CoreCount" type="xs:byte" default="1"/>
  <xs:attribute name="Frequency" type="xs:positiveInteger" use="required"/>
  <xs:attribute name="Notes" type="xs:string" use="optional"/>
</xs:complexType>

```

Теперь процессор представлен в виде комплексной структуры, включающей 4 атрибута. Атрибут `Manufacturer` предназначен для описания производителя процессора, `CoreCount` определит количество ядер, `Frequency` – тактовой частоты, `Notes` – примечания.

Подключение XML-схемы к документу

Присоединение XML-схемы к экземпляру документа обеспечивается с помощью атрибута `schemaLocation`. Порядок объявления следующий: сначала следует пространство имен URI, затем пробел и расположение схемы. Первые строки XML-документа, ссылающегося на файл схемы `computers.xsd`, могут выглядеть так, как представлено в листинге 19.20.

Листинг 19.20. Подключение схемы `computers.xsd` к документу XML

```

<?xml version="1.0" encoding="utf-8"?>
<COMPUTERS xmlns="http://www.myurl.ru/computers"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.myurl.ru/computers computers.xsd">
<!--Элементы документа -->
</COMPUTERS>

```

Допускается одновременно ссылаться на несколько схем, в этом случае они просто разделяются запятыми.

Полученные знания позволяют нам создать достаточно серьезную версию схемы для хранения данных о системном блоке компьютера, вы ее найдете в приложении 2.

Поддержка XML в СУБД

Большинство современных СУБД практически одновременно с выходом стандарта XML включило поддержку этого формата в свои продукты.

Компания Oracle и ее одноименный продукт приобрели возможность работы с XML-документами начиная с Oracle9i, для этого был введен специальный тип данных XMLType и подключена новая подсистема, названная XML DB.

Microsoft SQL Server 2000 развил инструкцию выборки SELECT, добавив в нее конструкцию FOR XML. В результате мы получили возможность осуществлять выборку не в виде обычного реляционного отношения, а в виде кода XML. Кроме того, в SQL Server появилась функция OPENXML, позволяющая открыть документ соответствующего формата и, например, использовать полученные данные в операции соединения или перенаправить инструкциям INSERT...SELECT или SELECT INTO.

Более того, существуют СУБД, полностью основанные на технологии XML, пожалуй, самая известная из них – eXist. Это кроссплатформенная СУБД с открытым кодом (для загрузки дистрибутива обратитесь на сайт разработчика www.exist-db.org), которая для работы с данными применяет языки XQuery и XSLT.

Резюме

Стандарт XML предназначен для создания иерархически структурированных документов, позволяющих хранить и обмениваться данными. XML молниеносно захватил огромный кусок рынка программного обеспечения. Достаточно сказать, что стандарт XML применяется в текстовых редакторах, базах данных, электронной коммерции, интернете, графических приложениях. Вокруг XML сосредоточено многочисленное семейство стандартов и языков, опирающихся на XML или выполняющих вспомогательные функции. Среди них:

- XSL (eXtended Stylesheet Language) и XSLT (XSL Transformations) – расширяемый язык преобразования стилей, предназначенный для преобразования XML-данных в другие форматы, чаще всего в HTML;
- XPath (XML Path Language) – язык, позволяющий обслуживать отдельную часть XML-документа;
- XQUERY – язык запросов XML;
- XLink – опирающийся на атрибуты синтаксис для использования ссылок в XML-документах;
- XForms – технология форм, конкурирующая с формами HTML;
- SVG (Scalable Vector Graphics) – многоцелевой формат масштабируемой векторной двумерной графики.

Для повышения качественного уровня структуризации документа и придания дополнительной строгости к хранимым в документе XML-данным стандартом предусмотрена возможность создания метаязыка данных. Рассмотренное в главе описание документов в формате DTD позволяет определить связи между элементами документа (отношение главный–дочерний), поставить признак обязательности, назначить последовательность расположения элементов, предоставить варианты выбора значений, ограничить количество элементов.

Несколько ограниченные возможности DTD очень скоро перестали устраивать разработчиков, создающих проекты средней и высокой сложности. Например, для корректного обмена данными между БД как минимум требуется уметь проверять тип данных, но описание DTD на это неспособно. Именно поэтому в 2001 году консорциум W3C ввел новый, обладающий более широкими семантическими возможностями стандарт, в котором для определения содержания документа вводилось понятие XML-схемы (XML schema).

Вопросы для самопроверки

1. Каково назначение расширяемого языка разметки?
2. Для чего применяются анализаторы XML?
3. Как проверяется корректность документа XML?
4. Для чего предназначено определение документа DTD?
5. Что входит в состав документа XML?
6. По каким правилам следует описывать элементы документа XML?
7. Каковы назначение и порядок применения атрибутов в документе XML?
8. Что такое пространство имен?
9. Каковы преимущества XML Schemas над DTD?
10. Спроектируйте документ XML и XML Schemas для описания:
 - а) домашней фонотеки;
 - б) каталога книг в библиотеке;
 - в) прайса в магазине электроники.

Глава 20

Клиент-серверные БД

Начиная с 1980-х годов разработчики проектов БД стали проявлять повышенный интерес к архитектуре клиент-сервер. Основная идея клиент-серверной технологии в БД заключается в разделении функций обработки и представлении данных между участниками процесса обработки данных. Здесь под клиентом подразумевается некий процесс, запрашивающий сервис или ресурс от другого серверного процесса. В свою очередь, сервер – это процесс, который предоставляет сервисы и/или ресурсы клиентскому процессу. В клиент-серверной архитектуре клиенты соответствуют интерфейсной части (front-end) системы, а сервер выступает в роли прикладной части (back-end).

Период взаимодействия клиента и сервера называется сеансом (сессией). Один и тот же сервер, как правило, способен одновременно поддерживать один и более сеансов, таким образом обслуживая несколько клиентов.

Определение

Сеанс (session) – определенный период времени, в течение которого клиент может много раз взаимодействовать с сервером, причем и клиент, и сервер поддерживают данные друг о друге.

Технически сервер и клиент могут исполняться на одном и том же компьютере (такое размещение очень удобно на этапе программирования), но весь смысл идеи раскрывается, только когда сервер развертывается на выделенной, высокопроизводительной станции. В таком случае сервер сможет предоставлять свои услуги одновременно нескольким клиентам.

Модель взаимодействия открытых систем

Помимо клиента и сервера, клиент-серверные системы включают и третий компонент – промежуточное программное обеспечение (mid-

Middleware). По своей сути промежуточное ПО представляет собой процесс (процессы), обеспечивающий взаимодействие между клиентом и сервером (рис. 20.1).

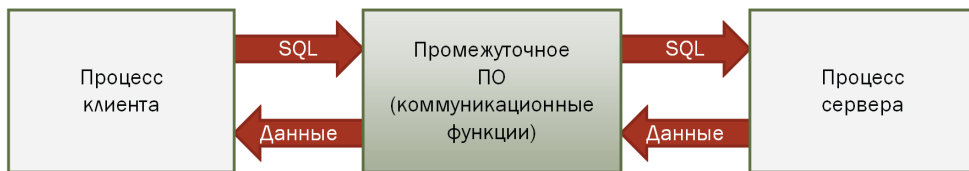


Рис. 20.1. Взаимодействие клиентского и серверного процессов

В современных клиент-серверных системах промежуточное ПО в первую очередь осуществляет коммуникационные функции, доставляя серверу запросы клиента и возвращая назад данные. В компьютеризированных системах коммуникационные функции реализуются операционными системами, которые, в свою очередь, взаимодействуют с сетевым оборудованием.

Полноценное архитектурное решение «клиент-сервер» появилось на свет одновременно со всеобщим признанием концепции открытых систем. Задачей этой концепции является стандартизация аппаратных и программных интерфейсов, что в результате привело к упрощению комплексирования компьютерных систем.

Замечание

Модель взаимодействия открытых систем (Open Systems Interconnection, OSI) была предложена организацией ISO в 1984 году. Эталонная модель включает 7 независимых уровней, причем каждый нижележащий уровень предоставляет определенные услуги для вышеразположенного уровня.

Реализованный в OSI многоуровневый подход заключается в том, что все множество модулей, предоставляющих определенные сервисы, разбивают на группы и упорядочивают по уровням, образуя 7-уровневую иерархию (рис. 20.2). Состав уровней OSI таков [32, 33, 38]:

- **прикладной уровень.** Обеспечивает преобразование данных, специфичное для каждого конкретного приложения, в нашем случае для СУБД и клиентского приложения, отправляющего SQL-запрос в адрес БД;
- **уровень представления.** Осуществляет преобразование данных общего характера (преобразование данных в формат, по-

нятный для принимающего узла; шифрование данных; сжатие данных);

- **сеансовый уровень.** Устанавливает соединение между сервером и клиентом и управляет им, при необходимости восстанавливает разорванное соединение;
- **транспортный уровень.** В интересах вышележащих уровней осуществляет свободную от ошибок, ориентированную на работу с сообщениями сквозную передачу;
- **сетевой уровень.** Служит для образования единой транспортной системы, объединяющей несколько сетей, обеспечивает маршрутизацию и управление загрузкой канала передачи, представленного только конечными точками;
- **канальный уровень.** Осуществляет свободную от ошибок передачу по отдельному каналу связи;
- **физический уровень.** Выполняет реальную передачу бит данных по физическим каналам связи.

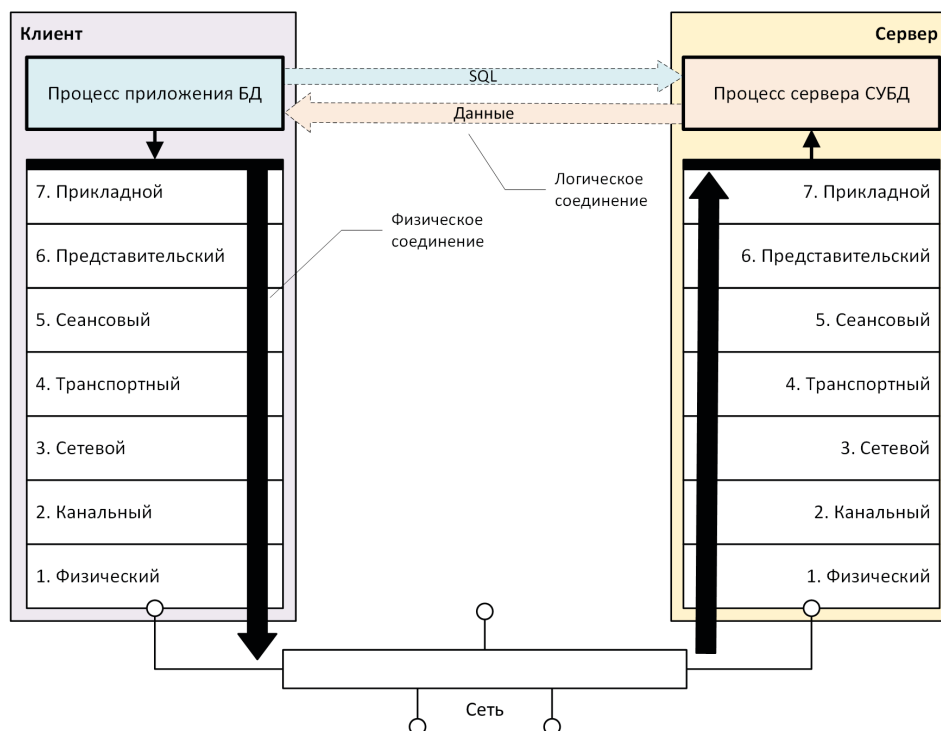


Рис. 20.2. Взаимодействие клиентского и серверного процессов

Группа программных, программно-аппаратных или просто аппаратных модулей, составляющих каждый уровень, формируется так, чтобы все модули уровня для выполнения возложенного на них функционала обращались с запросами только к модулям соседнего ниже лежащего уровня. Для взаимодействия между уровнями одного и того же узла используются интерфейсы, определяющие набор функций, которые ниже лежащий уровень предоставляет уровню, расположенному выше. Для обмена сообщениями между модулями одного и того же уровня, но расположенными на разных узлах, разрабатываются протоколы.

Определение

Протокол – это формализованные правила, с помощью которых осуществляется сетевое взаимодействие расположенных в разных узлах модулей одного и того же уровня.

Определение

Интерфейс – это формализованные правила, с помощью которых взаимодействуют находящиеся в одном узле модули, реализующие протоколы соседних уровней.

Уровни работают таким образом, что расположившемуся над ними конечному пользователю можно не заботиться о том, каким образом осуществляется передача данных от одного сетевого узла к другому. Точно так же, как вы, совершая звонок по мобильному телефону, не задумываетесь об особенностях распространения радиосигнала, о помехоустойчивости, задаче поиска абонента, преобразовании аналогового сигнала в цифровую форму и т. п., – вместо этого вам просто требуется набрать номер, все остальное сделает ваш телефон и коммуникационное оборудование.

Замечание

В современных компьютерных сетях доминирующие позиции занимает стек протоколов TCP/IP, гарантирующий доставку пакетов между узлами сети. И хотя стек TCP/IP появился на несколько лет раньше, чем модель OSI, заложенные в нем идеи вполне современны и во многом соответствуют требованиям открытых систем. Так, протокол интернета IP можно отнести к сетевому уровню модели OSI, а TCP принадлежит транспортному уровню. Позднее над уровнями TCP/IP был создан ряд высокоуровневых надстроек, активно применяемых в клиент-серверных БД, – это сокеты, именованные каналы, удаленный вызов процедур и т. п.

Клиент-серверные СУБД

Центральным звеном большинства клиент-серверных БД выступает система управления базами данных. Помимо решения стандартных задач (напомним о существовании функциональных обязанностей СУБД, сформулированных доктором Коддом в 1980-х годах, основные из них были изложены в *главе 2*), сервер БД обеспечивает:

- 1) прозрачный доступ к данным для клиентов. В идеале у удаленного пользователя должно сложиться впечатление, что БД расположена на его же машине;
- 2) независимость от аппаратной, сетевой и программной платформ для станции, на которой выполняется клиентское приложение;
- 3) доставку по сети запросов SQL к БД и обработку их;
- 4) возврат клиенту результатов выполнения запросов.

На сегодняшний день клиент-серверные СУБД стали наиболее востребованной архитектурой в проектах БД. Это произошло благодаря существенным преимуществам данного решения над простыми файл-серверными БД. К достоинствам клиент-серверных проектов можно отнести:

- 1) повышение производительности системы за счет перераспределения обязанностей между сервером и клиентами;
- 2) централизованное хранение данных приводит к повышению целостности и непротиворечивости данных, т. к. все ограничения проверяются в одном месте;
- 3) из-за того, что данные сосредоточены в одном месте, а не распределены по всем компьютерам предприятия, существенно упрощается решение задачи обеспечения безопасности данных;
- 4) снижается нагрузка на сеть предприятия, т. к. обмен между клиентом и сервером сводится к передаче SQL-запроса и возврату запрашиваемых данных.

Замечание

В *главе 22* мы сравним преимущества и недостатки централизованных и распределенных баз данных.

Модели распределения функций

Ключевой принцип технологии клиент-сервер в проектах БД заключается в разделении между участниками процесса функций по работе с

данными. Обычно выделяют следующий перечень уровней функций [38, 47, 50].

1. **Функции презентационной логики** определяют особенности представления данных на экране и графический интерфейс пользователя в целом.
2. **Функции бизнес-логики** представляют собой алгоритмы решения задач приложения, указанная группа функций обычно реализуется на процедурном SQL и/или на языках программирования высокого уровня.
3. **Функции логики БД** обеспечивают решение задач выборки, вставки, удаления и редактирования данных. Перечисленные функции обработки данных обычно реализуются средствами SQL.
4. **Функции СУБД** обеспечивают управление информационными ресурсами.

В зависимости от того, как перечисленные группы функций распределены между клиентом и сервером, говорят о том или ином типе 2-уровневого клиент-серверного проекта баз данных (рис. 20.3).

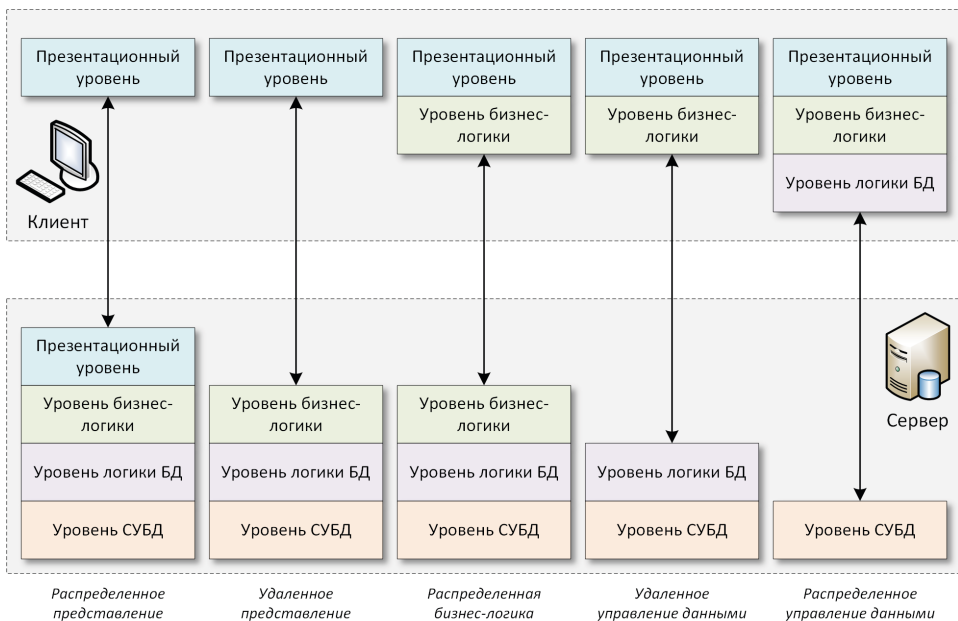


Рис. 20.3. Распределение функций между клиентом и сервером в 2-уровневых проектах БД

На ранних этапах развития клиент-серверной архитектуры доминирующим подходом считалось распределенное представление и удаленное представление данных. И в первом, и во втором случаях клиент частично или полностью выполнял лишь задачи, связанные с выводом данных на экран и вводом/редактированием этих данных пользователем. Все остальное делал сервер. Из-за мизерных программных и аппаратных затрат на клиента (очень часто в качестве клиентов использовались бездисковые станции с удаленной загрузкой) подобные проекты обычно называют «тонким клиентом» (thin client).

В настоящее время, когда стоимость персональных компьютеров стала не столь высокой, а производительность существенно возросла, более популярными стали модели с распределенной бизнес-логикой и удаленным управлением данными. В подобных проектах клиентские станции частично или полностью забирают у сервера функции, реализующие бизнес-логику проекта БД. Тем самым с сервера существенно снижается нагрузка и, как следствие, упрощается масштабирование проекта.

Модель распределенного управления данными предполагает, что клиент возьмет на себя ответственность практически за весь функционал проекта БД, оставив за сервером лишь группу задач, связанных с централизованным хранением и обработкой данных в СУБД. Подобный подход допускает, что клиент сможет на временной основе загружать данные с сервера в свою собственную локальную БД и работать с ними в условиях отсутствия постоянного соединения с сервером. За то, что основная нагрузка по обработке данных выполняется клиентом, его называют «толстым» (fat client).

Хотя 2-уровневые модели распределения обязанностей между клиентом и сервером сегодня наиболее популярны, имеются и еще более сложные решения. Среди них выделяется 3-уровневая модель сервера приложений (Application Server, AS). Идея AS заключается во внедрении между сервером и клиентом дополнительного программного уровня – своеобразного интеллектуального моста между удаленными клиентскими приложениями и СУБД (рис. 20.4).

Мост выступает не просто в качестве банального посредника, перекачивающего потоки бит из БД к клиенту и обратно. Основная задача промежуточного приложения – облегчение работы клиента за счет предоставления ему максимально возможного перечня услуг, вплоть до того, что клиент превращается просто в средство просмотра данных (тонкого клиента), а вся программная логика сосредоточивается в приложении-посреднике.



Рис. 20.4. Модель сервера приложений в 3-уровневых проектах БД

Промежуточный уровень выступает не только помощником клиентского приложения, но и оказывает существенную поддержку серверу БД, перекладывая на себя часть его задач. Например, по поддержке бизнес-правил, упорядочиванию данных, фильтрации и поиску, построению отчетов и т. д. В той ситуации, когда СУБД обслуживает большое количество клиентов, любая помощь, снимающая часть вычислительной нагрузки, не окажется лишней.

Обсуждая плюсы многоуровневых проектов, отметим, что наличие дополнительного логического звена в цепи БД–клиент существенно упрощает сопровождение системы в целом. Благодаря тому что в тонких клиентах присутствует минимум программной логики (нет кода – нет ошибок), разработчик ПО промежуточного уровня практически не отвлекается на выпуск обновлений для пользовательских компьютеров. Есть возможность сосредоточить основное внимание на совершенствовании промежуточного ПО.

Очевидный минус 3-уровневых решений – повышение сложности (и как следствие – стоимости) проекта БД за счет внедрения дополнительного слоя обработки данных. Для реализации подобного проекта потребуется задействовать высококвалифицированных программистов, имеющих все необходимые компетенции по разработке сервера приложений.

Резюме

Модель клиент-сервер основана на распределении функций между двумя типами процессов – клиентом и сервером. Клиентский процесс нуждается в помощи серверного процесса, для этого он запрашивает от сервера определенные услуги. Сервер выполняет поставленную задачу и возвращает клиенту полученные результаты.

Архитектура клиент-сервер опирается на концепцию открытых систем. Эталонная модель взаимодействия открытых систем (Open

Systems Interconnection, OSI) была предложена организацией ISO в 1984 году.

Клиент-серверные базы данных позволяют предоставлять транспарентный доступ к данным одному или нескольким клиентам. Как правило, основная обработка данных в таких системах выполняется СУБД, а клиент освобождается от локальной обработки данных.

Вопросы для самопроверки

1. Опишите назначение уровней в модели взаимодействия открытых систем (OSI).
2. Раскройте принципы клиент-серверной архитектуры.
3. Поясните смысл терминов «интерфейс» и «протокол».
4. Какой функционал возлагается на СУБД, работающую в клиент-серверной архитектуре?
5. Какими достоинствами обладают БД, работающие в клиент-серверной архитектуре?
6. Какой сетевой протокол в настоящее время наиболее распространен?
7. Как могут распределяться обязанности между сервером и клиентом в 2-уровневой клиент-серверной модели БД?
8. Какое принципиальное отличие «тонкого» клиента от «толстого»?
9. Прокомментируйте особенности 3-уровневой модели сервера приложений.

Глава 21

Особенности разработки клиента БД

Выпуск проекта БД – сложный и кропотливый процесс, объединяющий в себе физическую реализацию БД и разработку клиентской части проекта. С процессом подготовки к проектированию, концептуальным, логическим и физическим этапам проектирования БД мы уже хорошо знакомы. Перечисленные работы осуществляются на стороне сервера разработчиками БД. Теперь настал черед обсудить некоторые важные особенности разработки клиентского программного обеспечения – программного продукта, предоставляющего доступ к данным конечным пользователям.

Во время разработки клиентского ПО, помимо программного обеспечения, поставляемого совместно с СУБД, задействуется широкий спектр инструментального ПО: языки программирования, интегрированные среды разработки, CASE-системы, графические и текстовые редакторы и др.

Выбор языка программирования

Для разработки программного обеспечения БД, работающего на стороне клиента, обычно используют высокоуровневые объектно-ориентированные языки программирования, такие как C++, C#, Delphi, Java, и другие стандартизированные [53] и пока не вошедшие в стандарт языки. Перечисленные языки программирования позволяют создавать кроссплатформенные приложения (одинаково хорошо работающие в большинстве современных операционных систем) с развитым графическим интерфейсом пользователя.

Так же, как и в случае с выбором СУБД, при выборе языка программирования и среды разработки программисты руководствуются требова-

ниями заказчика, характером проекта, экономическими параметрами, собственными предпочтениями и привычками и т. п. Однако ключевым критерием, способным склонить чашу весов в пользу того или иного языка программирования, считается наличие у среды разработки развитых технологий доступа к данным, позволяющим клиенту обращаться к серверу БД.

Технология доступа к данным ODBC

Технология открытого доступа к БД ODBC (Open Database Connectivity) представляет собой программный интерфейс (API) доступа к базам данных. Интерфейс был разработан относительно давно – в начале 1990-х, однако до сих пор является востребованным. Основным создателем ODBC считается Microsoft, хотя свою лепту внесли Access Group, X/Open и Simba Technologies. Первые две организации вместе с Microsoft трудились над спецификацией CLI (Call Level Interface), на которой и базируется ODBC, а Simba Technologies оказывала помощь при кодировании действий разработчиков.

Замечание

Спецификация интерфейса уровня вызовов CLI в 1993 году была утверждена в качестве составной части стандарта SQL (ISO-стандарт ISO/IEC 9075-3:1995 Information technology – Database languages – SQL – Part 3: Call-Level Interface (SQL/CLI)).

Появление ODBC позволило разработчикам клиентских приложений для Windows использовать одинаковый интерфейс доступа к данным, не вникая в подробности организации взаимодействия с различными источниками данных. Все задачи решаются средствами программного интерфейса и соответствующими драйверами (рис. 21.1).

В состав ODBC входят [54]:

- 1) ODBC API – библиотека вызовов функций, набора кодов ошибок и инструкций SQL для доступа к данным в СУБД;
- 2) диспетчер драйверов ODBC загружающей драйвера базы данных от имени приложения;
- 3) драйверы базы данных ODBC, представляющие собой одну или несколько библиотек DLL, умеющих обрабатывать вызовы ODBC для конкретных СУБД;
- 4) библиотеки управления курсорами ODBC;
- 5) администратор ODBC.

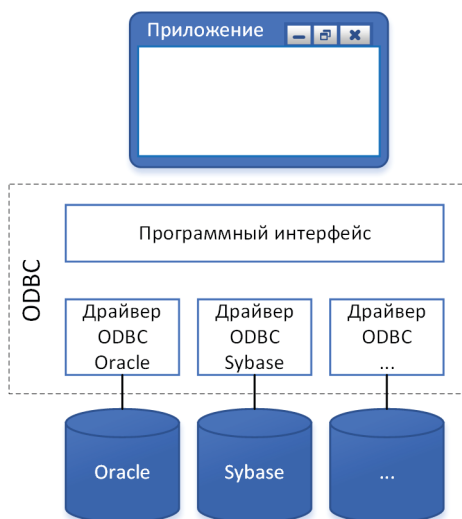


Рис. 21.1. Модель проекта на основе ODBC

Замечание

ODBC – одна из самых старых, но до сих пор применяемых технологий. С помощью ODBC разработчики ПО могут создавать приложения, используя единый интерфейс доступа к данным, не заботясь о нюансах взаимодействия с разными источниками. Однако за более чем четверть века после появления ODBC были разработаны новые, зачастую более доброжелательные интерфейсы доступа к БД, в их числе: ADO .NET, FireDAC и JDBC.

Технология доступа к данным ADO .NET

Если вы являетесь приверженцем языка C# или разрабатываете приложения на любом другом языке, поддерживающем платформу .NET Framework компании Microsoft, скорее всего, при проектировании клиентского приложения проекта БД вы воспользуетесь технологией доступа данных ADO .NET (ActiveX Data Object .NET).

ADO .NET предоставляет собой набор классов и клиентских библиотек, обеспечивающих доступ к самому широкому спектру источников данных, в частности к SQL Server (компании Microsoft), Oracle, данным в формате XML, а также к источникам данных, предоставляемым при помощи OLE DB и ODBC. Механизм доступа к источникам данных построен таким образом, чтобы программист мог не задумываться об особенностях той или иной целевой СУБД (рис. 21.2).

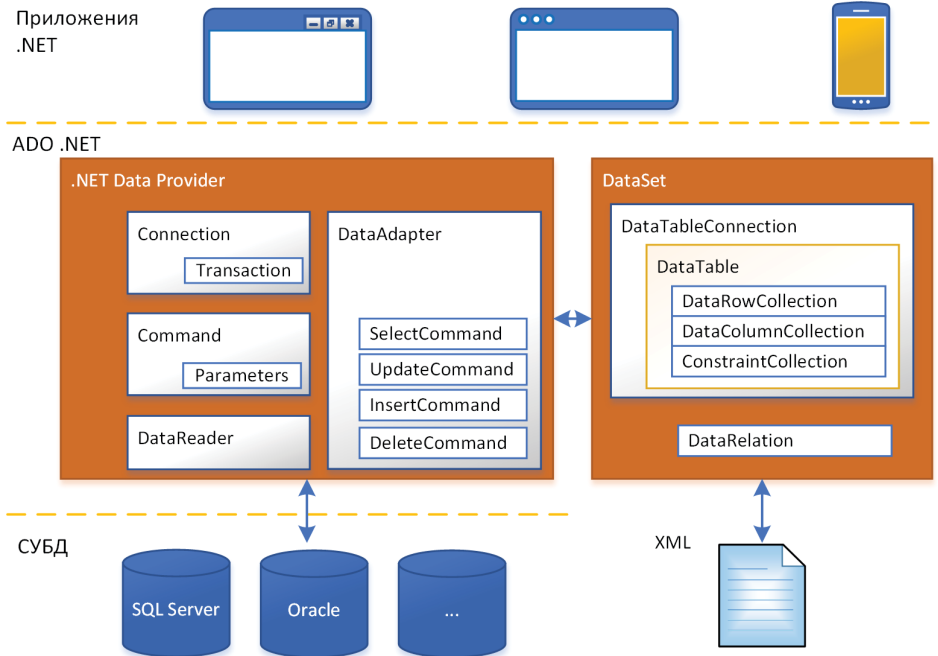


Рис. 21.2. Объектная модель проекта ADO .NET

Раскроем назначение основных объектов, составляющих проект ADO .NET:

- доступ к данным осуществляют те или иные провайдеры данных .NET Data Provider, которые и отвечают за взаимодействие с конкретным сервером БД;
- объект **Connection** используется для установления соединения с источником данных, а также для управления транзакциями;
- объект **Command** позволяет манипулировать данными источника, а также выполнять хранимые процедуры, в этих целях могут использоваться параметры **Parameters** для передачи данных в обоих направлениях;
- объект **DataAdapter** служит связующим звеном между набором данных **DataSet** и источником данных. Он использует объект **Command** для выполнения команд SQL как для заполнения **DataSet** данными, так и для обратной передачи измененных клиентом данных к источнику. Для выполнения этих функций объект имеет 4 метода: **SelectCommand**, **InsertCommand**, **UpdateCommand** и **DeleteCommand**;

- объект `DataReader` представляет однонаправленный поток данных от источника только на чтение. Если приложение клиента не модифицирует данные и не требуется произвольная выборка данных, а достаточно их однократного просмотра, то использование `DataReader` вместо `DataSet` позволит сохранить вычислительные ресурсы, а также поднять быстродействие приложения;
- объект `DataSet` осуществляет универсальный доступ данных (независимо от их источника) и одинаково хорошо работает как с данными, поступившими из БД, так и с XML. Экземпляр класса `DataSet` содержит коллекцию одного или нескольких объектов `DataTable`, состоящих из строк и столбцов данных, а также первичный ключ, внешний ключ, ограничение и связанные сведения о данных в объектах. В отличие от `DataReader`, объект `DataSet` осуществляет локальное кеширование данных на стороне клиента для их последующей обработки, поэтому он более ресурсоемок;
- в наборе `DataSet` с несколькими объектами `DataTable` можно использовать объекты `DataRelation` для связи таблиц друг с другом, для перемещения по таблицам, а также для возвращения дочерних или родительских строк из связанной таблицы.

Технология доступа к данным FireDAC

Архитектура FireDAC применяется для доступа к данным в приложениях, написанных в среде проектирования Delphi и C++ Builder компании Embarcadero. FireDAC представляет собой надстройку, способную взаимодействовать с большинством современных СУБД. Технология представлена в виде нескольких слоев драйверов и библиотек. Каждый из слоев отвечает за свой уровень абстракции данных (рис. 21.3):

- слой драйверов обеспечивает доступ к низкоуровневому API соответствующих БД;
- адаптер данных отвечает за более высокий слой абстракции, на котором осуществляется работа со структурированными данными, например с отношениями главный–подчиненный, вложенными данными;
- локальное хранилище данных предназначено для хранения и обработки данных на уровне приложения;
- отладчик и монитор производительности позволяют программисту выявлять, локализовать и устранять ошибки в коде, а также оценивать его эффективность;

- на верхнем уровне абстракции находятся визуальные и невидимые компоненты, позволяющие создавать кроссплатформенные проекты БД.

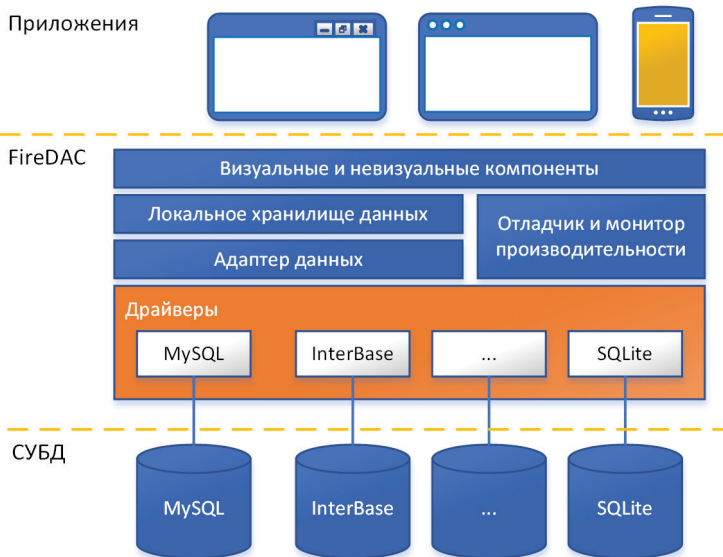


Рис. 21.3. Архитектура FireDAC

Кроме практически не ограниченной всеядности (как уже отмечалось, FireDAC поддерживает все современные популярные СУБД), эту технологию выделяет высокая производительность доступа к данным и удобство применения.

Замечание

Российскому читателю платформа FireDAC должна быть интересна вдвойне, так как у ее истоков стоит отечественный разработчик Дмитрий Арефьев.

Клиентское приложение, создаваемое на основе FireDAC, строится из заранее подготовленных компонентов, которые программист обнаружит на палитре компонентов среды разработки. Например, клиент, предназначенный для работы с СУБД MySQL, скорее всего, будет состоять из компонентов, представленных на рис. 21.4 [36].

Все имеющие прямое отношение к построению клиент-серверных проектов БД объектно-ориентированные компоненты Delphi (C++ Builder) разделяются на следующие категории:

- 1) компоненты, обеспечивающие соединение приложения с базой данных, в линейке FireDAC такой компонент называется TFDConnection;
- 2) компоненты, реализующие объектно-ориентированное представление реляционных наборов данных (TFDTable, TFDQuery, TFDStoredProc и TFDMemTable);
- 3) источник данных, связывающий конкретный набор данных (например, таблицу) с элементами управления данными;
- 4) визуальные элементы управления данными, умеющие отображать и редактировать записи в наборе;
- 5) вспомогательные компоненты с широким спектром решаемых задач (например, настройка параметров соединения, управление несколькими соединениями одновременно, определение характеристик транзакций и т. п.).

FireDAC поддерживает впечатляющий перечень СУБД: SQL Server компании Microsoft, Oracle, InterBase, FireBird, Informix, PostgreSQL, IBM DB2 Server, Sybase SQL Anywhere, Mongo DB и т. д.

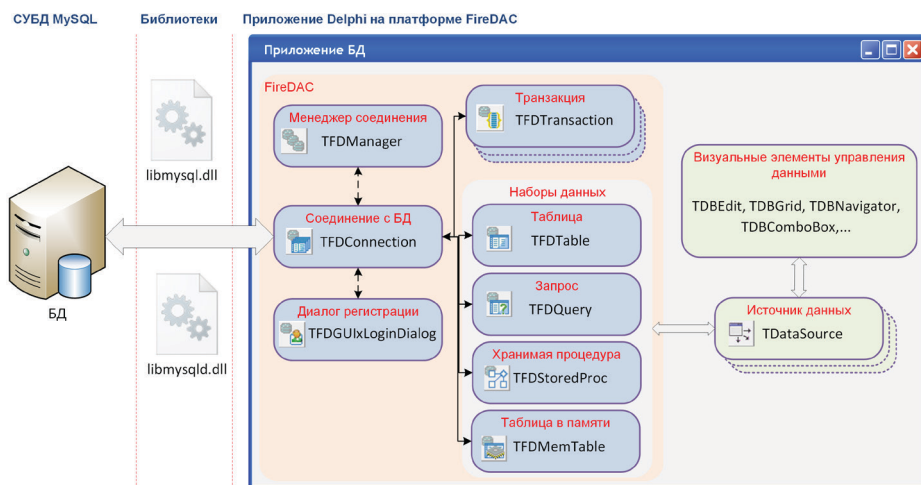


Рис. 21.4. Клиентское приложение БД для платформы MySQL – FireDAC

Технология JDBC

При разработке клиентских приложений на языке Java для соединения с БД задействуют платформенно-независимый промышленный стандарт JDBC (Java DataBase Connectivity). Архитектура приложения, использующего JDBC для доступа к БД, предложена на рис. 21.5.

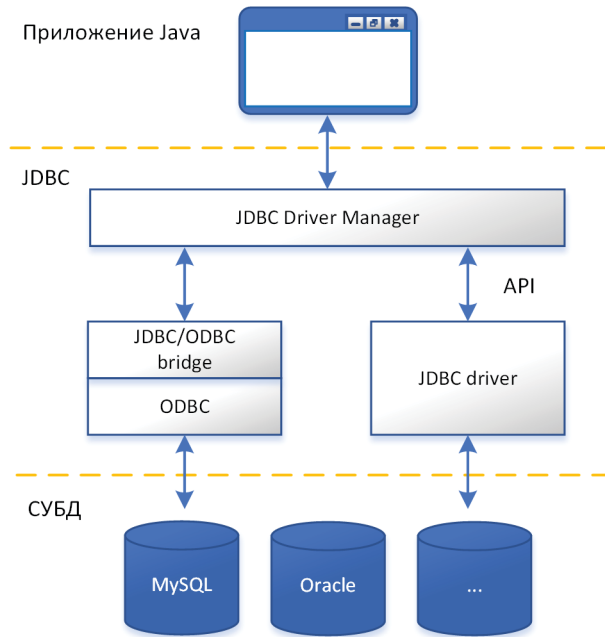


Рис. 21.5. Архитектура JDBC

Для доступа к БД технология JDBC применяет 2- или 3-звенные модели. В 2-звенных моделях приложение обращается непосредственно к БД при посредничестве JDBC-драйвера, написанного для конкретной СУБД. В 3-звенной модели работа с БД производится через стек ODBC и мост JDBC/ODBC. В обоих случаях за соединение Java-программы с соответствующим драйвером отвечает менеджер драйверов JDBC Driver Manager.

Интерфейс клиента

Насколько бы не был идеален исходный код, рациональны и эффективны запросы, глубоко продуманы концептуальная, логическая и физическая модели БД, все равно потенциальный пользователь встретит ваш проект «по одежке». Обычному оператору компьютера, скорее всего, окажутся безразличны цикломатическая сложность и изящество программных конструкций, ему гораздо важнее удобство и простота работы с программным продуктом. Так что первая оценка, которую получит проект, скорее всего, будет выставлена за пользовательский интерфейс. А если интерфейс окажется неудачным, то, возможно, первая

оценка окажется последней, а предпочтения будут отданы разработкам конкурентов...

Как создать проект с качественным пользовательским интерфейсом? Постараемся ответить на этот вопрос.

Замечание

Под пользовательским интерфейсом понимают совокупность программных и аппаратных средств, обеспечивающих взаимодействие пользователя с компьютером.

Сколько людей, столько и мнений

Как у вас, так и у меня сложились свои предпочтения и привычки. У каждого имеется свое собственное мнение насчет буквально всего, что окружает нас в этом мире, от выбора одежды до любимой музыки. Так что вполне естественно, что у любого человека есть свой взгляд на то, как должен выглядеть пользовательский интерфейс программного продукта. А теперь вопрос: можно ли создать интерфейс, который способен угодить каждому?

Вопрос, скорее, риторический. Ответ вы уже знаете. Но, пожалуй, стоит попробовать удовлетворить предпочтения если не всех, то, по крайней мере, именно той категории пользователей, на которую и рассчитан программный продукт. А для этого надо добиться, казалось, немногого – сделать так, чтобы пользователь получил то, чего он ожидал. Так в чем же проблема?

А вот это уже не риторический вопрос, и ответ на него таков: проблема заключается в программистах. Дело в том, что у программиста и пользователя по определению сложились разные взгляды на интерфейс.

Программистов интересуют функциональность, эффективность, технологичность, надежность, внутренняя стройность и другие (обычно напрямую не связанные с удобством пользователя) характеристики разрабатываемого ими продукта. Попробуйте найти не имеющего отношения к программированию пользователя, который способен по достоинству оценить эти качества. Даже и не пытайтесь! Ведь у него на первый план выступают абсолютно другие критерии.

Пользователю абсолютно безразлично, что между его клиентским приложением и сервером БД проложено несколько десятков метров сетевого кабеля, он не имеет представления о конфликте транзакций и даже не предполагает, чем структурное программирование отличается от объектно-ориентированного. В основе пользовательской модели ин-

терфейса лежит обобщенное представление обычного человека о том, какие процессы происходят в программе во время ее работы. Что получится в результате, ответить достаточно сложно, однако достоверно известно, что результирующая пользовательская модель коррелирует с:

- знаниями пользователя предметной области разрабатываемого программного обеспечения;
- интуитивным пониманием того, как выполняются операции в этой предметной области;
- уровнем владения компьютером;
- сложившимися у него привычками работы с компьютером.

Так что если программист намерен создать успешную модель пользовательского интерфейса, то ему, во-первых, придется в совершенстве разобраться с предметной областью, для автоматизации которой предназначена программа, во-вторых, понять психотип будущего оператора компьютера и, в-третьих, обойти проблемы чрезмерных затрат на разработку интерфейсной части проекта. Задача, прямо скажем, нетривиальная.

Пользовательские критерии качества интерфейса

Существуют ли объективные критерии, по которым можно сказать, что этот интерфейс имеет шансы на успех, а этот – потенциальный неудачник? Безусловно. Ряд исследований, основанных на опросах пользователей, утверждает, что такими критериями являются [21]:

- простота освоения и запоминания операций программного продукта (основными показателями выступают время, затраченное оператором на изучение и продолжительность сохранения информации в памяти человека);
- скорость достижения результатов при использовании программного продукта – определяется количеством действий (команд, операций), необходимых для решения задачи (допустим, поиска нужной продукции на складе);
- субъективная удовлетворенность при эксплуатации системы (удобство работы, утомляемость и т. д.).

Несмотря на минимализм, с предложенным перечнем трудно не согласиться. Особенно радует, что первая пара критериев имеет четко выраженные количественные показатели (соответственно, время на изучение (запоминание) и количество действий, необходимых для достижения результата). Обладая количественной оценкой, мы смо-

жем выбирать лучший интерфейс, сравнивая разные его варианты. С третьим критерием несколько сложнее – он не количественный, а качественный. Здесь результат во многом зависит от предпочтений и настроения пользователя, его опыта работы за компьютером и еще от нескольких десятков других факторов.

Рекомендации по проектированию

Клиентское приложение БД может обладать всего одной или двумя формами, а может быть, их окажется несколько десятков. Какие общие рекомендации можно дать прикладному программисту, чтобы визуальный интерфейс приложения вместо отторжения вызывал у пользователя только положительные эмоции?

Предложим на суд читателя следующий перечень советов:

- главный совет заключается в том, что интерфейс не призван заставлять пользователя долго размышлять при выполнении той или иной операции – форма должна быть простой и понятной с первого взгляда;
- заголовок формы должен иметь краткое содержательное название, прочитав которое, пользователь однозначно и точно поймет, для чего форма предназначена;
- цветовое оформление должно иметь привлекательный внешний вид. Другими словами, не раскрашивайте элементы интерфейса в ультрамариновые цвета, а вместо этого используйте стандартные цветовые схемы, предлагаемые операционной системой. Стандартные цветовые решения разработаны профессиональными дизайнерами и не вызывают раздражения у подавляющего большинства людей. Таким образом, обнаружив в среде проектирования у элемента управления свойство, отвечающее за цвет семь, раз подумайте, прежде чем поменять значение по умолчанию. Если же такая необходимость все-таки существует (допустим, чтобы подчеркнуть важность элемента управления), то вместо явного назначения цвета воспользуйтесь константами, связанными с цветовой схемой. В этом случае если пользователь изменит тему оформления операционной системы, ваш интерфейс корректно подстроится под новую гамму;
- в проекте должны использоваться общепринятая терминология и сокращения, и не важно, что просторечие «платежка» кажется короче и удобнее для размещения в качестве заголовка элемен-

- та меню, чем «платежное поручение». Надписи, пояснительные подписи, всплывающие подсказки и т. п. должны целиком и полностью соответствовать терминологии той области знаний, для которой создавался программный продукт;
- названия и подписи элементов управления, в том числе заголовки столбцов таблиц, следует делать одинаковыми во всех формах приложения;
 - последовательности столбцов в таблицах и объединения полей ввода на панелях группировки должны иметь четкую логическую последовательность, и если оператор компьютера вводит в поле фамилию клиента, то, скорее всего, следующими полями должны оказаться имя и отчество, а не температура за окном;
 - при вводе данных везде, где это возможно, внедряйте вспомогательные средства, упрощающие ввод (поля подстановки, списки выбора, подбор продолжения текста по начальным символам и т. п.);
 - все формы приложения должны предоставлять пользователю единые средства перемещения курсора по полям ввода. В идеале пользователь захочет обладать возможностью передавать фокус ввода столбцам таблицы и элементам управления формы, не отрывая рук от клавиатуры (например, применяя клавишу **Tab** для движения вперед и комбинацию **Ctrl+Tab** для возврата назад);
 - все формы приложения должны обладать единым функционалом по просмотру и редактированию строк таблиц. Доступ к операциям должен осуществляться несколькими способами, например для удаления строки можно задействовать: элемент главного меню, элемент всплывающего меню, быструю кнопку на панели главной формы и комбинацию клавиш **Ctrl+Del**;
 - элементы управления, предоставляющие доступ к тем или иным операциям с данными, всегда должны находиться в актуальном состоянии. Например, если приходная накладная открыта в режиме только для чтения, то все элементы интерфейса, отвечающие за редактирование, должны быть переведены в неактивное состояние;
 - все элементы управления следует снабдить краткими всплывающими подсказками, в ситуации, когда краткой подсказки недостаточно, дополнительные пояснения следует отображать в дополнительной статусной строке;
 - там, где это возможно, позволяйте пользователю подстраивать интерфейс под себя (изменять ширину и последовательность

- столбцов в таблицах, формировать пользовательские панели инструментов, перенастраивать «быстрые» клавиши и т. п.);
- необходимо предусмотреть простые и надежные средства предотвращения ввода ошибочных данных (маски ввода, ограничения на длину текста, всплывающие предупреждения и т. п.);
- наличие средств визуального выделения обязательных и необязательных к заполнению полей;
- возникновение исключительной ситуации должно сопровождаться сообщением об ошибке с понятной инструкцией о дальнейших действиях пользователя;
- все потенциально опасные действия с данными (в особенности удаление) должны требовать от пользователя подтверждения;
- во время выполнения длительных операций в ситуации, когда пользователь вынужден ожидать их завершения, отобразите на экране динамическое сообщение (шкалу) с указанием процентов выполнения;
- о завершении особо длительных операций с данными следует уведомить оператора текстовым сообщением и звуковым оповещением.

Безусловно, представленный список можно и продолжить, например много полезных решений можно найти в работах [11, 12, 26]. Однако если в своем проекте клиентского приложения БД вы реализуете хотя бы представленные выше рекомендации, интерфейс вашего программного продукта избежит многих нареканий.

Резюме

Этап разработки БД и клиентских приложений завершается обязательным тестированием, во время тестирования специалисты проверяют корректность, полноту и качество разработанного программного обеспечения.

В самом общем случае тестирование разделяют на структурное (тестирование «белого ящика») и функциональное (тестирование «черного ящика»). При проведении тестирования белого ящика разработчики тестов имеют полный доступ к исходным кодам – это необходимое условие для как можно более глубокого исследования управляющей структуры программы. При осуществлении функционального тестирования программный продукт рассматривается как черный ящик, заглянуть в который не представляется возможным. На этот раз разработчики тес-

тов стараются осуществить полную проверку всех функциональных требований к программе, симулируя работу конечного пользователя.

Базаданных и клиентские приложения могут быть переданы заказчику и введены в эксплуатацию только после успешного проведения полного цикла тестов и устранения всех выявленных дефектов.

Вопросы для самопроверки

1. Какие языки программирования и среды разработки могут использоваться для разработки клиентских приложений БД?
2. Охарактеризуйте технологии доступа к данным:
 - a) ODBC;
 - b) ADO .NET;
 - c) FireDAC;
 - d) JDBC.
3. Сравните известные вам технологии доступа к данным, в каких случаях, на ваш взгляд, надо отдавать предпочтение той или иной технологии?
4. Какие критерии качества пользовательского интерфейса вам известны?
5. Каких рекомендаций следует придерживаться программистам при проектировании пользовательского интерфейса?

Глава 22

Распределенные БД

До сих пор при обсуждении систем баз данных мы не раз говорили о преимуществах централизованного хранения данных на отдельном сервере. Такой подход в союзе с архитектурным решением клиент-сервер предоставляет нам ряд существенных удобств [19]:

- возможность совместного доступа к данным;
- устранение избыточности данных;
- снижение противоречивости и поддержка целостности данных;
- упрощение организации защиты данных;
- относительная простота осуществления транзакций.

Каждое из перечисленных преимуществ достаточно очевидно и вряд ли нуждается в дополнительных комментариях. Однако всегда ли жесткая централизация выступает наиболее рациональным решением при построении информационной системы предприятия? Настало время критически переосмыслить это утверждение.

Предпосылки децентрализации

В период бурного развития реляционной модели в 1970–1980-е годы централизация данных предприятия на одном сервере рассматривалась как естественное решение, которое вряд ли вызывало критические отзывы от эксплуатирующего БД персонала. Но в конце 1980-х по всему миру наметились серьезные изменения в работе крупных коммерческих компаний, международных организаций и органов государственного управления. Причин тому несколько, приведем наиболее существенные из них.

1. Процессы глобализации привели к необходимости географической децентрализации коммерческих и управленческих опера-

ций. Например, если ваша авиакомпания осуществляет транспортные и пассажирские перевозки по всему миру, то она, скорее всего, нуждается в распределенных хранилищах данных. Ведь для покупки билета из пункта «А» в пункт «В» как-то странно обращаться в пункт «С», в котором располагается единственный сервер штаб-квартиры компании. Тем более что проблему может усугубить загрузка сервера обработкой многочисленных транзакций от разлетевшихся по всему свету тысяч пассажиров, которые в этот же момент времени изучают расписание полетов и приобретают билеты. В подобной ситуации более логичным окажется совершить все операции по оформлению перелета прямо в пункте вылета.

2. Быстрое развитие технологий удешевило стоимость компьютеров, ускорило процесс построения локальных сетей с выходом в интернет (при этом в разы повысив скорость обмена данными).
3. Применяемый в корпоративных сетях предприятий централизованный подход принципиально не противоречил идее децентрализации, надо было просто найти решение, позволяющее строить распределенные БД на фундаменте уже функционирующих централизованных хранилищ.
4. Наконец, все сказанное в пп. 1–3 привело к возрастанию требований пользователей к предоставляемым им сервисам.

Чтобы дополнить картину, отметим ряд недостатков, свойственных для централизованных систем:

- 1) высокая вероятность отказа в обслуживании при перегрузке СУБД;
- 2) относительно невысокая надежность, свойственная единственному серверу.

Таким образом, с одной стороны, все возрастающие требования бизнеса и управленческих структур, а с другой – ограничения централизованных систем хранения и обработки данных к началу 1990-х г. привели к возникновению противоречия между потребностями пользователей в децентрализации хранилищ данных и возможностями существующих БД. Указанное противоречие можно устранить только за счет децентрализации данных – разработчикам следовало научиться распределять данные по нескольким серверам (или, как часто говорят, сайтам), так чтобы БД в буквальном смысле оказались ближе к пользователям.

Определение

Распределенная база данных (distributed database) представляет собой набор логически связанных между собой данных, которые физически распределены в некоторой компьютерной сети.

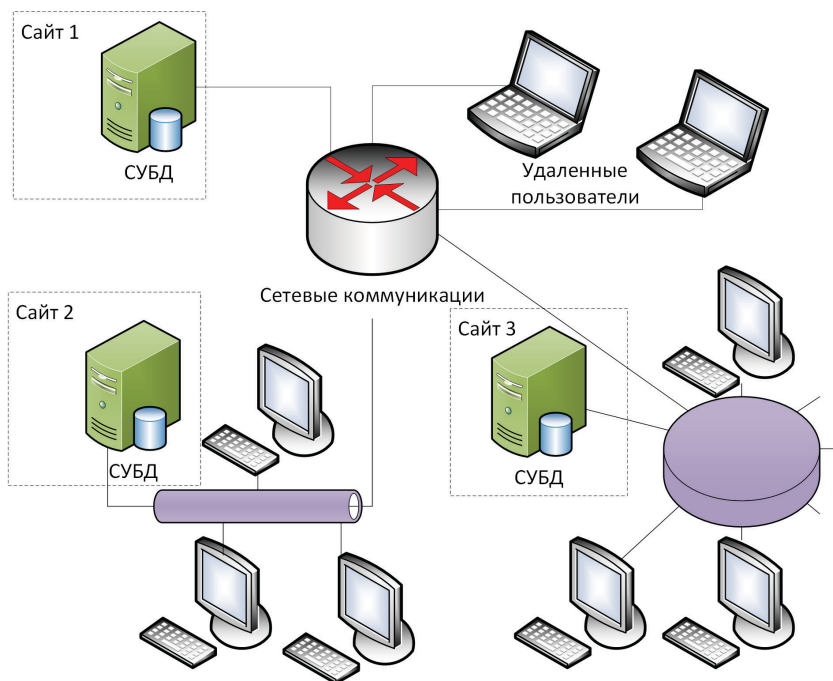


Рис. 22.1. Вариант конфигурации распределенной БД

Система управления распределенной базой данных

Что такое система управления распределенной базой данных (СУРБД)? Начнем с того, что, изучив даже самый полный прейскурант программного обеспечения, вы не обнаружите программный продукт с названием «СУРБД». Такого, готового к немедленному применению программного обеспечения в природе пока не существует, и в обозримом будущем вряд ли появится. На самом деле любая СУРБД представляет собой сложную систему, включающую в себя:

- 1) две и более СУБД, под управлением которых находятся соответствующие фрагменты БД;
- 2) локальные клиентские приложения, не требующие доступа к данным на других сайтах;

- 3) глобальные клиентские приложения, имеющие доступ к данным на других сайтах.

Одним словом, СУРБД – это по-настоящему сложный программный продукт, создаваемый высокопрофессиональными разработчиками специально для индивидуальных заказчиков.

Определение

Система управления распределенными базами данных (СУРБД) – программный комплекс, предназначенный для управления единой логической базой данных, разделенной на некоторое количество фрагментов.

Считается, что в составе распределенной СУБД должно находиться четыре следующих компонента [25]:

- 1) стандартная СУБД, предназначенная для управления локальными данными на сайте;
- 2) программное обеспечение, предназначенное для организации сетевого взаимодействия сайтов между собой;
- 3) глобальный системный каталог, содержащий служебную информацию о параметрах, специфичных для распределенной системы;
- 4) собственно СУРБД, выступающая управляющим элементом над всеми перечисленными выше компонентами.

СУРБД должна поддерживать весь функционал, присущий традиционным СУБД (см. главу 2), и, кроме того, обеспечивать выполнение специфичных для распределенных систем функций:

- 1) установка и поддержка соединений с удаленными сайтами;
- 2) обработка и оптимизация распределенных запросов;
- 3) поддержка параллельной обработки данных с выполнением требований ACID для транзакций (см. главу 16);
- 4) поддержка в актуальном состоянии глобального сетевого каталога;
- 5) обеспечение восстановления данных в условиях вероятного отказа некоторых сайтов.

Правила распределенных БД от Криса Дейта

В конце 1980-х годов Крис Дейт сформулировал фундаментальный принцип распределенной базы данных – для пользователя распределенная система должна выглядеть точно так же, как нераспределенная

(позднее этот принцип получил название «нулевое правило»). Затем Дейт опубликовал 12 правил, конкретизирующих требования к распределенным БД [19]:

- 1) локальная независимость (автономность), все операции на локальном сайте должны контролироваться этим сайтом;
- 2) независимость от центрального узла, т. е. локальные сайты должны рассматриваться как равноценные;
- 3) непрерывное функционирование, предполагает высокую степень надежности и доступности программного и аппаратного обеспечения сайтов;
- 4) независимость от расположения, пользователь не должен задумываться, где физически находятся интересующие его данные;
- 5) независимость от фрагментации (прозрачность фрагментации), система обеспечивает независимость данных от их фрагментации и обладает механизмом сборки данных из фрагментов;
- 6) независимость от репликации (прозрачность репликации). Для пользователей должно быть создано такое окружение, чтобы они, по крайней мере на логическом уровне, полагали, что в действительности данные не дублируются;
- 7) обработка распределенных запросов, запросы могут охватывать данные, расположенные на нескольких сайтах;
- 8) управление распределенными транзакциями, включает решение двух фундаментальных для всех БД задач: управление параллельностью и управление восстановлением. И это все с учетом дополнительного коэффициента сложности из-за распределенного характера БД;
- 9) независимость от аппаратного обеспечения;
- 10) независимость от операционной системы;
- 11) независимость от сети;
- 12) независимость от типа СУБД, требует, чтобы все СУБД поддерживали один и тот же интерфейс.

Как видите, все 12 правил нацелены на обеспечение выполнения нулевого правила распределенных БД.

Аспекты проектирования распределенных БД

Наверное, не стоит повторяться, что разработка распределенных БД является делом подготовленных специалистов, имеющих знания, навы-

ки и умения проектирования классических клиент-серверных систем и представляющих особенности функционирования децентрализованных систем. Вместе с тем надо понимать, что даже самый совершенный гуру в области разработки информационных систем когда-то тоже был обычным неопытным новичком, совершающим множество своих «непоправимых» ошибок на пути к созданию первых распределенных БД. Нам с вами окажется весьма полезным изучить опыт подобных первопроходцев, тем более что на сегодня он неплохо систематизирован. В первую очередь стоит выделить три аспекта, характерных для распределенных систем [25]:

- 1) фрагментация;
- 2) распределение;
- 3) репликация.

Перечисленные аспекты должны учитываться разработчиком еще на самых начальных этапах процесса проектирования распределенных систем.

Фрагментация

Любая база данных или реляционное отношение, подлежащее хранению в децентрализованной системе, может разделяться на некоторое число фрагментов и распределяться по сайтам системы. Каждый из сайтов должен иметь доступ к глобальной схеме разделения, чтобы правильно сформировать запрос и при необходимости правильно собрать фрагменты в единое целое. Для этого информация о фрагментации передается в отдельный каталог распределенных данных (distributed data catalog, DDC).

Наиболее простой случай фрагментации предполагает, что таблицы БД размещаются по тем сайтам, на которых они окажутся наиболее востребованными. Такой подход не предполагает какие-либо действия по разделению таблиц на фрагменты – они просто целиком располагаются «поближе» к месту использования. Подобная фрагментация БД является тривиальной, в подобных случаях, скорее, стоит говорить о разбиении (а не о фрагментации) одной БД на несколько.

Более интересен подход, когда разделению на фрагменты подвергаются сами таблицы. В таком случае специалисты выделяют три подхода (рис. 22.2):

- 1) горизонтальная фрагментация. Отношение фрагментируется покортежно, т. е. подмножества строк одной таблицы разделяются

- между несколькими сайтами. Соединение фрагментированной по горизонтали таблицы может осуществляться за счет предусмотренной стандартом SQL операции слияния UNION (см. главу 11);
- 2) вертикальная фрагментация. Отношение разделяется на подмножества столбцов и распределяется по сайтам, при этом, как правило, столбец(ы) первичного ключа включае(ю)тся в каждый из фрагментов;
 - 3) смешанная фрагментация, представляющая комбинацию горизонтальной и вертикальной фрагментаций.

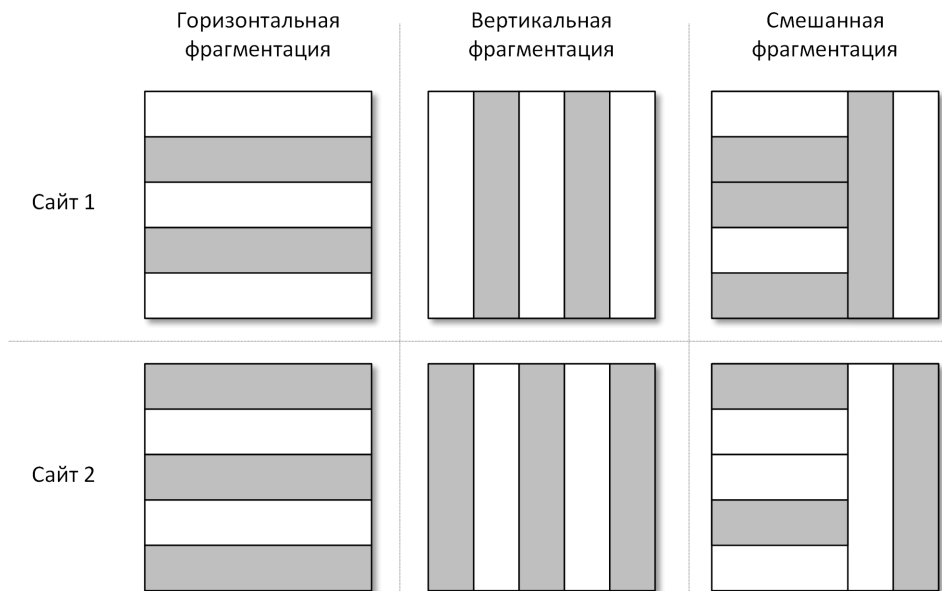


Рис. 22.2. Разделение отношения на фрагменты

При фрагментации отношений разработчик распределенной БД должен руководствоваться рядом критериев:

- 1) полнота – данные подверженной фрагментации таблицы не должны быть утеряны, т. е. все элементы данных таблицы обязаны присутствовать в каком-то из фрагментов;
- 2) восстановимость – должна иметься возможность восстановления отношения из фрагментов;
- 3) в случае горизонтальной фрагментации соблюдается критерий непересекаемости, исключающий дублирование одного и того же элемента данных в двух и более фрагментах. При вертикальной фрагментации непересекаемость обеспечить невозможно из-за

необходимости хранения на сайтах повторяющихся атрибутов первичных и внешних ключей, иначе фрагменты кортежей просто не соберутся в отношение.

Распределение

Для того чтобы обеспечить условия для эффективной работы СУРБД, необходимо рационально распределить фрагменты по сайтам. Как правило, распределение напрямую связано с особенностями использования хранящихся в БД данных. Например, если мы говорим о крупном предприятии, обладающем множеством отделов и служб, то логичнее всего будет в домене, принадлежащем отделу кадров, расположить сайт с фрагментами персональных дел сотрудников. Эмпирически доказано, что 80 % нагрузки на этот сайт будет создавать именно отдел кадров и лишь 20 % составят внешние запросы. Это же правило «80/20» относится ко всем остальным потребителям услуг распределенной БД.

Репликация

Репликация (replication) представляет собой механизм синхронизации содержимого нескольких копий данных.

Желательность применения механизма репликации в распределенных системах обусловлена двумя факторами. Во-первых, наличие на сайтах локальной копии данных позволяет повысить производительность при обработке данных. Во-вторых, увеличится доступность данных за счет наличия дополнительных реплик.

Всегда надо помнить о главном недостатке репликации – проблеме обновлений, суть ее заключается в том, что если реплицируемый объект обновился, то все его копии должны быть обновлены.

Предусмотрено несколько подходов организации репликации данных в распределенных БД.

1. Полная репликация обеспечивает размещение полной копии БД на всех сайтах системы. Это дорогостоящее решение, затраты которого в первую очередь связаны с необходимостью закупки устройств хранения данных и оплаты высокого сетевого трафика. Зато в результате мы получаем высоконадежную систему со всегда доступными данными.
2. Выборочная репликация предполагает размещение на сайтах копий только наиболее востребованных (но редко модифицируемых) фрагментов данных. В этом случае стоимость хранения и затраты на передачу данных существенно снижаются, но за это

мы расплачиваемся некоторым ухудшением характеристик надежности и доступности.

3. Отказ от репликации приводит к тому, что БД разделяется на непересекающиеся фрагменты. По себестоимости это самое дешевое решение, однако в таком случае надежность и доступность данных оказываются на самом низком уровне.

Особенности управления системным каталогом

Неотъемлемой частью любой БД выступает системный каталог, содержащий необходимые для работы метаданные. В обычных централизованных клиент-серверных системах в каталоге располагаются сведения о типах данных, ограничениях целостности, представлениях и хранимых процедурах, зарегистрированных пользователях и их правах и т. п. В распределенных базах данных, в дополнение к традиционным метаданным, в системный каталог попадает дополнительная управляющая информация о фрагментации, распределении, репликации и т. д. Разработчикам распределенной БД остается только ответить на вопрос об организации доступа к метаданным системного каталога для каждого из сайтов.

К настоящему времени в РБД сложилось три базовых подхода к размещению системного каталога:

- 1) централизованное размещение, предполагающее, что системный каталог существует в единственном экземпляре и физически хранится на центральном сайте;
- 2) полная репликация каталога, что обеспечивает поддержку актуальных копий каталогов на всех сайтах системы;
- 3) поддержка каждым сайтом своего собственного каталога, описывающего только БД этого сайта.

Не исключается комбинирование представленных выше подходов, например сайты обладают собственными каталогами, и, помимо этого, центральный сайт ведет общий системный каталог, объединяющий управляющую информацию со всех сайтов.

Распределенные транзакции

Требования, предъявляемые к распределенным транзакциям, практически ничем не отличаются от классических требований ACID к обычным транзакциям, выполняющимся в централизованных клиент-сер-

верных БД. Однако выполнение этих требований в распределенных системах существенно сложнее. Только задумайтесь над тем, какую программную логику придется реализовать, чтобы обеспечить атомарность, согласованность, изолированность и устойчивость в ситуации, когда глобальная транзакция затронула данные, распределенные по нескольким сайтам!

Каким образом можно обеспечить целостность и непротиворечивость данных при фиксации или откате транзакции в распределенной системе? Существует несколько решений такой задачи, наиболее показательные из них:

- 1) применение централизованного протокола двухфазной блокировки. Это тот случай, когда, несмотря на то что данные распределены, менеджер управления транзакциями размещается только на одном сайте. Глобальный менеджер управляет работой локальных планировщиков транзакций, которые отвечают за корректное выполнение транзакций на своих СУБД и при этом используют обычные правила протокола двухфазной блокировки;
- 2) распределенный протокол двухфазной блокировки. При таком подходе каждый локальный менеджер транзакций отвечает за управление блокировками своих данных. При этом учитывается тот факт, что в распределенных БД наборы данных регулярно реплицируются – другими словами, существует несколько копий данных. Поэтому при обработке транзакции локальный менеджер допускает ее к определенной первичной копии данных и только после успешной фиксации изменений распространяет изменения на все остальные копии.

Дополнительные трудности, с которыми сталкиваются проектировщики распределенных БД, связаны с выявлением и устранением распределенных взаимных блокировок. Сложность выявления факта блокировки повышается, если СУРБД не работает в режиме централизованного протокола двухфазной блокировки, ведь теперь допускается, что две и более глобальные транзакции формируются и сопровождаются с разных сайтов распределенной БД.

Преимущества распределенных БД

Системы, функционирующие на фундаменте распределенной базы данных, по сравнению с традиционными централизованными системами, имеют дополнительные преимущества:

- соответствие структуре организации. Наиболее востребованные данные располагаются рядом с потребителем – на локальном сайте, что позволяет получать доступ к своим данным без обращений к другим сайтам;
- повышение доступности данных. В большинстве случаев пользователи работают только с небольшим подмножеством данных БД. Благодаря децентрализации это подмножество может храниться максимально близко к потребителю, зачастую даже локально благодаря механизму репликации;
- хорошая масштабируемость системы. Сетевая распределенная структура БД предполагает простой механизм добавления к системе как новых сайтов с данными, так и новых пользователей без снижения производительности;
- повышение производительности. Благодаря тому что данные хранятся на разных сайтах, в обработке запросов участвует не один, а несколько серверов, что благотворно сказывается на повышении скорости доступа к данным;
- повышение надежности. Централизованная система по определению более уязвима, по сравнению с распределенной. Если же в распределенной БД будет реализован механизм репликации, создающий копии данных на нескольких сайтах, то уровень надежности становится едва ли не абсолютным;
- локальная автономность. Если сайт обладает полной копией данных, то он получает локальную автономность в работе и не столь критично переносит сбои в доступе к сети организации.

Недостатки распределенных БД

В свое время профессором Калифорнийского университета Эриком Брюером была предложена теорема CAP, название которой сложилось из заглавных букв трех терминов: согласованность (consistency), доступность (availability) и устойчивость к разделению (partition tolerance).

В максимально сжатой форме суть теоремы CAP можно свести к следующему: «Невозможно одновременно обеспечить следующие три свойства распределенной системы – согласованность, доступность и устойчивость к разделению сети» [55]. Таким образом, теорема CAP выступает своеобразным камнем преткновения на пути развития всех распределенных систем и в том числе децентрализованных БД. Хотите получить распределенную БД? Пожалуйста! Но только выберите любые два качества из аббревиатуры CAP, а третье, увы, окажется недоступным...

Еще одна проблема распределенных БД связана с отсутствием общепринятого стандарта на подобные системы. И хотя названная проблема известна специалистам с конца 1980-х годов, но пока не нашла своего решения. Масло в огонь подливает еще тот факт, что на сегодняшний день фактически отсутствует универсальная методология преобразования централизованных БД в распределенные. А раз нет методологии, то нет и соответствующих ей инструментальных средств, упрощающих работу разработчиков децентрализованных БД, администраторов данных, программистов и прочих специалистов.

Помимо собственно проблемы САР и отсутствия стандарта, распределенные БД обладают целым букетом дополнительных недостатков. Среди них:

- высокая сложность системы. Для того чтобы реализовать распределенную БД и при этом сохранить прозрачность системы для конечного пользователя, придется изрядно потрудиться и создать программный комплекс, сложность которого в разы превосходит сложность традиционной централизованной клиент-серверной системы;
- трудности в разработке системы. Порядок проектирования распределенных БД не стандартизирован, так что программистам во многом приходится решать поставленные перед ними задачи на основе опыта и интуиции, зачастую методом проб и ошибок;
- высокая стоимость системы. Стоимость системы напрямую зависит от ее сложности. Отметим, что сложность определяется не только высокой ценой на ее разработку, но и тратами на аппаратную составляющую, затратами на сопровождение, документирование, администрирование и т. д.

Таким образом, современная распределенная БД представляет собой штучный товар, создать который под силу только высококлассным специалистам. Но если эта задача окажется решенной, то вы получите все преимущества децентрализованной системы.

Резюме

Внедрение распределенной базы данных, соответствующей организационной структуре предприятия, расширяет возможности совместного использования информации, так как позволяет сделать данные, поддерживаемые каждым из подразделений, общедоступными, обеспечив

при этом их сохранение именно в тех местах, где они чаще всего используются.

По сравнению с централизованными клиент-серверными системами, распределенные БД позволяют повысить надежность, доступность и производительность системы, но, с другой стороны, за это приходится расплачиваться высокой сложностью системы.

С точки зрения конечного пользователя, распределенные БД должны выглядеть точно так же, как и общепринятые централизованные. Такой результат достигается за счет обеспечения прозрачности распределения данных и прозрачности транзакций.

Вопросы для самопроверки

1. Что послужило причинами создания распределенных БД?
2. Прокомментируйте правила распределенных БД от Криса Дейта.
3. Поясните смысл теоремы CAP.
4. Для чего нужна фрагментация отношений?
5. Какие схемы фрагментации вам известны?
6. Что понимается под термином «распределение», и как на распределение влияет правило «80/20»?
7. Как может быть организована репликация в распределенных БД?
8. Какие особенности системного каталога свойственны распределенным БД?
9. Что понимается под прозрачностью распределенной БД для конечного пользователя?
10. Прокомментируйте преимущества и недостатки распределенных БД.

Глава 23

Объектно-ориентированная модель данных

Как бы это не показалось удивительным, но уже в середине 1980-х (в самый расцвет реляционной парадигмы) стали появляться утверждения, что король если пока и не умер, то вот-вот отойдет в мир иной, и на смену реляционной модели придет новая модная концепция объектно-ориентированной модели данных (object-oriented data model, OODM)¹.

В свою очередь, на базе OODM будет воздвигнута конструкция объектно-ориентированных баз данных (ООБД), которые станут работать под управлением объектно-ориентированных СУБД (ООСУБД). В свою очередь, объектно-ориентированные СУБД будут представлять собой симбиоз системы программирования и СУБД, основанной на объектно-ориентированной модели данных.

Замечание

Для того чтобы читатель всегда оставался в курсе последних достижений в области OODM, стоит регулярно просматривать сайт www.odbms.org (расшифровывается как Operational Database Management Systems). Данный портал посвящен самым разнообразным современным технологиям хранения и обработки данных, в том числе OODM.

С тех времен минуло больше тридцати лет. То, что реляционные СУБД до сих пор находятся в добром здравии, читателю уже хорошо известно, разберемся, что стало с ООБД.

Предпосылки появления модели

Разговоры о необходимости разработки нового направления OODM возникли не на пустом месте. К середине 1980-х разработчики инфор-

¹ Вместо термина OODM иногда используется его синоним ODM (object data model).

мационных систем практически полностью перешли на реляционную модель, а опыт эксплуатации реляционных БД позволил выявить не только положительные, но и, увы, отрицательные качества детища доктора Кодда.

Преимущества реляционной модели мы обсуждали практически на протяжении всей книги, теперь настал час перейти к критике. Суть ее в следующем: при проектировании сложных информационных прикладных систем (автоматизированное проектирование, автоматизированное производство, автоматизированная разработка программного обеспечения, мультимедиа-системы, издательское дело, геоинформационные системы, медицинские системы и т. д.) разработчики столкнулись с рядом препятствий, обойти которые оказалось очень сложно или вообще невозможно.

Анализ ключевых недостатков реляционной модели осуществлялся во многих исследованиях [19, 25, 38]:

- **упрощенное представление сущностей.** Подлежащие учету сведения о той или иной сущности всего-навсего представляют собой кортеж из нескольких атрибутов, что далеко не всегда обеспечивает полноценное описание. Положение дел усугубляется, когда в результате нормализации сущность фрагментируется по нескольким отношениям. В таком случае для сборки сущности в единое целое приходится осуществлять операции соединений. Как следствие снижается скорость доступа к данным;
- **ограниченность семантики.** В арсенале разработчика реляционной БД не предусмотрено никаких других способов описания данных и связей между ними, кроме отношения. Между отношениями можно организовать всего два вида связи: 1:1 и 1:M, а для создания связи M:N приходится создавать дополнительное отношение. Таких ограниченных возможностей явно недостаточно для полноценного представления сущностей реального мира;
- **слабая поддержка ограничений целостности и корпоративных ограничений.** И действительно, реляционная модель стоит на страже только целостности сущностей и ссылочной целостности. Все остальное должно реализовываться вручную разработчиками БД;
- **однородность данных.** Двухмерные плоские таблицы, составляющие базис реляционной модели, однородны как по вертикали (однотипные атрибуты столбца), так и по горизонтали (идентичный перечень атрибутов для всех кортежей). Подобное ограни-

чение зачастую мешает адекватному представлению сущностей. Хотя нельзя не отметить, что однородность данных можно рассматривать и как преимущество реляционной модели, существенно упрощающее обработку данных;

- **ограниченный набор операций.** С задачей полноценного моделирования объектов реального мира сложно справиться за счет относительно небольшого набора операций, предусмотренных теорией множеств и впоследствии описанных в рамках первых версий стандарта SQL².

Преимущества ООБД

Чего ожидают от объектно-ориентированных баз данных пользователи? Безусловно, устранения недостатков реляционной модели и как минимум получения определенных преимуществ, среди них:

- **естественное представление объектов реального мира.** Это вполне реализуемо, так как объектно-ориентированная концепция обладает большей полнотой и способна работать с подробной семантической информацией. Очень важно то, что, в отличие от традиционных БД, которые просто хранили данные, перспективные ООБД позволяют говорить о появлении у объектов поведенческого аспекта. Таким образом, объекты (в отличие от отношений) получают возможность реагировать на окружающую обстановку;
- **работа со сложными объектами в прикладных информационных системах.** По сравнению с реляционной моделью, объектно-ориентированный подход, безусловно, более удобен при работе со специализированными областями знаний (автоматизация производства, наука, медицина и т. д.), ведь на этот раз вместо плоских таблиц в распоряжение программистов предоставляются классы и сконструированные на их основе объекты с их атрибутами и методами. Классы и объекты могут быть сколь угодно сложными, порог сложности ограничивается лишь уровнем воображения заказчиков БД и квалификацией программистов;
- **расширение базовых типов данных.** Возможность создания широчайшего спектра пользовательских типов данных и классов, безусловно, увеличивает функциональные возможности ООБД;

² Стоит отметить, что с выходом стандарта SQL:1999 возможности реляционной модели по работе с объектами существенно расширились.

- **многократное использование классов** упрощает и ускоряет разработку БД, что повышает технологичность в целом ООБД;
- **улучшенная интеграция с другими СУБД** в единую структуру.

Замечание

Объекты в объектно-ориентированных БД по определению находятся на более высоком уровне абстракции, чем таблица в реляционной модели.

Объектно-ориентированная терминология

Во многом ответ на вопрос «Что такое ООБД?» дает сам термин «объект». Объект – это очень широкое понятие. В окружающем нас мире объект – это то, к чему можно прикоснуться руками: это клавиша на клавиатуре, сама клавиатура, настольная лампа, спящая под ней кошка и проезжающий за окном автомобиль. У объектов имеются характеристики, взглянув на которые, мы можем судить об их состоянии. Лампа может быть включена или выключена, клавиша нажата или отпущена, автомобиль может двигаться по дороге со скоростью 60 км/ч, а может, как кошка, мирно дремать, но на этот раз не под лампой, а на стоянке или в гараже.

Объекты, с которыми привыкли иметь дело программисты, сильно отличаются от объектов реального мира. Но, впрочем, в обеспечении абсолютной идентичности и нет смысла. Полагаю, что никто из читателей и не рассчитывал повернуть ключ зажигания в автомобиле, «собранный» в ООБД, и уехать на нем на прогулку. Программные объекты призваны лишь моделировать физические объекты, а для этого вместо шестеренок, винтиков и болтиков они должны обладать атрибутами (полями) и методами (рис. 23.1).

Напрямую доступ к своим атрибутам объект обычно не предоставляет. Вместо этого у него имеются посредники: функции (которые в объектно-ориентированных технологиях обычно называются **методами** (methods)). Настольной лампой управляют два ключевых метода: включить и выключить. У автомобиля этих методов значительно больше: он заводится, набирает скорость, изменяет направление движения, замедляет движение. В первую очередь методы воздействуют на атрибуты объекта, так, практически все методы автомобиля сосредоточены вокруг полей, описывающих его скорость и направление движения.

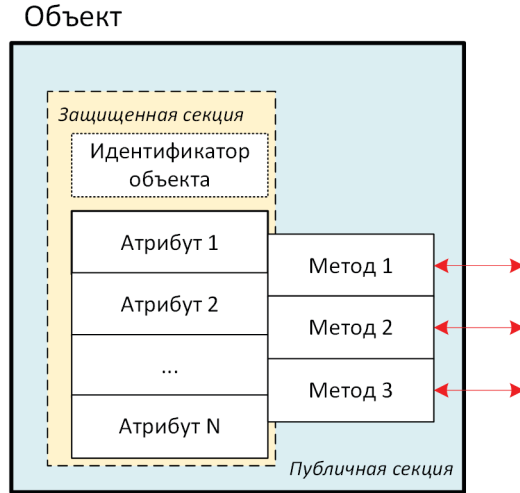


Рис. 23.1. Строение объекта

Замечание

Объекты одного класса отличаются друг от друга только значениями атрибутов, а их методы одинаковы. Поэтому в ООБД методы объектов одного класса достаточно сохранить лишь однажды, а затем предоставить к ним доступ всем экземплярам класса.

Как правило, объект создается многослойным, как минимум состоящим из двух секций: частной (private) и публичной (public). В частной секции сосредоточиваются все нуждающиеся в защите элементы объекта, доступ к ним закрыт. Публичная секция, напротив, доступна для обращения к объявленным в ней методам из вне объекта.

При изменении значений атрибутов изменяется состояние его владельца. Более того, при необходимости объект может узнать о состоянии объектов, с которыми он связан, для этого, например, можно обратиться к публичным методам интересующего нас объекта.

Объект не может быть соткан из воздуха, при создании нового объекта (или, как часто говорят, экземпляра класса) вся необходимая для этого информация берется из соответствующего ему класса. Класс – это не что иное, как чертеж проектируемого объекта, но не нарисованный карандашом, а описанный на языке программирования. В исходный код класса включается перечень атрибутов будущего объекта и управляющие ими методы.

Признанный гуру в области объектно-ориентированного проектирования Гради Буч (Grady Booch) выделяет четыре обязательных элемента, без которых идея объектно-ориентированного подхода оказывается недееспособной [16]. К обязательным ингредиентам концепции ООП относятся: абстрагирование, инкапсуляция, модульность и наследование.

Абстрагирование

Все наши приложения описывают реальные физические объекты и/или процессы, в которых участвуют эти объекты. Возьмем даже программы-игрушки, например стратегии. В них моделируется поведение полководца, гоняющего свою электронную армию по экрану монитора с целью подчинить себе весь мир. В хороших стратегиях учитываются боевые качества юнитов, толщина брони, дальнобойность и калибр артиллерии, характеристики местности и многие другие факторы, оказывающие прямое или косвенное влияние на результат боя. Чем больше мелочей смог описать разработчик, тем правдоподобнее впечатление. Но тем не менее это не реальность, а всего лишь ее абстрактное представление, удобное для игроков. Приведем другой пример: рассмотрим такой объект, как автомобиль. Это весьма сложная конструкция, включающая тысячи деталей, работающих (или нет) по сложным математическим, физическим, химическим и другим законам. Но с точки зрения начинающего водителя любой автомобиль можно абстрагировать как транспортное средство, предназначенное для перемещения грузов из точки А в точку Б. Чтобы закончить абстракцию, добавим к перечисленному марку, цвет и номерной знак над бампером. И все! Среднестатистическому водителю нет никакого дела до того, как от двигателя крутящий момент добирается до колес его механического коня.

Замечание

Абстракция выделяет ключевые характеристики некоторого объекта, отличающие его от всех других видов объектов, и отбрасывает незначительные детали. Таким образом, абстракция определяет концептуальные границы объекта с точки зрения программиста или пользователя.

Инкапсуляция

Следующим за абстрагированием столпом, на котором держится объектно-ориентированная концепция, считается инкапсуляция. Основ-

ное назначение инкапсуляции – скрытие от пользователя особенностей реализации класса.

По сути, инкапсулируемый объект можно разделить на две логические части: открытый для всех интерфейс и закрытая область, в которой скрываются логика и данные объекта. Проводя параллель с окружающим нас миром, скажу, что точно так же от нас спрятаны внутренности телевизора. Снаружи только необходимые органы управления, используя которые, мы можем найти интересующий нас канал, изменить громкость, яркость. При этом телезрителя мало интересует частота гетеродина, работа декодера PAL или уровень напряжения в блоке дежурного питания, а для многих названные мною термины сродни тарабарщине. Если что-то сломалось, то мы вызываем мастера. Он специалист по инкапсуляции и бесстрашно снимает заднюю стенку телевизора...

Уже интуитивно понятно, что благодаря инкапсуляции повышается надежность функционирования программного объекта, т. к., в отличие от четырех-пяти винтов в задней стенке телевизора, неопытный программист не сможет выкрутить винты из объекта, исповедующего объектно-ориентированные традиции. Но, помимо повышения общей надежности объекта, инкапсуляция обеспечивает программную логику его работы, а это еще более важное звено в идее объектно-ориентированного программирования.

Модульность

Модульность – это способность системы строиться на основе совокупности взаимодействующих самостоятельных элементов. Подавляющее большинство окружающих нас вещей состоит из таких элементов. Например, системный блок компьютера содержит модули процессора, памяти, видеокарты, которые, в свою очередь, также состоят из модулей (микросхем, диодов, транзисторов, резисторов и т. д.).

Поддержание модульности в программировании сродни искусству. В классическом структурном программировании нужны определенные способности, чтобы грамотно разнести используемые совместно подпрограммы по модулям, в объектно-ориентированном программировании требуется уметь разделить классы и объекты. Деление на модули значительно упрощает процесс разработки программного обеспечения: мы собираем программу из заранее подготовленных кубиков. Наивысший пилотаж – умение программиста разработать программу, в которой можно заменять один «кубик» другим без переделывания всего остального исходного кода.

Наследование

Еще одним ингредиентом объектно-ориентированного программирования считается наследование. Это совсем простое понятие. Его смысл в том, что при описании нового класса допустимо брать за основу другой, родительский класс. Таким образом, достигается преемственность поколений – дочерний класс наследует все методы и свойства своего предка. Ограничений на длину цепочки наследования, как правило, нет, так что могут создаваться классы, имеющие в родословной несколько предков. Таким образом, самый последний в иерархии наследования класс – если так можно выразиться, самый опытный: он впитал все лучшее от всех своих «дедушек» и «прадедушек».

В результате объединения усилий инкапсуляции, модульности и иерархичности появляется едва ли не самый сильный из козырей объектно-ориентированного программирования – **полиморфизм**. Дословная расшифровка этого термина обозначает обладание многими формами. Идея полиморфизма заключается в праве экземпляра некоторого класса представлять все классы из его иерархической цепочки. Более того, благодаря полиморфизму мы получаем право скрывать или перепределять поведение унаследованных методов. Поэтому различные по содержанию процедуры и функции всевозможных объектов могут использовать одно и то же имя. Вызов же метода будет приводить к выполнению кода, соответствующего конкретному экземпляру объекта.

Идентификатор объекта

В реляционной модели для однозначной идентификации кортежа отношения используется первичный ключ. Для назначения ключа создатель таблицы может пойти двумя путями – воспользоваться искусственным атрибутом (например, автоинкрементного типа) либо назначить на роль пользовательского ключа один или несколько атрибутов с реальными данными, которые гарантируют уникальность идентификации строки.

В ООБД объекты также нуждаются в идентификации, для этого каждому объекту присваивается идентификатор (Object ID), идентификатор связывается с объектом на все время его существования и не меняется. Но технология идентификации объекта гораздо сложнее, особенность заключается в том, что существует два типа идентификаторов [25]:

- 1) физические идентификаторы, в которых закодирована информация о расположении объекта на вторичном устройстве памяти (жестком диске);

- 2) логические идентификаторы, не зависящие от физического расположения объекта.

Для быстрого доступа к объекту по идентификатору в рамках ООБД существует механизм подстановки указателей (pointer swizzling), осуществляющий преобразование между физическим и логическим адресами.

Внимание!

Между указателями в ООБД и указателями объектно-ориентированного языка программирования есть очень важное отличие. Указатели на объекты в приложении, написанном на языке программирования, хранят адрес объекта в адресном пространстве процесса программы, такие указатели действительны только во время выполнения программы. При перезапуске программы значения адресов объектов обязательно изменятся. У объектов в ООБД указатель действителен на всем временном интервале существования объекта, даже если последний не загружен в память.

Благодаря принятому в ООБД способу идентификации объектов мы получаем важное преимущество – объект может ссылаться на неограниченное количество объектов и принадлежать нескольким объектам одновременно.

Манифест объектно-ориентированных СУБД

К началу 1990-х реляционная модель окончательно утвердилась в качестве базовой модели для большинства СУБД, во многом этому способствовал выход стандарта для языка SQL (см. главу 11). Понимая важность стандартизации, разработчики перспективной объектно-ориентированной модели постарались сделать первый шаг в этом направлении и в 1989 году сформулировали манифест объектно-ориентированных СУБД [3].

Манифест ООСУБД включает 13 обязательных требований. Обратите внимание на то, что соблюдение первых 8 правил позволит считать систему объектно-ориентированной, а выполнение правил 9–13 – считать систему СУБД.

1. Поддержка составных объектов, которые могут создаваться путем применения конструкторов из простых типовых объектов, таких как целые числа, вещественные числа, символы, булевы типы

данных и т. п. Как минимум ООСУБД должна обладать функционалом по созданию множеств, строк и кортежей. Более развитые системы должны поддерживать массивы и списки.

2. Поддержка идентичности объектов за счет применения уникальных идентификаторов. Таким образом, даже у абсолютно эквивалентных объектов должны иметься разные идентификаторы. Поддержка идентифицируемости объектов необходима не только для обеспечения уникальности, но и для успешной реализации операций присваивания, копирования, проверки идентичности и равенства объектов.
3. Поддержка инкапсуляции как краеугольного столпа объектно-ориентированной парадигмы. В случае ООБД инкапсуляция крайне важна для обеспечения модульности (для конструирования сложных объектов) и для поддержки логической независимости данных (мы можем изменить реализацию класса на нижних слоях системы, не принуждая к изменению использующие этот класс более высокоуровневые программы).
4. Поддержка типов или классов. В большинстве объектно-ориентированных языков программирования различие между типом и классом мало заметно. Однако разработчики манифеста решили акцентировать внимание будущих создателей ООСУБД на том, что здесь это не тождественные понятия.
В ООБД тип соответствует понятию абстрактного типа данных в высокоуровневых языках программирования. Концептуально тип состоит из двух частей: доступной для пользователей интерфейсной части и защищенной части, ответственной за реализацию интерфейса. Таким образом, понятие типа актуально во время компиляции.
Спецификация класса может ничем не отличаться от спецификации типа, разница между классом и типом раскрывается во время выполнения. Для создания экземпляра класса используется фабрика объектов (object factory), а после создания объект помещается на склад объектов (object warehouse). Пользователь может воздействовать на склад, применяя операции ко всем экземплярам класса.
5. Поддержка наследования типов или классов за счет использования перегрузки. Дочерний тип (подтип) и дочерний класс (подкласс) должны обладать возможностью наследовать поля и методы от супертипа или суперкласса соответственно.

6. Перекрытие, перегрузка и позднее связывание. Программистам, знакомым с любым из современных объектно-ориентированных языков программирования, очень хорошо знакомы эти термины – они обеспечивают полиморфизм объекта.

Благодаря полиморфизму (и наследованию) реализуется право объекта представлять все классы из его иерархической цепочки. Например, перекрытие (override) позволяет скрывать унаследованные методы, переопределяя метод предка в дочернем классе. При этом метод сохраняет свое прежнее имя.

Перегрузка (overload) также применяется для создания одноименных методов, но на этот раз принадлежащих одному и тому же классу. Перегружаемые методы должны обладать разным набором атрибутов и разной реализацией.

Для поддержания полиморфизма имена методов должны транслироваться в их адреса во время выполнения – это называется поздним связыванием (late binding). Это означает, что адреса методов не могут быть статическими, а напротив, допускают возможность изменения. Именно динамические адреса позволяют наследникам изменить поведение метода.

7. Язык ООБД должен обладать вычислительной полнотой, т. е. это должен быть язык программирования общего назначения (C++, C#, Delphi), а не применяемый в реляционной модели структурированный язык запросов SQL.
8. Расширяемость набора типов данных. В стандартной поставке ООСУБД должен существовать набор предопределенных типов данных и механизм для определения новых типов.
9. Поддержка перманентности данных. Перманентность (persistence) означает наличие механизма сохранения данных после завершения выполнения процесса с возможностью их последующего использования в другом процессе. Перманентность должна поддерживаться для всех объектов без исключения (требование ортогональности) и быть неявной (т. е. пользователь не должен явным образом вызывать операцию сохранения объекта).
10. Поддержка больших данных. Система должна легко справляться с очень большими по размерам данными (например, мультимедиа). Для этой цели ООСУБД должна обладать возможностью эффективного управления вторичной памятью.

11. Параллелизм. Поддержка одновременной совместной работы многих пользователей.
12. Восстановление. Восстановление после сбоев аппаратного и программного обеспечения.
13. Средства обеспечения незапланированных запросов. Предоставление простых способов создания запросов к данным. При этом средство построения запросов обязано соответствовать следующим критериям: оно должно быть высокоуровневым и в достаточной степени декларативным (пользователь указывает, что надо сделать, а не как это сделать); должен существовать механизм оптимизации запросов, чтобы последний выполнялся максимально эффективно; средство запросов не должно зависеть от приложений.

В манифесте предусматривается еще ряд опционных возможностей: множественное наследование, распределенная обработка в сети, проектные транзакции, механизм поддержки версий и т. п. Однако удивляет тот факт, что разработчики манифеста практически не упоминают такие краеугольные камни систем баз данных, как безопасность и целостность данных.

Замечание

Надо знать, что манифест систем объектно-ориентированных баз данных не был единственным манифестом СУБД. Спустя буквально год появился второй манифест. По своей сути он стал реакцией сообщества разработчиков SQL-ориентированных баз данных на первый манифест. Во втором манифесте рассматривались характеристики перспективных СУБД третьего поколения (напомним, что речь идет о начале 1990-х) – так называемых объектно-реляционных. Что очень важно, второй манифест имел как теоретические, так и практические последствия, в частности в середине 90-х вышли две претендующие на звание объектно-реляционных СУБД (Informix Universal Server и Oracle 8), кроме того, объектные расширения языка SQL были зафиксированы в стандарте SQL:1999.

В 1995 г. вышел очередной – третий манифест (его авторами являются Хью Дарвен и Кристофер Дейт). Если постараться кратко выразить суть третьего манифеста, то можно сказать следующее: смерть реляционной модели сильно преувеличена. Ключевой идеей третьего манифеста является то, что изложенные в первых двух манифестах объектно-ориентированные идеи легко реализуются в рамках реляционной модели, просто они должны быть полностью ортогональны реляционной модели. Как говорится, круг замкнулся...

Стандарт ODMG

С целью разработки стандарта для объектно-ориентированных СУБД в 1991 году был образован консорциум (Object Data Management Group, ODMG). За время существования ODMG выпустила три стандарта, последний из них ODMG 3.0 был опубликован 2000–2001 годах [5].

В стандарте ODMG 3.0 можно выделить следующие элементы:

- объектная модель Object Management Group's Object Model;
- языки спецификаций объектов. Язык определения объектов (Object Definition Language, ODG) использовался для определения типов объектов, соответствующих объектной модели ODMG. Формат обмена объектами (Object Interchange Format, OIFG) использовался для сброса и загрузки текущего состояния в файл (набор файлов) или из него;
- объектный язык запросов (Object Query Language, OQL). OQL строился на основе SQL, и по аналогии со своим «страшим братом» SQL объектный язык запросов стал декларативным (непроцедурным). Основное отличие OQL от SQL заключается в том, что объектный язык обладает куда более существенными объектно-ориентированными возможностями;
- описаны особенности взаимодействия объектной модели и языков C++, Java и Smalltalk.

В середине 2000-х предпринимались попытки возобновить работу над 4-й версией стандарта ODMG, однако существенного результата они пока не принесли.

Что было сделано на практике?

Теоретическое обоснование объектно-ориентированной модели данных крайне важно на этапе разработки концепции перспективных ООСУБД. Но любая теория должна подкрепляться практикой, иначе утрачивается смысл научных изысканий. Настал черед познакомиться с наиболее известными СУБД, попытавшимися на рубеже 1980–1990-х годов воплотить в жизнь объектно-ориентированную идею работы с данными.

Postgres

Существенный шаг навстречу ОО-концепции был сделан уже в конце 1980-х при разработке СУБД Postgres (сегодня широко известны на-

следники этой системы – бесплатная СУБД PostgreSQL и коммерческая СУБД Informix Universal Server). Возглавлявший проект Майкл Стоунбрейкер (Michael Stonebraker) реализовал одно из существенных требований ООБД – сделал так, что в столбцах отношений Postgres стало возможным хранить определяемые пользователями типы данных.

Подчеркнем, что Майкл Стоунбрейкер не претендовал на то, чтобы Postgres называли ООСУБД. Более того, создатель Postgres – в большей степени сторонник объектно-реляционного подхода, чем чисто объектно-ориентированного. Это утверждение подтверждает уже упоминавшийся в этой главе второй манифест, к которому Стоунбрейкер имеет непосредственное отношение. Отметим, что второй манифест считают ответом инженеров-практиков, сторонников SQL-ориентированных систем идеям, изложенным учеными в первом манифесте.

Замечание

Полезно знать, что Postgres называют темпоральной СУБД. Это означает, что любое изменение кортежа в отношении приводит к появлению его новой копии, а предыдущий вариант просто скрывается от пользователя. Такое решение позволяет извлечь из БД любой вариант кортежа, указав время его существования.

UniSQL

Первой СУБД, получившей титул объектно-реляционной, считается корейская система UniSQL. Не станем обсуждать коммерческий успех (или, скорее, его отсутствие) у UniSQL, но научная ценность системы, появившейся на свет в 1991 году, безусловна. Разработчики СУБД под руководством Вона Кима³ пошли по наиболее рациональному пути и не стали отрицать реляционной модели. Вместо этого было принято решение о расширении реляционной модели, заключающееся в следующем:

- 1) атрибуты отношения получают право, помимо литеральных значений, хранить объекты;
- 2) отменяется требование атомарности значения атрибутов;
- 3) при построении таблиц (в понимании UniSQL-классов) допускается ключевой для ООП механизм наследования;
- 4) классы включают операции.

Еще одно интересное решение, реализованное в UniSQL, получило название **федеративной системы БД**. Суть идеи заключалась в том,

³ Известность доктору Вону Киму пришла еще во время его участия в экспериментальном проекте прототипа реляционной СУБД System R компании IBM.

что корейская СУБД научилась формировать единое представление данных, хранящихся как собственно в UniSQL, так и в ряде других поддерживаемых СУБД (Oracle, Sybase, Informix). Для этих целей сервер UniSQL размещался на логическом уровне, располагающемся над совместимыми СУБД, и в интересах клиентских приложений формировал глобальную схему данных.

Cache

Cache – это кроссплатформенная (поддерживаются операционные системы UNIX, Linux, Windows, Mac OS X) иерархическая СУБД производства компании InterSystems. Система позволяет:

- хранить объекты (данные и методы их обработки), обеспечивая независимость хранения данных от способа их представления;
- при определении класса объектов одновременно автоматически на языке SQL генерируется реляционное описание этого класса;
- доступ к данным может быть осуществлен тремя способами:
 - прямой доступ к данным (Cache Direct Access) позволяет непосредственно работать со структурами хранения;
 - реляционный доступ к данным (Cache SQL);
 - объектный доступ (Cache Objects) позволяет взаимодействовать с языками программирования Java, Visual C++, Delphi.

Важной особенностью Cache является то, что данные находятся под управлением многомерного сервера данных, что позволяет хранить данные в виде многомерных разреженных массивов.

Versant Object Database

Versant Object Database (VOD) представляет собой программное обеспечение для управления ООБД, разработанное корпорацией Versant.

База данных VOD полностью совместима с языками программирования Java, C# и C++, с некоторыми ограничениями поддерживается Smalltalk и Python. Перечисленные языки могут использоваться для разработки базы данных. Для доступа к данным используется фирменный язык Versant Query Language (VQL), по своим возможностям напоминающий SQL:92.

VOD поддерживает транзакции, работающие по протоколу двухфазной блокировки, система хорошо масштабируема и обладает высокой производительностью.

ObjectStore

ObjectStore представляет собой коммерческую ООСУБД компании Object Design. Предназначена для обработки данных, созданных приложениями, использующими методы объектно-ориентированного программирования.

Для создания объектов БД и прозрачного доступа к ним используется язык C++, вместе с тем благодаря технологии подстановки указателей обращение к объектам можно осуществлять из большинства современных объектно-ориентированных языков программирования.

Что пошло не так?

После провозглашения лозунга о «моральном устаревании» реляционной модели и неминуемом приходе на смену ей более прогрессивной ООБД прошло примерно тридцать лет. Немалый срок для подведения итогов. К какому результату мы пришли сегодня?

В первую очередь стоит констатировать тот факт, что реляционная модель (со всеми ее недостатками) по-прежнему является доминирующей моделью в области хранения и обработки данных. А объектно-ориентированная модель, несмотря на ее безусловную успешность в области языков программирования и проектировании прикладного ПО, так и не смогла занять достойное место в сегменте БД.

Что пошло не так с ООБД?

Дело в том, что между теорией баз данных (для которой и создавалась реляционная модель) и технологией разработки прикладного ПО (основная область интересов ОО парадигмы) есть ряд принципиальных отличий. Остановимся на главном из них.

Кто не слышал поговорку про то, что математика является царицей наук? Это выражение принадлежит Карлу Фридриху Гауссу, и оно было произнесено в далеком XVIII веке и вряд ли когда будет опровергнуто. Если у вас есть математический аппарат, описывающий ту или иную область знаний, значит, в вашем распоряжении есть все необходимое для решения любой задачи в этой области.

Реляционная модель данных опирается на глубоко проработанный и одновременно простой в понимании и практическом применении математический аппарат теории множеств. Если читатель пороется в математических справочниках, то он обнаружит примерно такое определение: теория множеств – это раздел логики и математики, в рамках которого изучаются классы (множества) элементов произвольной при-

роды. Таким образом, теория множеств (а как следствие и реляционная модель) является универсальным инструментом, позволяющим решить практически любую задачу в области БД. При этом акцентируем внимание читателя на одном немаловажном замечании – теории множеств абсолютно безразлично, какого рода объекты составляют ее множества, как следствие мы можем создавать БД для множеств, формулировка которых нам неизвестна на данный момент! Недаром в определении говорится о «произвольной природе» элементов множества.

А как обстоят дела с объектно-ориентированной концепцией? На каком базовом математическом аппарате построена идеология ООБД и используемого в таких базах данных языка доступа к объекту?

Будет неверным утверждать о том, что ООБД обойдены вниманием математиков. Например, в работе [2] выделяется два подхода: алгебра *select-project-join* (SPJ) и полное исчисление предикатов высшего порядка, которые могут быть полезны при построении математической модели ООБД. Оба упомянутых аппарата могут составить основу так называемого функционального подхода (задавать объекты, базовые функции, применимые к объектам, и структурные операции над функциями).

Кроме того, выделяется так называемый дедуктивный подход [19]. Суть подхода заключается в представлении запросов для ООБД в виде формул формальной теории, получаемых с помощью логического вывода в этой теории.

Но в сравнении с изяществом математического аппарата реляционной модели и простотой соответствующего ей языка SQL теоретический фундамент ООБД пока выглядит невзрачно.

Недостатки ООБД

Как сторонники, так и критики объектно-ориентированного подхода в построении БД сходятся во мнении, что движению вперед мешает ряд пока окончательно не устраненных противоречий. Основные из них [19, 25, 38]:

- 1) отсутствие универсальной модели данных для ООБД. Мы уже упомянули основную причину этого – недостаточную простоту и проработанность математического аппарата;
- 2) отсутствие современных общепризнанных стандартов. Продекларированные в 1990-х манифесты и стандарт ODMG 3.0 начала 2000-х не в полной мере отвечают современным потребностям разработчиков объектно-ориентированных БД и нуждаются в

совершенствовании. На сегодняшний день их в лучшем случае можно рассматривать лишь как точку отсчета для перспективного стандарта;

- 3) низкий уровень обеспечения безопасности. Пока в ООБД основным механизмом защиты данных выступает разрешение/запрет доступа пользователя к отдельным объектам или классам. Повышение степени детализации (доступ к отдельным методам, управление наследованием, передача полномочий) приводит к существенному росту сложности ПО, без какого-то существенно-го выигрыша в безопасности;
- 4) высокая сложность. Развитые функциональные возможности ООБД приводят к общему усложнению процессов разработки и эксплуатации программного обеспечения;
- 5) сложность оптимизации запросов. Задача формирования быстрого в выполнении запроса вступает в противоречие с концепцией объектно-ориентированного программирования. Суть противоречия в том, что оптимизатор запросов должен знать реализацию объекта, а инкапсуляция препятствует ему в этом;
- 6) низкая производительность. Особенно производительность снижается в условиях параллельного доступа, т. к. основным элементом блокировки выступает объект;
- 7) недостаточная совместимость между ООБД разных производителей усложняет переход с одного программного продукта на другой.

В качестве «вишенки на торте» можно упомянуть сложность освоения ООБД рядовыми пользователями, ведь в сравнении с реляционными БД (для создания таблицы которой не требуется особой подготовки) в объектно-ориентированных системах пользователь должен быть компетентен в вопросах программирования. В данном контексте приведем немного сокращенную цитату из книги К. Дж. Дейта, которая обобщает мнение многих специалистов [19]:

«Реляционные СУБД поступают от изготовителя готовыми к использованию. Иными словами, как только система установлена, пользователи могут начать строить базы данных, разрабатывать приложения, запускать запросы и т. д.

Объектную же СУБД можно считать лишь некоторого рода набором средств построения СУБД. После исходной установки объектная СУБД не готова к немедленному использованию. Сначала она должна быть приспособлена к определенной области применения опытными специалистами...»

К сказанному нечего добавить, кроме того что если вы планируете разработать и внедрить ООБД, то существенной статьей расходов станет оплата труда высокопрофессиональных программистов (которых, кстати, не так просто найти).

Накалу страстей способствует еще тот факт, что реляционные СУБД не стоят на месте. Компании-производители очень быстро оценили требования рынка и стали встраивать объектно-ориентированный функционал в уже существующие СУБД (в этом плане хрестоматийным примером можно считать современные версии Oracle). По этому поводу позволим себе еще одну цитату [38]: *«проблема рынка ООСУБД состоит еще в том, что многие возможности ОО-подхода встраиваются в СУРБД при сохранении концептуальной простоты этих моделей, тем самым снижая привлекательность самих ООСУБД»*.

Объектно-реляционные СУБД

Попытка прямолинейного применения объектно-ориентированного подхода в форме ООБД если и не потерпела фиаско, то, по крайней мере, не стала столь успешной, как планировалось. Вместе с тем никто не ставил под сомнение факт о необходимости использования объектно-ориентированных идей в области хранения и обработки данных. Выходом из сложившейся ситуации стало объединение всех преимуществ реляционной модели и идей объектно-ориентированного программирования. Результатом стало появление объектно-ориентированных СУБД.

Замечание

Суть объектно-реляционной СУБД заключается в том, что над проверенной временем реляционной платформой размещаются все необходимые объектно-ориентированные расширения.

В 1994 г. Михаэл Стоунбрейкер опубликовал статью, в которой говорится, что объектно-реляционная СУБД – это СУБД, которая добавляет к SQL следующие объектно-ориентированные концепции [42]:

- 1) уникальные идентификаторы;
- 2) типы, определяемые пользователем;
- 3) операторы, определяемые пользователем;
- 4) методы доступа, определяемые пользователем;
- 5) сложные объекты;
- 6) функции, определяемые пользователем;
- 7) перегрузку (переопределение);

- 8) динамическую расширяемость;
- 9) наследование как данных, так и функций;
- 10) массивы.

Предсказания начали сбываться вместе с выходом стандартов SQL-1999 и SQL-2003, кроме того, производители реляционных СУБД стали активно внедрять объектно-ориентированные расширения в свои продукты. К подобным решениям относятся такие СУБД, как Oracle Database, Informix, DB2, PostgreSQL.

Рассмотрим небольшой фрагмент кода на PL/SQL, демонстрирующий работу объектно-реляционных средств в СУБД Oracle (см. листинг 23.1).

Листинг 23.1. Добавление нового типа данных в БД Oracle

```
create or replace Address_Type
as object
(zip_code number,
 city varchar2(30),
 street varchar2(25),
 home varchar2(10),
 flat varchar2(5),
 member function getAddress return varchar2
)
not final;
create or replace type body Address_Type as
  member function getAddress return varchar2 is
  begin
    return zip_code||' '||city||' '||street||', '||home||', '||flat;
  end;
end;
```

В нашем примере мы создаем структуру, предназначенную для хранения адреса. Её особенность – наличие в структуре функции `getAddress`, позволяющей возвращать адрес в формате обычной текстовой строки. Это и есть простейший пример объявления объектного типа, обладающего своим собственным методом.

Резюме

Применение объектно-ориентированной концепции на рынке баз данных в первую очередь позволит разработчикам с более высокой точностью моделировать объекты реального мира и в целом повысит производительность работы программистов.

Объектно-ориентированные языки призваны создавать прикладное ПО, специализирующееся на решении не общих, а конкретных задач. Вы можете написать великолепный класс «отдел кадров», который сможет решать задачи по приему сотрудника на работу, переводу с должности на должность, увольнению на пенсию лиц, достигших предельного возраста, и т. п. И это будет более высокий уровень абстракции, чем традиционное для реляционной модели манипулирование кортежами отношений! Однако насколько окажутся применимы разработанные вами методы для конкретного предприятия, останется открытым вопросом. А если предприятию потребуется ввести новый атрибут с новыми данными либо изменить характеристики уже существующего? Реляционная модель справится с такой задачей в два счета. Что не скажешь про ООБД, ведь класс не столь поворотлив и универсален, как плоская двухмерная таблица.

К сожалению, до сегодняшнего дня ООБД не смогли занять достойного места на рынке программного обеспечения. Причин тому много, но основная из них – отсутствие глубокой теоретической проработки ОО-модели с построением адекватной математической модели, имеющейся у реляционной модели. Более того, сторонники реляционного подхода перешли в атаку и расширили реляционную модель, поддерживая некоторые элементы ОО-подхода. Современные объектно-реляционные СУБД поддерживают создание новых типов данных и сложных объектов, наделяют объекты методами, обеспечивают процедуру наследования и при этом остаются на фундаменте глубоко проработанной и проверенной временем реляционной модели.

Вопросы для самопроверки

1. На каких концепциях строится объектно-ориентированное программирование?
2. Какие недостатки реляционной модели послужили толчком к созданию ООБД?
3. Что понимается под объектом с точки зрения ООБД?
4. Что такое класс?
5. На основе каких ингредиентов строится ОО-концепция?
6. Как организована идентификация объектов в ООБД?
7. Существуют ли стандарты для ООБД?
8. С какими трудностями столкнулись создатели ООСУБД?
9. Какие современные СУБД поддерживают работу с объектами?
10. Что представляют собой объектно-реляционные СУБД?

Глава 24

Документ-ориентированные БД

Читателю наверняка знаком один из фундаментальных философских постулатов, утверждающий, что развитие всегда идет по спирали вверх. История формирования современных подходов к хранению данных целиком и полностью подтверждает закон науки о знаниях. Помните, что реляционная модель и правила нормализации данных требовали, чтобы любая сущность хранилась только в одном месте? Для достижения этого результата мы с вами разделяли БД на несколько отношений, связанных друг с другом отношениями 1:M и M:N. Строгая нормализация имеет много преимуществ (см. главу 7), однако обратной стороной медали выступают объективные сложности, возникающие перед нами в момент сборки в единое целое данных из десятков таблиц. И вот философская спираль сделала очередной виток, появились новые подходы к хранению данных, среди них и **документ-ориентированная модель**. Это полный антипод нормализации, не допускающий никакой декомпозиции! Вместо этого новая модель нацелена на работу с объектом как с единым целым.

Чем плоха нормализация?

Рискнем ввести читателя в состояние когнитивного диссонанса. Итак, на протяжении практически всей книги мы всячески нахваливали нормализацию и призывали активно применять нормальные формы при разработке баз данных. Что поменялось к этой главе? Неужели только сейчас наступило прозрение и мы поняли, что все это время глубоко заблуждались?

Нет, не заблуждались. В ситуации, когда разработчик стремится гарантированно обеспечить целостность и непротиворечивость данных, а также поддержать широчайший спектр возможностей построения запросов с помощью языка SQL, без нормализации не обойтись. Одна-

ко в определенных условиях один из столпов реляционной модели из помощника превращается в противника. Опыт эксплуатации реляционных БД показал, что нормализованные данные:

- недостаточно эффективны в распределенной среде;
- при использовании join-объединений ограничивают скорость чтений;
- физическая структура многотабличной организации данных не соответствует реальным данным, в первую очередь их физической структуре.

Замечание

Критикуя нормализацию, не забываем о ее ключевых достоинствах. Во-первых, благодаря нормализации устраняются аномалии обновления данных и обеспечивается целостность данных. Во-вторых, строгое структурирование нормализованных данных позволяет обеспечить широкий спектр произвольных запросов к данным.

БД ключ-значение

«Лобовым» способом повышения производительности хранилищ данных стало упрощение модели данных. Вместо реляционных таблиц можно использовать ассоциативный массив или словарь, позволяющий работать с данными по ключу. Подобные БД получили название хранилищ ключ-значение (key-value store). Данные здесь хранятся как совокупность пар ключ-значение, в которых ключ служит уникальным идентификатором. При этом в качестве ключей и значений может использоваться что угодно: от простых типов данных до сложных объектов.

Где может пригодиться столь упрощенная схема? В первую очередь при организации быстрого кеширования. Допустим, нам потребуется кешировать огромное количество HTML-страниц, тогда ключом станет URL-адрес страницы, а значением – собственно страница.

Самыми известными программными решениями ключ-значение стали БД Amazon DynamoDB и Apache Cassandra. Базы данных ключ-значение хорошо масштабируются, однако за повышение производительности БД мы расплатились ухудшением качества модели, достаточно сказать лишь то, что никакая системная информация о структуре данных в подобных БД не собирается.

Замечание

Хранилища ключ-значений в первую очередь востребованы для хранения очень больших объемов данных, поэтому их разработка ведется многими крупными интернет-компаниями (Amazon, Apache, Google, Facebook).

Документ-ориентированные БД

Документ-ориентированная БД (document-oriented database) – БД, специализирующаяся на хранении иерархических структур данных (документов) и обычно реализуемая с помощью подхода NoSQL. Основу таких БД составляют документные хранилища (англ. document store), обычно имеющие древообразную структуру. Такая структура опирается на корневой узел, от которого исходят внутренние и листовые узлы. Как раз листовые узлы и являются хранилищами документов.

Замечание

В качестве примеров наиболее известных документ-ориентированных СУБД можно привести MongoDB, CouchDB, Google Cloud Datastore и RavenDB.

В сравнении с реляционной моделью документ-ориентированная модель выглядит весьма необычно.

Во-первых, можно забыть о привычных нам структурных единицах (атрибут–кортеж–отношение). Базовая структурная единица документ-ориентированной модели – документ, кроме того, еще существует понятие коллекции – контейнера для схожих документов. С некоторой степенью допущения можно считать коллекцию аналогом таблицы, а документ – строкой, но в реляционной таблице все строки обладают идентичной структурой, а в документ-ориентированной БД все документы могут быть произвольными. В качестве иллюстрации рис. 24.1 предложен вариант данных в реляционной форме и в документ-ориентированном формате.

Во-вторых, операции выборки и модификации данных (реализованные в реляционной модели при посредничестве языка SQL с его мощным механизмом построения произвольных запросов) с участием операций соединения в документ-ориентированных БД отсутствуют, т. к. по определению здесь соединять нечего, ведь тут единицей хранения выступает неделимый документ.

В-третьих, привычных транзакций, позволяющих объединять в единое целое несколько операций, вы также не обнаружите. Поклонники

документ-ориентированной модели полагают, что здесь более важно реализовать надежные операции обновления встроенных в сложный документ структур. Кроме того, документ-ориентированные СУБД, как правило, ориентированы на работу с большими массивами данных, находящимися в распределенных системах, соответственно, обеспечение их обработки в рамках транзакций весьма проблематично – из-за этого может пострадать как скорость обработки, так и доступность данных.

Внимание!

Основное отличие между реляционными и документ-ориентированными БД заключается в том, что первые определяют столбцы на уровне отношения, в то время как вторые определяют поля на уровне документа, т. е. любой документ внутри коллекции может иметь свой собственный уникальный состав.

Экземпляр данных в реляционной форме

SUPPLIERS

SUPPLIER_ID	SUPPLIER
1	ОАО Юнитеп

COUNTRIES

COUNTRY_ID	COUNTRY
5	Япония

DELIVERYNOTES

DELIVERYNOTE_ID	SUPPLIER_ID	DNDATE	DNNUM
14	1	2018-11-08	614

VENDORS

VENDORS_ID	COUNTRY_ID	VENDOR
121	5	SONY

GOODSCLASS

GODSCLASS_ID	PARENT_ID	GOODCLASS
45	16	LCD TV

GOODSLIST

GOODSLIST_ID	DELIVERYNOTE_ID	VENDORS_ID	GODSCLASS_ID	GOODS	...
65327	14	121	45	BRAVIA KDL43WF804BR	...

Агрегат

```

{
  "supplier_id": 1,
  "supplier": "ОАО Юнитеп"
}

{
  "vendors_id": 121,
  "country": "Япония",
  "vendor": "Sony"
}

{
  "deliverynote_id": 14,
  "supplier_id": 1,
  "dndate": "2018-11-08",
  "dnnum": 614
  {
    "goodslist_id": 65327,
    "vendors_id": 121,
    "goodclass": "LCD TV",
    "goods": "BRAVIA KDL43WF804BR",
    "...": ...
  }
}

```

Рис. 24.1. Представление данных в реляционной и документ-ориентированной формах

NoSQL

Автором термина NoSQL можно считать программиста Карло Строззи (Carlo Strozzi), он в 2010 году так назвал свою СУБД – Strozzi NoSQL. Упомянутая СУБД хранит все данные в виде текстовых файлов и вместо SQL для доступа к данным использует шелловские скрипты. В момент выхода в свет программного продукта Карло Строззи термин NoSQL понимался прямолинейно и означал категорический отказ от SQL, и как бы это не показалось странным, с понятием «NoSQL» в его нынешнем виде он ничего общего не имеет....

Замечание

СУБД Strozzi NoSQL распространяется по правилам лицензии GNU General Public License и находится в открытом доступе по адресу: <http://www.strozzi.it/shared/nosql/>.

Сегодня большинство специалистов предпочитает расшифровывать NoSQL как Not Only SQL (не только SQL), хотя до сих пор существуют и сторонники прямого определения термина.

Что вкладывается в понятие NoSQL?

1. СУБД NoSQL не используют реляционную модель данных.
2. Структура данных в базах NoSQL никоим образом не регламентирована.
3. Подлежащие хранению сущности рассматриваются как целостные объекты, т. е. не существует никакой нормализации, «размазывающей» сущность по нескольким отношениям.
4. Не поддерживается традиционный SQL (в первую очередь речь идет о DML), хотя, с другой стороны, разработчики стараются реализовать свои языки запросов, по крайней мере, синтаксически похожие на привычный ANSI SQL.

В качестве примера конструкций NoSQL обратимся к синтаксису MongoDB [56]. Так, листинг 24.1 демонстрирует вставку в коллекцию «goods» БД «warehouse» документа с описанием телевизора.

Листинг 24.1. Вставка новой записи с описанием телевизора

```
use warehouse;
db.goods.insert({goodsclass: 'TV', vendor: 'LG', model: 'LG 43UJ634V', tuner:
['DVB-T2', 'DVB-C', 'DVB-S2'], resolution: '3840x2160', picture: 'Ultra HD 4K',
digonal: '43', interface: ['HDMI', 'USB', 'RJ-45', 'S/PDIF'], vesa: '200x200'});
```

Простейшая форма обновления данных принимает два аргумента: селектор, определяющий, что мы хотим обновить, и новое значение поля (листинг 24.2).

Листинг 24.2. Обновление значения в поле qled

```
db.goods.update({model: 'SAMSUNG QE49Q6FNAUX'}, {qled: 'true'});
```

В предложенном примере для телевизора модели «SAMSUNG QE49Q6FNAUX» был поставлен признак наличия технологии QLED.

Для выборки данных применяют селекторы. Например, листинг 24.3 находит все документы с информацией о телевизорах компании SONY.

Листинг 24.3. Выборка всех телевизоров компании SONY

```
db.goods.find({goodsclass: 'TV', vendor: 'SONY'});
```

Предложенный пример {поле1: значение1, поле2: значение2} работает как логическое «И». Если же нам необходимо описать условие «ИЛИ», обращаемся к листингу 24.4.

Листинг 24.4. Выборка всех телевизоров с диагональю 43 или 46 дюймов

```
db.goods.find({goodsclass: 'TV', $or: [{digonal: '43'}], [{digonal: '46'}]});
```

В последнем примере нам пришлось использовать оператор \$or и присвоить ему массив значений.

Во время выборки данных можно осуществлять их сортировку (листинг 24.5).

Листинг 24.5. Сортировка телевизоров по возрастанию размера диагонали

```
db.goods.find({goodsclass: 'TV'}),sort({digonal: 1});
```

Обратите внимание на то, что, назначая сортировку sort, мы указываем поля, по которым надо сортировать, используя 1 для сортировки по возрастанию и -1 для сортировки по убыванию.

Замечание

Доступные для загрузки файлы сервера MongoDB вы сможете найти в интернете по адресу <https://www.mongodb.com/download-center/community>.

Представленные пять примеров, конечно же, не раскрывают даже малой толики возможностей NoSQL, но зато дают читателю представление о самой концепции языка. Как видите, он прост и интуитивно понятен. Однако стандарта NoSQL пока не существует, и вероятность его появления в обозримом будущем невысока.

Внимание!

В октябре 2018 г. компания MongoDB объявила об изменении условий распространения исходных текстов проекта и переводе кода с AGPLv3 на новую лицензию

SSPL (Server Side Public License), в которой расширяются требования при использовании приложений как сервисов в облачных системах. На момент написания этих строк лицензия SSPL относится к разряду открытых лишь формально и официально не одобрена OSI, поэтому есть некоторые юридические нюансы по поводу бесплатного использования MongoDB в коммерческих целях.

Распределенная обработка MapReduce

Идея MapReduce зародилась в стенах компании Google как результат поиска наиболее адекватных решений по распределенной обработке больших объемов данных на компьютерных кластерах [4].

Суть парадигмы MapReduce заключается в том, что большая задача разделяется на ряд небольших заданий, каждое из которых может быть выполнено на любом из узлов кластера.

Допустим, в нашем исходном наборе данных имеются записи, содержащие разнотипные документы, и нам необходимо разбить их на однотипные группы. Решение такой задачи в MapReduce будет осуществлено в три этапа (рис. 24.2).

1. Фаза Map представляет собой предварительную параллельную обработку входных данных в соответствии с правилами, заданными пользователем. Реализованные в коде функции Map() правила применяются к каждой поступающей на вход записи, в результате мы получаем последовательность пар ключ-значение (k, v) .
2. Пары ключ-значение (k, v) , возвращаемые Map, обрабатываются и сортируются по ключу. Ключи разделяются между всеми функциями Reduce(), поэтому все пары ключ-значение с одним и тем же ключом k отправляются на вход одной и той же функции Reduce().
3. Фаза Reduce осуществляет обработку ключа и всех ассоциированных с ним значений $(k, [v_1, v_2, \dots])$. Функция Reduce() обрабатывает входные данные в соответствии с правилами, запрограммированными пользователем, и возвращает последовательность из нуля, одной или более пар ключ-значение.

Замечание

В технологии MapReduce следует соблюдать принцип локальности ссылок – для того чтобы снизить объем передаваемых данных по сети, функция Map должна применяться на той же станции, на которой и хранятся данные.

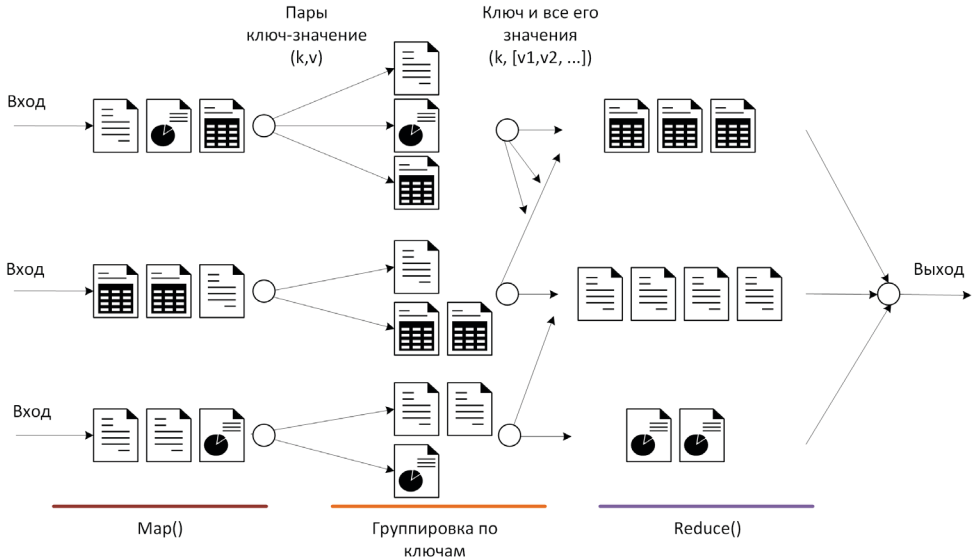


Рис. 24.2. Модель MapReduce

Существенной особенностью модели MapReduce является то, что все запуски функций Map() и Reduce() могут осуществляться независимо на разных компьютерах, входящих в распределенную систему (кластер). То же самое можно сказать и о промежуточной операции группировки по ключам. Все это в целом обеспечивает выполнение важного в работе с большими данными в распределенных системах принципа горизонтальной масштабируемости.

Замечание

В ситуации, когда компьютерная система перестает справляться с нагрузкой, можно воспользоваться как вертикальным, так и горизонтальным масштабированием. Вертикальная масштабируемость предполагает наращивание вычислительных ресурсов сервера (процессор, память и т. п.), но ресурсы нельзя увеличивать бесконечно – очень скоро наступает предел. Поэтому при работе с большими данными предпочтительнее горизонтальная масштабируемость, которая решает проблему за счет наращивания числа машин.

Сегментирование

В ситуации, когда документ-ориентированная БД эксплуатируется в рамках небольшого офиса в качестве платформы для развертывания

СУБД, достаточно единственного сервера. Но по мере роста предприятия, объема данных и числа запросов к ним нам потребуются более серьезные возможности по осуществлению операций чтения/записи, что окажется непосильным для одного, пусть даже весьма проворного компьютера. В таком случае для обеспечения заданного уровня производительности осуществляют сегментирование – распределение БД на несколько серверов.

Замечание

Среди популярных документ-ориентированных СУБД не все применяют сегментирование. Например, от сегментирования отказалась CouchDB. В этой СУБД на всех компьютерах хранятся полные реплики БД.

Какой показатель может служить маркером, свидетельствующим о необходимости сегментирования? Специалисты полагают, что сервер, на котором эксплуатируется БД, должен обеспечивать возможность размещать рабочий набор данных и его индексы в оперативной памяти. С ростом БД объем свободного ОЗУ постепенно сокращается – это и есть повод перевода БД на сегментированный кластер (рис. 24.3).

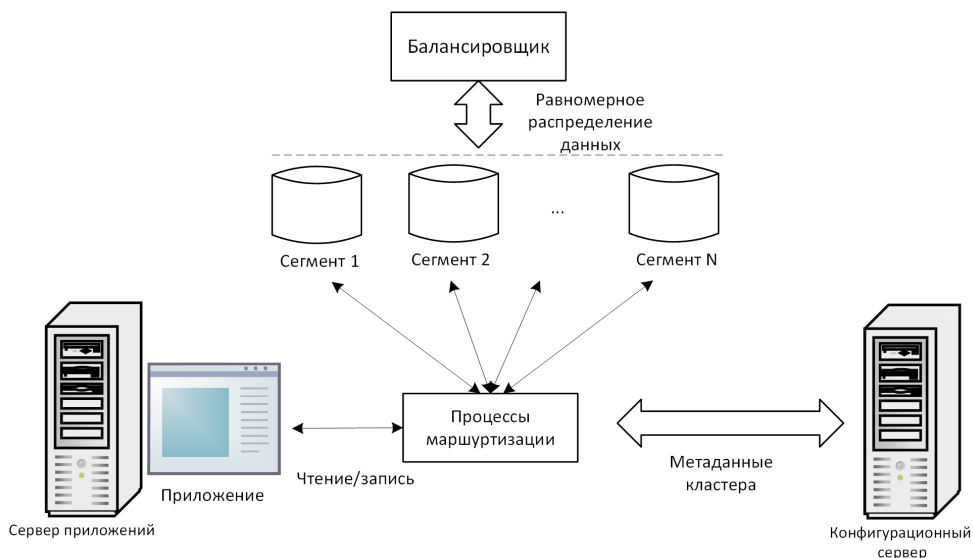


Рис. 24.3. Состав сегментированного кластера

В состав сегментированного кластера обычно входят следующие компоненты:

- 1) **сегменты**, физически сегмент представляет собой некоторое подмножество данных из БД;
- 2) **сервер приложений**, предоставляет в распоряжение пользователей приложения, реализующие интерфейс доступа к данным;
- 3) **процессы маршрутизации**, выступающие посредниками между приложениями и сегментами. Отдельный процесс маршрутизации отвечает за отправку команд на чтение/запись данных к нужному сегменту;
- 4) **конфигурационный сервер**, обеспечивает хранение метаданных сегментированного кластера и доступ к ним, как правило, в кластере должно быть развернуто несколько хостов с конфигурационными серверами;
- 5) **балансировщик**, осуществляет равномерное распределение данных между сегментами.

Сегментирование данных пользователей может осуществляться по географическому расположению, по типу документов, по синтетическому идентификатору. Последний подход наиболее хорош, когда стоит задача равномерно распределить данные по кластеру.

Репликация

Для достижения максимальной производительности и улучшенного масштабирования документ-ориентированные СУБД отказались от такой важной функции, как поддержка транзакций с полноценным соблюдением классических ACID-требований к ним. Как следствие пострадали такие важные показатели, как надежность и доступность данных.

Для того чтобы компенсировать негативные последствия отказа от транзакций и снижения риска утраты данных, в документ-ориентированных СУБД обязательно организуется репликация.

Внимание!

В документ-ориентированных системах под репликацией понимается размещение и обслуживание серверов БД на нескольких сетевых станциях.

Схемы организации репликации могут быть самые разнообразные, от полного копирования БД на всех узлах до формирования сложных алгоритмов создания частичных копий с заданным процентом перекрытия. Главное в том, что в случае сбоя в работе СУБД многочисленные реплики обеспечат восстановление утраченных данных.

Когда следует использовать документ-ориентированную модель?

Ключевое преимущество документ-ориентированного подхода – возможность использования неструктурированных данных. Этот фактор является доминирующим при принятии решения, какому типу СУБД следует отдать предпочтение на этапе концептуального проектирования БД. Но это не единственный критерий выбора, документ-ориентированная модель и NoSQL окажутся полезными в ситуации, когда:

- 1) подлежащие хранению данные не структурированы, требования к данным трудно формализуемы или могут изменяться с развитием проекта;
- 2) разработка системы хранения данных ведется в условиях цейтнота времени;
- 3) из всех требований к БД доминирующими выступают скорость обработки данных и масштабируемость.

Если же фундаментальными требованиями к БД выступают:

- 1) наличие строгой структуры подлежащих хранению данных;
- 2) обеспечение целостности данных;
- 3) применение проверенных и стандартизированных технологий;
- 4) развитая система безопасности, –

то в перечисленных случаях предпочтение следует отдать проверенной временем реляционной модели.

Резюме

Документ-ориентированные БД и движение NoSQL набирают популярность гигантскими темпами. Сегодня уже нельзя утверждать об исключительной монополии реляционной модели как безальтернативного хранилища данных. Теперь возможно не втискивать данные в узкие рамки структурных ограничений, задаваемые плоскими таблицами и правилами нормализации, а создавать хранилища, исходя из природы самих данных.

Безусловными сильными сторонами документ-ориентированных БД выступают:

- 1) хранение и обработка больших объемов неструктурированных данных;

- 2) поддержка облачных хранилищ и облачных вычислений;
- 3) высокая горизонтальная масштабируемость;
- 4) быстрая (по сравнению с реляционной) разработка БД.

Однако это не означает, что реляционные базы данных переходят в разряд рудиментарных. Просто мы вступаем в этап, когда для различных потребностей будут использоваться разные хранилища данных. Поэтому реляционные БД по-прежнему будут востребованы. Ко всему прочему благодаря высокой приспособляемости реляционной модели совсем недавно появились решения, способные эффективно работать с большими данными в распределенной среде, они получили название NewSQL.

Вопросы для самопроверки

1. Почему нормализация снижает производительность обработки данных в распределенной среде?
2. Какие преимущества документ-ориентированных БД вам известны?
3. Для каких целей создавались БД «ключ-значение»?
4. Какие основные отличия между реляционными и документ-ориентированными БД вам известны?
5. Какой смысл заложен в термин NoSQL?
6. В чем заключается идея распределенной обработки MapReduce?
7. Раскройте состав сегментированного кластера.
8. Почему в документ-ориентированных БД обязательно должна быть организована репликация данных?
9. В каких случаях при проектировании БД следует применять документ-ориентированный подход?
10. Какой, на ваш взгляд, основной недостаток NoSQL-решений?

Глава 25

Большие данные

С приходом XXI века в лексиконе специалистов по БД появился термин «большие данные» (Big Data). Новый термин быстро набрал популярность и постепенно от специалистов перекочевал в словарный запас обычных пользователей – возможно, поэтому стал толковаться последними излишне прямолинейно.

Ради эксперимента попробуйте среди друзей и знакомых провести опрос на тему: «Где та грань, перейдя которую, обычные данные превращаются в “большие”?» Можно ручаться, что у каждого опрашиваемого найдется свой вариант ответа. Например, для автора этих строк в начале 1990-х жесткий диск размером 10 Мб (в комплекте с персональным компьютером ЕС-1841) казался несбыточной мечтой. Вот это и есть настоящее хранилище для «Big Data» в 1990-е годы... Сегодня 10 Мб даже и данными не назовешь, так, один короткий музыкальный трек или фотография высокого разрешения. Кто знает, что будет считаться нормой через 20 лет? Нет проблем только у маркетологов – они придумают очередной новый термин, что-нибудь вроде «Extra Big Data».

Замечание

У термина «Big Data» есть дата рождения – 3 сентября 2008 года. Это дата выхода широко известного британского журнала Nature, посвященного поиску ответа на вопрос «Как могут повлиять на будущее науки технологии, открывающие возможности работы с большими объемами данных?».

Так, может, «большие данные» на самом деле не более чем дань моде, и со временем культ «Big Data» сойдет на нет? Давайте рассмотрим несколько примеров.

Если бы вы были владельцем крупной розничной сети магазинов, вам наверняка хотелось бы владеть информацией обо всех совершаемых в ней покупках. И это не просто сведения о сделках, гораздо по-

лезнее владеть полной информацией о категориях покупок, когда они совершались, какие товары входят в обобщенную покупательскую корзину, какие товары чаще всего возвращают и почему, что лучше продается утром, в обед и вечером, какие бонусные программы наиболее популярны.

Если вы разработчик крупной интернет-площадки (допустим, поискового сервиса), вам, конечно, захочется обладать всеми сведениями о своих клиентах, их предпочтениях, какие сайты они посещают, какие социальные сети им ближе, какие новости читают, какими товарами интересуются, в каких отелях они бронируют места, какую музыку слушают, какой авиакомпанией летают в отпуск.

Если вы банкир, то вам не обойтись без сведений о транзакциях клиентов, их кредитах и депозитах, выпуске карт, реструктуризации задолженностей, программе лояльности, курсах основных валют и драгоценных металлов и еще тысяче показателей.

Список можно продолжить: операторы связи, авиакомпании, транспортные перевозки, социальные сети, научные исследования, поисковые сервисы, обеспечение безопасности (например, распознавание лиц), военное дело.

Во всех случаях анализу подлежат действительно большие данные, которые постоянно накапливаются и обновляются. И если в вашем распоряжении имеются все необходимые инструментальные средства, чтобы собрать, обработать потоки данных и быстро сделать на их основе соответствующие аналитические выводы (big data analytics), то вы окажетесь на шаг впереди своих конкурентов, исследующих аналогичные проблемы по старинке.

Что такое «большие данные»?

Договоримся сразу – не станем понимать термин «Big Data» буквально. Здесь речь идет не столько о терабайтах (эксабайтах, петабайтах и т. п.), сколько о технологии обработки больших данных.

Не случайно, что сам термин «Big Data» появился на свет в 2000-е годы, этому способствовал ряд обстоятельств. Среди них:

- 1) повышение доступности сети интернет для конечного пользователя;
- 2) рост скорости передачи данных в глобальных сетях;
- 3) увеличение объемов носителей данных с одновременным удешевлением их стоимости;

4) рост компьютерной грамотности среди жителей нашей планеты.

Все привело к лавинообразному росту объемов передаваемых и обрабатываемых данных. В результате сегодня едва ли не все 7,5 млрд землян стали как генераторами, так и потребителями больших данных, зачастую даже не осознавая этого. Хотя, конечно же, немалую лепту в объем данных вносят неодушевленные системы (мобильные устройства, системы видеонаблюдения, операционные системы, банковские транзакции, различные датчики и т. п.).

В начале тысячелетия, в 2001 году, Доуг Ленси (Doug Lancy), в то время сотрудник аналитической компании Meta Group¹, в статье «3D Data Management: Controlling Data Volume, Velocity, and Variety» предложил три характеристики, свойственные для больших данных (правило 3V):

- Volume – физический объем данных;
- Velocity – скорость прироста данных и скорость быстрой обработки данных с целью получения результатов;
- Variety – вариативность, предполагает возможность одновременной обработки различных типов данных.

Как видите, данным, дабы претендовать на звание больших, только размера недостаточно, хотя именно в данном случае размер, безусловно, имеет значение.

Определение

Большие данные – это комплекс технологий, позволяющих обрабатывать как хранящиеся данные, так и поступающий в реальном времени поток разнотипных структурированных и слабоструктурированных данных значительных объемов для получения воспринимаемых человеком результатов.

Таким образом, технология больших данных на самом деле триединая – она призвана отвечать за:

- 1) хранение и управление огромными объемами данных (сегодня исследователи говорят о необходимости эффективной работы петабайтами данных);
- 2) организацию неструктурированных и слабоструктурированных данных (тексты, изображения, видео и т. д.);
- 3) получение из больших данных аналитических выводов, которые будут полезны в практической деятельности человека.

¹ В 2005 году Meta Group была поглощена исследовательской компанией Gartner.

Принципы работы с большими данными

Выделяют три базовых принципа работы с большими данными:

- горизонтальная масштабируемость, обеспечивающая обработку данных, предполагает, что данные распределены по узлам таким образом, что их обработка не приводит к снижению производительности системы;
- отказоустойчивость, должна позволять свести к минимуму влияние на работу с данными возможные отказы оборудования;
- локальность данных, требует, чтобы по возможности данные обрабатывались на том же компьютере, на котором они и хранятся. Иначе расходы на передачу данных от места хранения к месту обработки станут неподъемными.

Каким образом воплотить перечисленные принципы на практике? Один из подходов, широко применяемый для работы с распределенными данными, нам уже известен – это MapReduce (см. главу 23). Но механизм распределенной обработки MapReduce сам по себе не панацея, при работе с большими данными должны применяться соответствующие архитектурные решения, самое популярное из них – лямбда-архитектура.

Лямбда-архитектура

При осуществлении анализа данных современные компании и предприятия уже не могут довольствоваться только срезами аналитической информации, поставляемыми им хранилищами данных. Особенности ведения бизнеса предполагают необходимость одновременного анализа как уже имеющихся данных, так и данных, поступающих в реальном времени.

Нужен пример? У нас наверняка есть мобильное устройство (смартфон, планшет), с которым вы отправляетесь в поездку. Обращаясь за помощью к тому или иному приложению, развернутому на наших устройствах (допустим, сервису Google Maps), мы заставляем компанию Google решать весьма нетривиальную задачу. С одной стороны, от миллионов пользователей в реальном времени идет поток GPS-данных об их местоположении, а с другой – на основе накопленных на серверах Google сведений о наших с вами предпочтениях нам формируют соответствующий контент (не забывая еще о включении неназойливой рекламы расположенной неподалеку пиццерии).

Как обеспечить доступ к подобного рода большим и одновременно быстрым данным в реальном масштабе времени? Ответ таков – применить соответствующую проблеме технологию, которая получила название лямбда-архитектура.

Определение

Лямбда-архитектура (The Lambda Architecture) – это подход к процессингу больших данных, который использует преимущества обработки как пакетных, так и потоковых методов обработки.

Лямбда-архитектура состоит из трёх взаимодополняющих уровней (рис. 25.1).

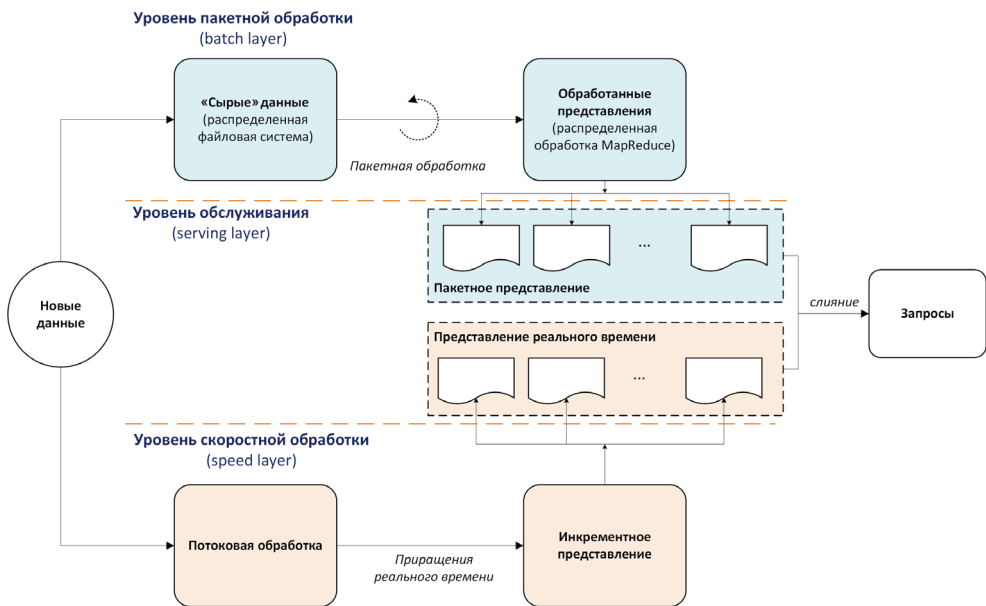


Рис. 25.1. Лямбда-архитектура

Уровень пакетной обработки (batch layer) представляет собой хранилище «сырых» данных. Здесь ведется процессинг по расписанию – через заранее заданные интервалы времени отправляются запросы к новым данным. Полученные данные просто добавляются к накопленному ранее архиву, не изменяя его предыдущие копии. Обычно уровень пакетной обработки реализуется на базе Apache Hadoop.

Уровень скоростной обработки (speed layer) осуществляет обработку поступающего в реальном времени потока данных в целях предо-

ставить потребителям данных самую последнюю актуальную информацию. Работа в реальном времени допускает некоторые «жертвы», например вы можете получить недостаточно точные или неполные данные. Однако эти погрешности с небольшим запозданием компенсируются уровнем пакетной обработки. Уровень скоростной обработки может быть построен силами следующих программных продуктов: Apache Storm, Apache Samza, Apache Spark и SQLstream.

Данные, полученные от уровней пакетной и скоростной обработки, сохраняются на **уровне обслуживания** (serving layer). Слой реагирует на запросы от операторов и возвращает им заранее подготовленные или подготовленные «на лету» представления. На стыке сервисного уровня и уровня скоростной обработки может работать как уже знакомая нам документ-ориентированная СУБД MongoDB, так и ряд других систем (Apache Cassandra, Apache HBase). Перечисленные программные продукты формируют представление данных реального времени. Пакетное представление данных помогут реализовать Elephant DB, Apache Impala, SAP HANA или Apache Hive.

Замечание

Упомянутые программные продукты не составляют исчерпывающий список приложений, способных работать с Big Data. Их объединяет только то, что в своем большинстве они представляют собой решения с открытым кодом и являются бесплатными или условно-бесплатными. Кроме них, на рынке имеются и продукты для работы с большими данными крупных производителей программного обеспечения, например Azure Cosmos DB (компания Microsoft), The Oracle Big Data Platform, IBM InfoSphere BigInsights.

Apache Hadoop

Для того чтобы получить возможность хранить большие данные в распределенных системах и при этом обеспечить управляемость всего кластера с данными, необходимо было разработать соответствующее программное обеспечение. Одним из пионеров этого направления стали программисты-энтузиасты из организации Apache Software Foundation. Ими в 2005 году был разработан проект под названием Apache Hadoop, который первоначально задумывался как система хранения данных, способная запускать задачи MapReduce, к сегодняшнему дню Hadoop превратился в целый стек компьютерных технологий, способных работать на уровне пакетной обработки в лямбда-архитектуре.

Замечание

Официальный сайт проекта читатель найдет в интернете по адресу <http://hadoop.apache.org/>. На момент написания этих строк для разработчиков проектов Big Data была доступна для скачивания версия Apache Hadoop 3.1.1, однако пока самой стабильной версией считается 2.9.1.

В ядро пакета Apache Hadoop входят три ингредиента:

- распределенная файловая система (Hadoop Distributed File System, HDFS);
- YARN – осуществляет управление ресурсами и задачами кластера, для этого он создает контейнеры для приложений, следит за их потребностями в ресурсах и выделяет дополнительные по мере необходимости;
- Common – представляет собой набор компонентов и интерфейсов для распределенных файловых систем и общего ввода-вывода:
 - HADOOP Resource Estimator является оценщиком ресурсов и анализатором нагрузок в вычислительном кластере. Он способен автоматически оценивать потребности в ресурсах на том или ином рабочем месте на основе истории работы;
 - Aliyun OSS Support – модуль, позволяющий интегрировать систему с сервисом Aliyun Web Services.

Визитная карточка Hadoop – распределенная файловая система. Если файловая система на персональном компьютере в самом общем случае представляет собой таблицу файловых дескрипторов и собственно данных, то в файловой системе HDFS все это отображено на клиент-серверную архитектуру (рис. 25.2). Благодаря этому HDFS надежна, хорошо масштабируема и позволяет хранить данные едва ли неограниченного объема.

В кластер HDFS входит один сервер NameNode (узел имен), он управляет пространством имен файловой системы и регулирует доступ к файлам клиентами. Кроме того, существует несколько узлов данных DataNodes, которые управляют хранилищем, прикрепленным к узлам, на которых они запускаются. HDFS предоставляет пространство имен файловой системы и позволяет сохранять пользовательские данные в файлах. Внутри файл разбивается на один или несколько блоков, и эти блоки хранятся в наборе DataNodes. Сервер имен NameNode выполняет операции с пространством имен файловой системы – это стандартные задачи, связанные с открытием, закрытием и переименованием фай-

лов и каталогов. Также сервер имен определяет отображение блоков в DataNodes. DataNodes отвечают за обслуживание запросов на чтение и запись от клиентов файловой системы. Узлы данных также выполняют создание, удаление и репликацию блоков по команде из NameNode.

Замечание

Гранды программной индустрии IBM, Oracle, Microsoft и ряд успешных новых компаний (Cloudera, MapR, Platfora и Trifacta) на базе Hadoop разработали свои решения по работе с большими данными.

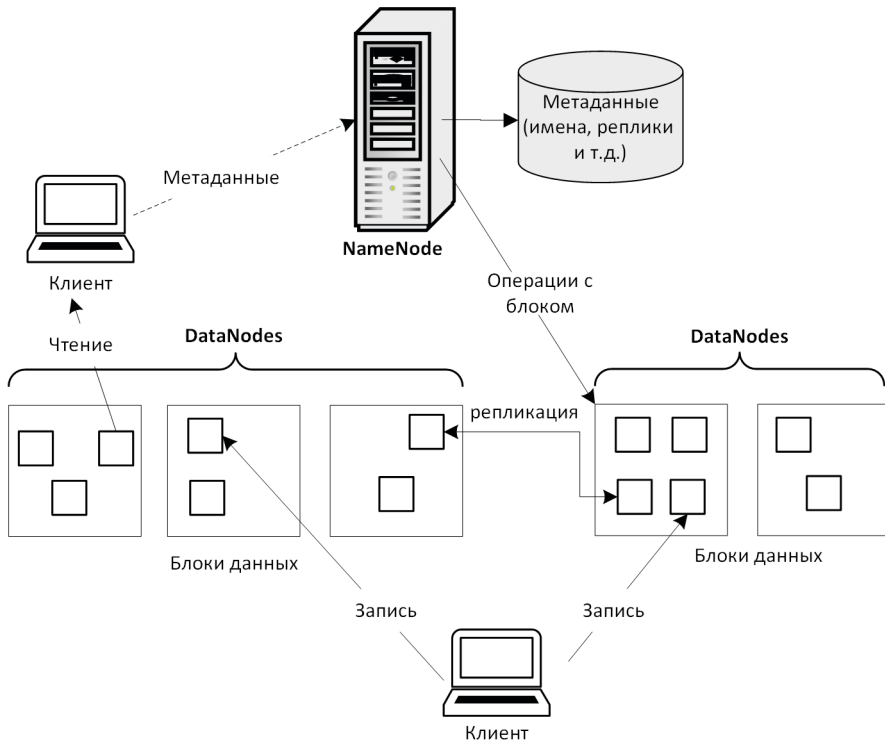


Рис. 25.2. Распределенная файловая система HDFS

Apache Storm

Apache Storm – фреймворк для распределенных потоковых вычислений, способный работать на уровне скоростной обработки лямбда-архитектуры. Позволяет обеспечить потоковую обработку данных в реальном времени. Это продукт с открытым кодом и может быть использован в соответствии с лицензией Apache License версии 2.0.

Замечание

Официальный сайт Apache Storm расположен по адресу <https://storm.apache.org/>.

Система создает граф вычислений реального времени, называемый топологией (topology). Принцип работы топологии аналогичен заданию MapReduce, за тем исключением, что задание MapReduce заканчивается, а топология будет работать бесконечно (пока вы его не остановите). Основными элементами топологии являются спауты (spout) и болты (bolt).

Спауты формируют поток в топологии. Поток представляет собой неограниченную последовательность кортежей, которые обрабатываются и создаются параллельно распределенным образом. Кортежи могут содержать целые и вещественные числа, строки, логические значения, байт-массивы и определяемые пользователем типы.

Болты выполняют преобразование потоков (агрегирование, соединение, фильтрация и т. п.). Болты, в свою очередь, могут передавать данные другим болтам для выполнения последовательных стадий обработки.

Apache Impala

Apache Impala представляет собой встроенную аналитическую БД для распределенной файловой системы HDFS и HBase. Система с открытым исходным кодом была разработана в 2013 году.

Замечание

Официальный сайт Apache Impala расположен по адресу <https://impala.apache.org/>.

Система использует привычный большинству разработчиков язык SQL-запросов. Таким образом, мы можем обращаться к данным, хранящимся в HDFS или Apache HBase, с помощью традиционных инструкций SELECT с поддержкой соединений JOIN и агрегирующих функций в режиме реального времени. Хорошо работает в многопользовательской среде с высокой конкуренцией запросов, кеширует часто запрашиваемые данные в памяти, способна управлять очередью запросов.

Apache Kafka

Рассмотрим еще один популярный продукт, разработанный в 2011 году компанией LinkedIn и получивший название Apache Kafka. Это распределенная система обмена сообщениями с высокой пропуск-

ной способностью между компонентами программной системы, работающая по принципу «публикация–подписка» (Publish and subscribe).

Замечание

Официальный сайт Apache Kafka расположен по адресу <http://kafka.apache.org/>.

Apache Kafka позволяет пропускать через централизованную среду огромное количество сообщений, а затем хранить их, не беспокоясь о производительности и не опасаясь, что данные будут потеряны.

Потоковая платформа имеет три ключевые возможности:

- 1) публикация и подписка на потоки записей, похожие на очередь сообщений или корпоративную систему обмена сообщениями;
- 2) отказоустойчивый способ хранения потоков записей;
- 3) обработка потока записей осуществляется по мере их появления.

Существенная особенность Apache Kafka, по сравнению с другими системами, реализующими схожий функционал, заключается в том, что Kafka все потоки записей не хранит в оперативной памяти, а сразу сбрасывает их на диск. На первый взгляд может показаться, что такое решение должно существенно затормозить работу системы, однако особенности хранения потоков, умноженные на высокоэффективные алгоритмы, позволяют доставлять сообщения до потребителей практически моментально.

NewSQL

Уже первый опыт работы с большими данными показал, что многим компаниям по-прежнему актуальны СУБД, обеспечивающие поддержку надежных транзакций. Банки, торговые площадки, крупные финансовые учреждения и ряд других организаций должны быть уверены, что платежная операция будет гарантированно выполнена. Но в это же время системы NoSQL, специально «заточенные» на распределенные данные, как известно читателю, не поддерживают транзакции в классическом их понимании (см. главу 24).

Для решения проблемы транзакций для больших данных было предложено разработать усовершенствованные реляционные СУБД, которые смогли бы соединить в себе преимущества NoSQL-решений и классические реляционные системы с транзакциями ACID. В результате в 2011 году в глоссарии программистов появился очередной термин

NewSQL, он ознаменовал появление нового витка в спирали развития SQL-ориентированных БД.

Для того чтобы заставить новые СУБД работать в условиях распределенного хранения и при этом обеспечить высокую скорость и надежность, разработчики пошли на неординарный шаг – они приняли решение обеспечить хранение данных не на медленных жестких дисках, а непосредственно в ОЗУ. За счет того, что NewSQL разворачивается в кластере, содержащем несколько узлов, выход из строя одного-двух узлов не является критичным. Кроме того, NewSQL предполагает регулярное резервное копирование данных на жесткие диски, что также повышает уровень надежности.

Пока рано делать выводы о перспективах NewSQL (ведь это направление развития реляционной модели находится в самом начале пути), но уже сегодня существует несколько интересных разработок в этой области, в частности стоит отметить системы VoltDB и NuoDB.

Замечание

Примечательно, что создатели NuoDB (страничка в интернете <https://www.nuodb.com>) свой язык запросов назвали «Elastic SQL», тем самым подчеркнув, что их разработка сумела подстроиться к работе с большими данными, сохранив преимущества традиционных реляционных СУБД.

Добыча данных

Данные, какими бы они не были большими, не стоят ровным счетом ничего, пока из них не извлечены полезные для нас знания. Другими словами, большие данные можно сравнить с огромной библиотекой, наполненной книгами на неизвестном нам языке. Для того чтобы библиотека стала нам полезной, надо научиться читать. Именно поэтому сразу с появлением термина «Big Data» родился еще один – «Data Mining».

Определение

Добыча данных (Data Mining) – это процесс обнаружения в низкоуровневых не-систематизированных данных ранее неизвестных полезных знаний, необходимых для принятия решений в различных сферах человеческой деятельности.

Специалисты по большим данным отмечают, что добыча данных имеет мало общего со стандартной бизнес-аналитикой, где простые арифметические операции над известными значениями приносят ре-

зультат. Так, в руководстве архитектора Oracle по большим данным [58] отмечается, что при работе с большими данными результат получается в процессе их очистки в ходе построения последовательности моделей: сначала выдвигается гипотеза, а затем строится статистическая, визуальная или семантическая модель. Удостоверившись в верности изначальной гипотезы, аналитиком выдвигается очередная. Этот процесс требует от аналитика либо интерпретации визуальных значений или составления интерактивных запросов на основе знаний, либо разработки адаптивных алгоритмов машинного обучения, способных получить искомый результат.

Как видите, добыча данных относится к междисциплинарной области знаний [57]. С одной стороны, Data Mining опирается на уже знакомую читателю теорию баз данных, но, кроме этого, специалисту по добыче знаний понадобятся: алгоритмизация, эволюционное программирование, искусственный интеллект, нейронные сети, статистика, распознавание образов, методы визуализации данных и т. д. Ко всему прочему добыча данных нуждается в помощи наук предметной области, для которой извлекаются знания, а это может быть медицина, химия, география, экономика и т. д.

Таким образом, большие данные выступают опорным инструментальным средством для более высокоуровневой технологии – добычи данных в интересах лиц, принимающих решение при решении сложных управленческих задач.

Резюме

Большие данные – это одно из перспективных направлений развития информационных технологий, которое уже сегодня активно применяется многими коммерческими и государственными структурами.

Работа с технологией больших данных обычно осуществляется на основе лямбда-архитектуры.

Существует множество программных решений различных разработчиков, которые нацелены на работу с большими данными и способны работать на разных уровнях лямбда-архитектуры. Нам как специалистам в области БД стоит отметить следующее ПО:

- SQL-ориентированный инструментарий: Hive, Cassandra, Impala, Shark, Spark SQL, Drill;
- NoSQL-решения: MongoDB, HBase;
- NewSQL-системы: VoltDB, NuoDB и SAP HANA.

Важно понимать, что технология больших данных не всемогуща. Она не сможет извлечь данные, которые мы не просили, и не способна дать ответы на вопросы, которые не были заданы. Большие данные позволяют вам взглянуть на проблему по-новому, только нам надо научиться использовать их, поэтому в центре технологии с модным названием «Big Data» оказывается аналитик – человек, обладающий всеми необходимыми для этого компетенциями.

Вопросы для самопроверки

1. Какими характеристиками должны обладать большие данные?
2. Чем отличается вертикальное и горизонтальное масштабирование?
3. Какие принципы должны соблюдаться при работе с большими данными?
4. Прокомментируйте назначение уровней лямбда-архитектуры.
5. Для чего предназначен проект Apache Hadoop?
6. Для чего предназначен проект Apache Kafka?
7. Что послужило причиной появления NewSQL?
8. Что такое добыча данных (Data Mining)?

Глава 26

Составление программной документации

Настало время раскрыть читателю секрет Полишинеля – в разработке БД процесс чистого программирования занимает далеко не все отводимое на проект время. Нисколько не занижая вклад программистов в общее дело создания программного продукта, отметим, что зачастую гораздо больше ресурсов уходит на решение других, на первый взгляд менее очевидных задач. Одной из таких задач выступает документирование, а если говорить точнее – составление программной документации. И чтобы сразу заставить вас проникнуться значимостью темы, приведем всего одну цифру – считается, что в крупных проектах документирование способно «проглотить» до 25 % от общей его трудоемкости (как с точки зрения времени, так и по финансовым параметрам).

Для чего нужна программная документация? Для того чтобы устранить ряд противоречий и потенциальных проблем, которые могут возникнуть как внутри коллектива разработчиков, так и во взаимоотношениях между разработчиком и заказчиком.

Во-первых, в документации юридически закрепляются требования заказчика к подлежащему разработке продукту. Если этого не сделать, то вы рискуете при сдаче покупателю, казалось, готового продукта выяснить, что вы разработали совсем не то, что ожидал ваш клиент.

Во-вторых, документация должна обеспечивать отчуждаемость программного продукта от разработчиков. Другими словами, документация должна вестись так, чтобы ничто не помешало передать эстафету по разработке (сопровождению, рефакторингу и т. п.) базы данных от одного коллектива разработчиков другому. Это очень важно для всех сторон процесса. На минутку представьте, что вы руководитель проекта, а из вашего коллектива уходит один из ключевых программистов, на

котором буквально держалась вся система... В этом случае наличие грамотного описания всего проекта может спасти программный продукт в целом и вас в частности.

В-третьих, документация призвана обеспечить возможность освоения и эффективного применения программ пользователями. Неопытный пользователь (как собирательный образ всех лиц, принципиально не читающих инструкции) в два счета в состоянии привести к краху любую самую надежную систему, и только соответствующие строки в руководстве, где черным по белому будет написано «Никогда не делай это!», смогут спасти репутацию создателей БД, когда к ним будут предъявлены необоснованные претензии.

Список далеко не полон, но согласитесь, что и приведенных в нем причин вполне достаточно, дабы задуматься над вопросом составления программной документации. Поэтому у профессиональных разработчиков БД составление документации является неотъемлемым элементом жизненного цикла программного продукта. Очень важен тот факт, что это не единовременная акция! Напротив, документация выпускается (распространяется, сопровождается) на всех этапах цикла. В больших компаниях для этого даже создаются специальные подразделения со своим руководством и коллективом специалистов, осуществляющие планирование и общее управление процессом документирования. Они обязательно добьются того, чтобы свою лепту в создание документации внесли все задействованные в проекте разработчики БД.

Виды программных документов

К программным относят документы, содержащие сведения, необходимые для разработки, изготовления, сопровождения и эксплуатации программ. Перечень программных документов, подлежащих к выпуску во время разработки программного обеспечения, в России регулируется отдельным стандартом: ГОСТ 19.101–77 «Единая система программной документации (ЕСПД). Виды программ и программных документов».

Замечание

Пусть читателя не смущает тот факт, что ГОСТ 19.101–77 введен в действие в далеком 1980 г. Этот стандарт и все другие документы ЕСПД (см. приложение 3) на момент написания данных строк являются актуальными и обязательными к исполнению.

Буквально на первых страницах стандарта [7] читатель обнаружит классификацию программных документов (табл. 26.1).

Таблица 26.1. Виды программных документов

Вид программного документа	Содержание программного документа
Спецификация	Состав программы и документации на нее
Ведомость держателей подлинников	Перечень предприятий, на которых хранят подлинники программных документов
Текст программы	Запись программы с необходимыми комментариями
Описание программы	Сведения о логической структуре и функционировании программы
Программа и методика испытаний	Требования, подлежащие проверке при испытании программы, а также порядок и методы их контроля
Техническое задание	Назначение и область применения программы, технические, технико-экономические и специальные требования, предъявляемые к программе, необходимые стадии и сроки разработки, виды испытаний
Пояснительная записка	Схема алгоритма, общее описание алгоритма и (или) функционирования программы, а также обоснование принятых технических и технико-экономических решений
Эксплуатационные документы	Сведения для обеспечения функционирования и эксплуатации программы

В рамках одной главы мы не в состоянии обсудить все документы ЕСПД, но, по крайней мере, дадим краткую аннотацию наиболее важных из них.

Техническое задание

Вполне возможно, между разработчиком и заказчиком будущей БД сложились настолько доверительные отношения, что вместо подписания вороха документов и юридически обязывающих договоров им достаточно просто пожать друг другу руку. И это нехитрое действие на все 100 % исключит все возможные будущие недомолвки между настоящими джентльменами. Во всех остальных случаях вам придется потрудиться перенести на бумагу все устные договоренности о будущем проекте и скрепить их подписями. Одним из важнейших документов, без создания которого не имеет смысла переходить к практическим действиям по проектированию БД, выступает техническое задание.

Определение

Техническое задание – это документ, в котором закреплены назначение и область применения программы, технические, технико-экономические и специальные требования, предъявляемые к программе, необходимые стадии и сроки разработки, виды испытаний.

На техническое задание существует стандарт ГОСТ 19.201–78 «Техническое задание. Требования к содержанию и оформлению». В соответствии со стандартом документ должен содержать следующие разделы:

- «Введение» – здесь описывают наименование, краткую характеристику области применения программы или программного изделия и объекта, в котором используют программу или программное изделие;
- «Основание для разработки» – тут указываются: документы, на основании которых ведется разработка; организация, утвердившая этот документ, и дата его утверждения; наименование темы разработки;
- «Назначение разработки» – в этом разделе приводится функциональное и эксплуатационное назначение программы или программного изделия;
- «Требования к программе или программному изделию» – включает следующие подразделы:
 - требования к функциональным характеристикам;
 - требования к надежности;
 - условия эксплуатации;
 - требования к составу и параметрам технических средств;
 - требования к информационной и программной совместимости;
 - требования к маркировке и упаковке;
 - требования к транспортированию и хранению;
 - специальные требования;
- «Требования к программной документации» – тут приводится предварительный состав программной документации и, при необходимости, специальные требования к ней;
- «Технико-экономические показатели» – здесь должны быть указаны: ориентировочная экономическая эффективность, предполагаемая годовая потребность, экономические преимущества

разработки по сравнению с лучшими отечественными и зарубежными образцами или аналогами;

- «Стадии и этапы разработки» – устанавливает необходимые стадии разработки, этапы и содержание работ, а также сроки разработки и определяют исполнителей;
- «Порядок контроля и приемки» – в этом разделе конкретизируются виды испытаний и общие требования к приемке работы.

Кроме того, стандарт не запрещает дополнять техническое задание приложениями.

Внимание!

В процессе создания технического задания должны принять участие обе заинтересованные стороны – заказчик БД и разработчик БД.

Пояснительная записка

Пояснительная записка должна содержать все сведения, необходимые для сопровождения и модификации программного обеспечения: сведения о его структуре и конкретных компонентах, общее описание алгоритмов и их схемы, а также обоснование принятых технических и технико-экономических решений.

Внимание!

Пояснительная записка должна содержать все сведения, достаточные для отчуждения программного продукта, т. е. обеспечить безболезненную передачу проекта от одного коллектива разработчиков другому.

По стандарту (ГОСТ 19.404–79) содержание пояснительной записки должно включать следующие разделы:

- «Введение» – здесь указывают наименование программы и (или) условное обозначение темы разработки, а также документы, на основании которых ведется разработка, с указанием организации и даты утверждения;
- «Назначение и область применения» – тут указывают назначение программы, краткую характеристику области применения программы;
- «Технические характеристики» – состоит из подразделов:

- постановка задачи на разработку программы, описание применяемых математических методов и, при необходимости, описание допущений и ограничений, связанных с выбранным математическим аппаратом;
 - описание алгоритма и (или) функционирования программы с обоснованием выбора схемы алгоритма решения задачи, возможные взаимодействия программы с другими программами;
 - описание и обоснование выбора метода организации входных и выходных данных;
 - описание и обоснование выбора состава технических и программных средств на основании проведенных расчетов и (или) анализов, распределение носителей данных, которые использует программа;
- «Ожидаемые технико-экономические показатели» – здесь указывают технико-экономические показатели, обосновывающие преимущество выбранного варианта технического решения, а также, при необходимости, ожидаемые оперативные показатели;
- «Источники, использованные при разработке» – включает перечень научно-технических публикаций, нормативно-технических документов и других научно-технических материалов, на которые есть ссылки в основном тексте.

В приложение к документу могут быть включены таблицы, обоснования, методики, расчеты и другие документы, использованные при разработке. В частности, для проектов БД в приложениях обязательно должны оказаться схемы данных, определения ограничений доменов, спецификация таблиц, индексов и т. п.

Эксплуатационные документы

Вновь возвратимся к ГОСТ 19.101–77, регламентирующему виды программных документов. Согласно этому стандарту, разработке подлежат виды эксплуатационных документов, представленные в табл. 26.2.

На взгляд автора, в случае разработки эксплуатационной документации для БД вместо руководства системного программиста документ корректнее назвать «Руководство администратора баз данных». Пожалуй, в этом не будет существенной ошибки, хотя, конечно же, самовольное изменение названий можно трактовать как отклонение от стандарта.

Еще одно замечание относительно документа с описанием языка. Подобный эксплуатационный документ актуален только в ситуации, когда программный продукт разработан на экзотическом языке программирования, если же в своем проекте вы опирались на стандартизированные языки (например, SQL или C++), то вряд ли есть смысл создавать описание с текстом из учебника по программированию.

Таблица 26.2. Виды эксплуатационных документов

Вид эксплуатационного документа	Содержание эксплуатационного документа
Ведомость эксплуатационных документов	Перечень эксплуатационных документов на программу
Формуляр	Основные характеристики программы, комплектность и сведения об эксплуатации программы
Описание применения	Сведения о назначении программы, области применения, применяемых методах, классе решаемых задач, ограничениях для применения, минимальной конфигурации технических средств
Руководство системного программиста	Сведения для проверки, обеспечения функционирования и настройки программы на условия конкретного применения
Руководство программиста	Сведения для эксплуатации программы
Руководство оператора	Сведения для обеспечения процедуры общения оператора с вычислительной системой в процессе выполнения программы
Описание языка	Описание синтаксиса и семантики языка
Руководство по техническому обслуживанию	Сведения для применения тестовых и диагностических программ при обслуживании технических средств

Руководство системного программиста

В соответствии с требованиями ГОСТ 19.503–79 руководство системного программиста (в нашем случае речь ведется об администраторе баз данных предприятия, на котором будет развернута БД) должно содержать всю информацию, необходимую для установки программного обеспечения, его настройки и проверки работоспособности.

В руководство включаются следующие разделы:

- «Общие сведения о программном продукте». Здесь указывается общее назначение и функции программы, сведения о технических и программных средствах, обеспечивающих выполнение данной программы;
- «Структура программы». В разделе приводятся сведения о структуре программы, ее составных частях, о связях между составными частями и связи с другими программами;
- «Настройка программы». Содержит описание действий по настройке программы на условия конкретного применения;
- «Проверка программы». Описание способов проверки, позволяющих дать общее заключение о работоспособности программы;
- «Дополнительные возможности». Описание дополнительных разделов функциональных возможностей программы и способов их выбора;
- «Сообщения системному программисту». Здесь указываются тексты сообщений, выдаваемых в ходе выполнения настройки, проверки программы, а также в ходе выполнения программы, описание их содержания и действий, которые необходимо предпринять по этим сообщениям.

Руководство оператора

Руководство оператора, или, как иногда говорят, «руководство пользователя», – едва ли не самый главный и одновременно самый редко изучаемый пользователями документ из пачки эксплуатационных документов. Состав руководства регламентируется ГОСТ 19.505–79 и должен содержать следующие разделы:

- «Назначение программы». Раздел содержит сведения о назначении программы и информацию, достаточную для понимания функций программы и ее эксплуатации;
- «Условия выполнения программы». Содержит условия, необходимые для выполнения программы;
- «Выполнение программы». Последовательность действий оператора, обеспечивающих загрузку, запуск, выполнение и завершение программы, приведено описание функций, формата и возможных вариантов команд, с помощью которых оператор осуществляет загрузки и управляет выполнением программы, а также ответы программы на эти команды;

- «Сообщения оператору». Тексты сообщений, выдаваемых в ходе выполнения программы, описание их содержания и соответствующие действия оператора.

Очевидно, что в проектах БД руководство оператора разрабатывается для клиентских приложений, с которыми станет работать персонал предприятия. Более сложные элементы, связанные с эксплуатацией БД, выносятся в руководство системного программиста и руководство программиста.

Документация в тексте программы

В компаниях, занимающихся проектированием ПО, и просто в профессиональных командах разработчиков от программистов требуют комментировать создаваемый ими исходный код. Это очень важное требование, соблюдение которого существенно упростит командную работу над проектом, но при одном условии – программисты должны знать, как комментировать свои листинги.

Попробуем дать несколько советов и в этой области, а для этого сначала приведем классификацию комментариев [28]:

- 1) юридические комментарии, например заявление об авторских правах разработчиков;
- 2) информативные, самый распространенный тип комментариев, содержащий пояснения к коду;
- 3) представление намерений, в которых программист описывает, что именно он рассчитывает получить от комментируемого участка кода;
- 4) пояснения, когда смысл какой-то операции стараются представить в удобочитаемой форме;
- 5) предупреждения о последствиях, применяются для предупреждения других программистов о нежелательных последствиях от каких-либо действий;
- 6) комментарии о будущих действиях, например вы оставили в коде только объявление функции, а её реализацией займетесь позднее;
- 7) комментарии усиления внимания, ставящие акцент на определенном участке кода, который на первый взгляд кажется не столь важным.

Таким образом, перед тем как приступить к комментированию исходного кода, убедитесь, что ваши пояснения относятся к той или иной

объективно необходимой категории комментариев. Если это не так, то, скорее всего, предлагаемый вами текст окажется ненужным. Приведем ещё ряд ошибок, часто возникающих при документировании исходного кода:

- 1) недостоверные комментарии, когда исходный код и соответствующие ему пояснения противоречат друг другу. Такая ситуация часто возникает при модификации кода – программист нашел и исправил ошибку в листинге, но забыл внести изменения в комментарий;
- 2) избыточный комментарий, когда чтение пояснительного текста занимает больше времени, чем изучение кода. Совсем необязательно комментировать каждую строку кода, вместо этого следует писать такой код, чтобы его смысл был прост и доступен;
- 3) неочевидный комментарий, когда связь между текстом и кодом весьма относительна, если отсутствует вовсе;
- 4) комментарии, утверждающие очевидное и не представляющие никакой полезной информации.

Если у читателя имеется опыт разработки программного обеспечения, то он наверняка замечал одну примечательную деталь – чем более путанный код у нас получается, тем он быстрее обрастает все более туманными комментариями. Исходя из этого наблюдения, сформулируем главное правило документирующего свой исходный код разработчика: рациональный и читабельный код с минимальными пояснениями гораздо лучше громоздкого, сложного кода с большим количеством комментариев. В этом контексте нельзя не упомянуть совет Брайна Кернигана: «Не комментируйте плохой код – перепишите его».

Резюме

Процесс составления программной документации выступает неотъемлемой частью работы над программным продуктом и продолжается на всем протяжении жизненного цикла программного обеспечения.

Целью документирования является обеспечение эффективного взаимодействия разных специалистов в производственной цепочке: заказчик БД – разработчик БД – прикладные программисты – конечный пользователь.

Рассмотренные в главе стандарты регламентируют состав и содержание программной документации, но не определяют порядок оформле-

ния материалов. Поэтому при компоновке текстового и графического материала разработчикам программного обеспечения целесообразно руководствоваться ГОСТ 7.32–2001 «Отчет о научно-исследовательской работе. Структура и правила оформления».

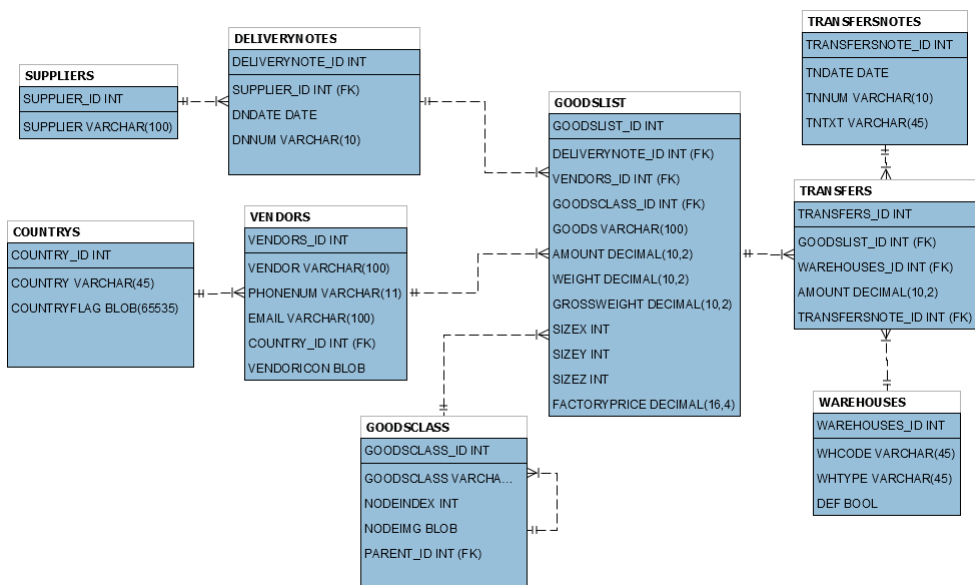
Завершая тему, напомним важное правило, которым должен руководствоваться разработчик документации: составлять документ так, чтобы он был понятен читателю (пользователю программного продукта).

Вопросы для самопроверки

1. Для чего нужна программная документация?
2. Что понимается под возможностью отторжения программного продукта?
3. Перечислите виды программной документации.
4. Для чего необходимо техническое задание?
5. Что должно быть описано в пояснительной записке?
6. Что должно содержать руководство оператора?
7. Какие разновидности комментариев в тексте программ вам известны?

Приложение 1

Модель БД «Склад»



Приложение 2

Пример XML-схемы

Листинг П2.1. XML-схема описания системного блока компьютера

```
<?xml version="1.0"?>
<xs:schema xmlns="http://tempuri.org/computers"
  elementFormDefault="qualified"
  targetNamespace="http://tempuri.org/computers"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="COMPUTER">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="ComputerManufacturer"/>
        <xs:element ref="CPU" />
        <xs:element ref="MOTHEBOARD"/>
        <xs:element minOccurs="1" maxOccurs="8" ref="HDD" />
        <xs:element ref="VIDEO" />
        <xs:element ref="RAM" />
        <xs:element minOccurs="0" maxOccurs="unbounded" ref="OTHER" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="ComputerManufacturer">
    <xs:simpleType>
      <xs:restriction base="xs:string">
        <xs:enumeration value="HP"/>
        <xs:enumeration value="IBM"/>
        <xs:enumeration value="Toshiba"/>
        <xs:enumeration value="Другой производитель"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:element>
</xs:schema>
```



```
        </xs:restriction>
    </xs:simpleType>
</xs:element>

<xs:element name="CPU">
    <xs:complexType>
        <xs:attribute name="Manufacturer" type="xs:string" use="required"/>
        <xs:attribute name="CoreCount" type="xs:byte" default="1"/>
        <xs:attribute name="Frequency" type="xs:positiveInteger" use="required"/>
        <xs:attribute name="Socet" type="xs:string" use="required"/>
        <xs:attribute name="Notes" type="xs:string" use="optional"/>
    </xs:complexType>
</xs:element>

<xs:element name="MOTHEBOARD">
    <xs:complexType>
        <xs:attribute name="Manufacturer" type="xs:string" use="required"/>
        <xs:attribute name="Model" type="xs:string" use="required"/>
        <xs:attribute name="Socet" type="xs:string" use="required"/>
    </xs:complexType>
</xs:element>

<xs:element name="HDD">
    <xs:complexType>
        <xs:attribute name="Manufacturer" type="xs:string" use="required"/>
        <xs:attribute name="Size" type="xs:positiveInteger" use="required"/>
        <xs:attribute name="Speed" type="xs:positiveInteger" default="7200"/>
    </xs:complexType>
</xs:element>

<xs:element name="VIDEO">
    <xs:complexType>
        <xs:attribute name="Manufacturer" type="xs:string" use="required"/>
        <xs:attribute name="Model" type="xs:string" use="required"/>
        <xs:attribute name="RAMSize" type="xs:positiveInteger" use="required"/>
    </xs:complexType>
</xs:element>

<xs:element name="RAM">
    <xs:complexType>
```

```

    <xs:attribute name="Manufacturer" type="xs:string" use="required"/>
    <xs:attribute name="Size" type="xs:positiveInteger" use="required"/>
    <xs:attribute name="Speed" type="xs:positiveInteger"/>
  </xs:complexType>
</xs:element>

<xs:element name="OTHER" type="xs:string" />
</xs:schema>

```

Листинг П2.2. Документ computers.xml, использующий схему computers.xsd

```

<?xml version="1.0" encoding="utf-8"?>
<COMPUTERS xmlns="http://www.myurl.ru/computers"
  xmlns:xs="http://www.w3.org/2001/XMLSchema-instance"
  xs:noNamespaceSchemaLocation="computers.xsd">
  <COMPUTER>
    <ComputerManufacturer>IBM</ComputerManufacturer>
    <CPU>
      <Manufacturer>Intel</Manufacturer>
      <CoreCount>2</CoreCount>
      <Frequency>2700</Frequency>
      <Socet>P775</Socet>
    </CPU>
    <MOTHEBOARD>
      <Manufacturer>ASUS</Manufacturer>
      <Model>GT400</Model>
      <Socet>P775</Socet>
    </MOTHEBOARD>
    <HDD>
      <Manufacturer>WesternDigital</Manufacturer>
      <Size>320</Size>
      <Speed>7200</Speed>
    </HDD>
    <HDD>
      <Manufacturer>Fujitsu</Manufacturer>
      <Size>500</Size>
      <Speed>7200</Speed>
    </HDD>
    <VIDEO>

```

```
<Manufacturer>NVideo</Manufacturer>
<Model>Unknown</Model>
<RAMSize>512</RAMSize>
</VIDEO>
<RAM>
  <Manufacturer>Samsung</Manufacturer>
  <Size>4</Size>
  <Speed>800</Speed>
</RAM>
<OTHER>TVTuner AVER</OTHER>
<OTHER>DVD-RW</OTHER>
</COMPUTER>
</COMPUTERS>
```

Приложение 3

Стандарты по единой системе программной документации

1. ГОСТ 19.004–80 Единая система программной документации. Термины и определения.
2. ГОСТ 19.101–77 Единая система программной документации. Виды программ и программных документов.
3. ГОСТ 19.102–77 Единая система программной документации. Стадии разработки.
4. ГОСТ 19.105–78 Единая система программной документации. Общие требования к программным документам.
5. ГОСТ 19.106–78 Единая система программной документации. Требования к программным документам, выполненным печатным способом.
6. ГОСТ 19.201–78 Единая система программной документации. Техническое задание. Требования к содержанию и оформлению.
7. ГОСТ 19.202–78 Единая система программной документации. Спецификация. Требования к содержанию и оформлению.
8. ГОСТ 19.301–79 Единая система программной документации. Программа и методика испытаний. Требования к содержанию и оформлению.
9. ГОСТ 19.401–78 Единая система программной документации. Текст программы. Требования к содержанию и оформлению.
10. ГОСТ 19.402–78 Единая система программной документации. Описание программы.
11. ГОСТ 19.403–79 Единая система программной документации. Ведомость держателей подлинников.

12. ГОСТ 19.404–79 Единая система программной документации. Пояснительная записка. Требования к содержанию и оформлению.
13. ГОСТ 19.501–78 Единая система программной документации. Формуляр. Требования к содержанию и оформлению.
14. ГОСТ 19.502–78 Единая система программной документации. Описание применения. Требования к содержанию и оформлению.
15. ГОСТ 19.503–79 Единая система программной документации. Руководство системного программиста. Требования к содержанию и оформлению.
16. ГОСТ 19.504–79 Единая система программной документации. Руководство программиста. Требования к содержанию и оформлению.
17. ГОСТ 19.505–79 Единая система программной документации. Руководство оператора. Требования к содержанию и оформлению.
18. ГОСТ 19.506–79 Единая система программной документации. Описание языка. Требования к содержанию и оформлению.
19. ГОСТ 19.507–79 Единая система программной документации. Ведомость эксплуатационных документов.
20. ГОСТ 19.508–79 Единая система программной документации. Руководство по техническому обслуживанию. Требования к содержанию и оформлению.

Список литературы

1. *Adam Shook, Donald Miner*. MapReduce Design Patterns. O'Reilly Media, Inc. 2012. 256 p.
2. *C. Beeri*. A formal approach to object-oriented databases. Data & Knowledge Engineering, vol. 5 (1990), p. 353–382.
3. *Malcolm Atkinson, Francois Bancilhon, David DeWitt, Klaus Dittrich, David Maier, Stanley Zdonik*. The Object-Oriented Database System Manifesto. Proc. 1st International Conference on Deductive and Object-Oriented Databases, Kyoto, Japan (1989). New York, N. Y.: Elsevier Science (1990).
4. *Jeffrey Dean, Sanjay Ghemawat*. MapReduce: Simplified Data Processing on Large Clusters. OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA (2004), p. 137–150.
5. The Object Data Standard: ODMG 3.0. Edited by R. G. G. Cattell and Douglas K. Barry, with contributions by Mark Berler, Jeff Eastman, David Jordan, Craig L. Russell, Olaf Schadow, Torsten Stanienda, and Fernando Velez. Morgan Kaufmann Publishers, Inc., 2000. ISBN 1-55860-647-5.
6. ГОСТ 34.321–96. Информационные технологии. Система стандартов по базам данных. Эталонная модель управления данными.
7. ГОСТ 19.101–77 Единая система программной документации (ЕСПД). Виды программ и программных документов.
8. ГОСТ 19.201–78. Техническое задание. Требования к содержанию и оформлению.
9. *Аткинсон Леон*. MySQL. Библиотека профессионала / пер. с англ. М.: Вильямс, 2002. 624 с.
10. *Браст Эндрю Дж., Форте Стивен*. Разработка приложений на основе Microsoft SQL Server 2005. Мастер-класс / пер. с англ. М.: Русская редакция, 2007. 880 с.
11. *Брауде Э.* Технология разработки программного обеспечения. СПб.: Питер, 2004– 655с.: ил.
12. *Вигерс Карл*. Разработка требований к программному обеспечению / пер. с англ. М.: Русская редакция, 2004. 576 с.

13. Гагарина Л. Г., Кокорева Е. В., Виснадул Б. Д. Технология разработки программного обеспечения: учеб. пособие / под ред. Л. Г. Гагариной. М.: ИД «ФОРУМ»: ИНФРА-М, 2008. 400 с: ил. (Высшее образование).
14. Гарольд Э., Минс С. XML: справочник / пер. с англ. СПб.: Символ-Плюс, 2002. 576 с.
15. Гарсия-Молина Гектор, Ульман Джеффри Д и Уидом Дженнифер. Системы баз данных. Полный курс / пер. с англ. М.: Вильямс, 2003. 1088 с.: ил.
16. Гради Буч. Объектно-ориентированный анализ и проектирование с примерами приложений / пер. с англ. М.: Вильямс, 2010. 720 с.
17. Грофф Дж., Вайнберг П. SQL: Полное руководство / пер. с англ. 2-е изд., перераб. и доп. К.: Издательская группа BHV, 2001. 816 с.
18. Дейв Энсор, Йен Стивенсон. Oracle. Проектирование баз данных / пер. с англ. К.: Издательская группа BHV, 1999. 560 с.
19. Дейт К. Дж. Введение в системы баз данных. 8-е изд. / пер. с англ. М.: Вильямс, 2006. 1328 с.
20. Дейт К. Дж. SQL и реляционная теория. Как грамотно писать код на SQL / пер. с англ. СПб.: Символ-Плюс, 2010. 480 с., ил.
21. Иванова Г. С. Технология программирования: учебник. М.: КНО-РУС, 2011. 336 с.
22. Кайт Том. Oracle для профессионалов / пер. с англ. Том Кайт. СПб.: ООО «ДиаСофтЮП», 2003. 672 с.
23. Кайл Бэнкер. MongoDB в действии / пер. с англ. А. А. Слинкина. М.: ДМК Пресс, 2012. 394 с: ил.
24. Клайн К. SQL: справочник. 2-е изд. / пер. с англ. М.: КУДИЦ-ОБРАЗ, 2006. 832 с.
25. Коннолли Томас, Бегг Каролин, Страчан Анна. Базы данных: проектирование, реализация и сопровождение. Теория и практика, 2-е изд. / пер. с англ. М.: Вильямс, 2001. 1120 с.
26. Константайн Л., Локвуд Л. Разработка программного обеспечения. СПб.: Питер, 2004. 592 с.: ил. (Серия «Классика computer science»).
27. Кузнецов С. Д. Основы баз данных: учеб. пособие. 2-е изд. М.: Интернет-университет информационных технологий; БИНОМ. Лаборатория знаний, 2007. 484 с.: ил.

28. *Мартин Р.* Чистый код: создание, анализ и рефакторинг. Библиотека программиста. СПб.: Питер, 2013. 464 с.: ил. (Серия «Библиотека программиста»).
29. *Марков А. С., Лисовский К. Ю.* Базы данных. Введение в теорию и методологию: учебник. М.: Финансы и статистика, 2006. 512 с.: ил.
30. *Молинаро Э.* SQL: сборник рецептов / пер. с англ. СПб.: Символ-Плюс, 2009. 672 с.
31. *Нильсен Пол.* Microsoft SQL Server 2005. Библия пользователя / пер. с англ. М.: ООО «И. Д. Вильямс», 2008. 1232 с.: ил.
32. *Олифер В., Олифер Н.* Компьютерные сети. Принципы, технологии, протоколы: учеб. для вузов. 5-е изд. СПб.: Питер, 2016. 992 с.: ил. (Серия «Учебник для вузов»).
33. *Осипов Д.* Delphi. Профессиональное программирование. СПб.: Символ-Плюс, 2006. 1056 с.
34. *Осипов Д. Л.* Базы данных и Delphi. Теория и практика. СПб.: БХВ-Петербург, 2011. 752 с.: ил.
35. *Осипов Д. Л.* InterBase и Delphi. Клиент-серверные базы данных. М.: ДМК Пресс, 2015. 536 с: ил.
36. *Осипов Д. Л.* MySQL и Delphi. Базы данных. Технология FireDAC. [б. м.]: Издательские решения, 2017. 624 с.
37. *Райордан Р.* Основы реляционных баз данных / пер. с англ. М.: Русская редакция, 2001. 384 с.: ил.
38. *Роб П., Коронел К.* Системы баз данных: проектирование, реализация и управление. 5-е изд., перераб. и доп. / пер с англ. СПб.: БХВ-Петербург, 2004. 1040 с.: ил.
39. *Седжвик Р.* Фундаментальные алгоритмы на C++. Анализ / Структуры данных / Сортировка / Поиск / пер. с англ. Р. Седжвик. К.: ДиаСофт, 2001. 688 с.
40. *Скляр А. Я.* Введение в InterBase. М.: Горячая линия – Телеком, 2002. 517 с.
41. *Смирнов С. Н.* Безопасность систем баз данных. М.: Гелиос АРВ, 2007. 352 с.: ил.
42. *Стоунбрейкер М.* Объектно-реляционные системы баз данных / Открытые системы. 1994. № 4.
43. *Тамре Л.* Введение в тестирование программного обеспечения / пер. с англ. М.: Вильямс, 2003. 368 с.

44. Теория и практика построения баз данных. 8-е изд. Д. Крэнке. СПб.: Питер, 2003. 800 с.: ил. (Серия «Классика computers science»).
45. Уорсли Дж., Дрейк Дж. PostgreSQL. Для профессионалов / пер. с англ. СПб.: Питер, 2003. 496 с.
46. Фицджеральд Майкл. Регулярные выражения: основы / пер. с англ. М.: ООО «И. Д. Вильямс», 2015. 144 с.: ил. Парал. тит. англ.
47. Фуфаев Э. В. Базы данных: учеб. пособие для студ. учреждений сред. проф. образования / Э. В. Фуфаев, Д. Э. Фуфаев. 7-е изд., стер. М.: Академия, 2012. 320 с.
48. Хантер Дэвид, Рафтер Джефф, Фаусетт Джо, ван дер Влист Эрик и др. XML. Базовый курс. 4-е изд. / пер. с англ. М.: ООО «И.Д. Вильямс», 2009. 1344 с.
49. Ховард М., Лебланк Д. Защищенный код / пер. с англ. 2-е изд., испр. М.: Русская редакция, 2004. 704 с.
50. Хомоненко А. Д., Цыганков В. М., Мальцев М. Г. Базы данных: учеб. для выс. учеб. заведений / под. ред. А. Д. Хомоненко. 4-е изд., доп. и перераб. СПб.: КОРОНА принт, 2004. 736 с.
51. Шаньгин В. Ф. Защита компьютерной информации. Эффективные методы и средства. М.: ДМК Пресс, 2008. 544 с.
52. Щеглов А. Ю. Защита компьютерной информации от несанкционированного доступа. СПб.: Наука и техника, 2004. 384 с.
53. [Электронный ресурс] <https://www.iso.org/ics/35.060/x/>: International Organization for Standardization. 35.060 – Languages used in information technology.
54. [Электронный ресурс] <https://msdn.microsoft.com/ru-ru/library/thzzea08.aspx>: Основы ODBC, Visual Studio 2015.
55. [Электронный ресурс] <https://www.osp.ru/os/2012/02/13014107/>: Леонид Черняк. Смутное время СУБД // Открытые системы. 2012. № 2.
56. [Электронный ресурс] <https://jsman.ru/mongo-book/index.html>: Karl Seguin. The Little MongoDB Book.
57. [Электронный ресурс] <http://www.mmds.org/>: Jure Leskovec, Anand Rajaraman, Jeffrey D. Ullman. Mining of Massive Datasets.
58. [Электронный ресурс] <https://www.oracle.com/technetwork/topics/entarch/articles/oea-big-data-guide-1522052.pdf>: An Enterprise Architect's Guide to Big Data Reference Architecture Overview, Oracle enterprise architecture white paper, march 2016.

Предметный указатель

A

ADO .NET [400](#)
Apache Hadoop [463](#)
Application Server [395](#)

B

Big data [458](#)
 правило 3V [460](#)

C

CASE-проектирование [128](#)
CORBA [53](#)

D

Data Mining [468](#)
DBMS [41](#)
Distributed database [414](#)
DTD
 определение типа документа [367](#)
 типы атрибутов [369](#)

E

ER-модель
 Crows Foot model [127](#)
 IDEF [128](#)
 IDEF1X [128](#)
 атрибут [114](#)
 мощность связи [120](#)
 подтип сущности [117](#)
 рекурсивная связь [124](#)
 связь [120](#)
 сильная связь [123](#)
 слабая связь [123](#)
 тип сущности [114](#)

F

FireDAC [402](#)

J

JDBC [404](#)

M

MapReduce [452](#)
MDX [357](#)

N

NewSQL [467](#)
NoSQL [449](#)
NULL [78](#)

O

Object Data Management Group [437](#)
Object Definition Language [437](#)
Object Query Language [437](#)
ODBC [399](#)
OLAP [350](#)
 куб [355](#)
 модели хранения данных [353](#)
 тест FASMI [352](#)
 требования к инструментарию [351](#)
 язык многомерных выражений [356](#)
Open Systems Interconnection [390](#)

S

SQL
 Add [264](#)
 ALL [231](#)
 Alter [264](#)
 ALTER INDEX [267](#)

- ALTER SCHEMA 254
- ALTER TABLE 264
- ANY 231
- AVG 232
- Begin...end 281
- Between 226
- CASE 282
- Check 257
- CLOSE 305
- Collate 256
- COMMIT 337
- COUNT 232
- CREATE DATABASE 253
- CREATE DOMAIN 255
- CREATE FUNCTION 292
- CREATE INDEX 266
- CREATE PROCEDURE 288
- CREATE ROLE 344
- CREATE SCHEMA 254
- CREATE TABLE 256
- CREATE TRIGGER 296
- CREATE TYPE 208
- CREATE VIEW 270
- DECLARE CURSOR 303
- Default 256
- DELETE 248
- DISTINCT 221
- DROP DATABASE 254
- DROP INDEX 268
- DROP SCHEMA 255
- DROP VIEW 273
- EXISTS 230
- FETCH 304
- Foreign Key 257
- GEN_ID 300
- Global 263
- Grant 344
- GRANT 345
- Group By 233
- Having 234
- IF..THEN..ELSE 281
- IN 229, 231
- INSERT 244
- LIKE 226
- LOOP 288
- MAX 233
- MERGE 249
- MIN 233
- OPEN 304
- Order By 223
- Primary Key 257
- References 257
- REPEAT 285
- REVOKE 347
- ROLLBACK 337
- SELECT 220
- SELECT...JOIN 237
- SELECT...JOIN ... USING 239
- SELECT...NATURAL JOIN 239
- SOME 231
- SUM 233
- UNION 243
- Unique 257
- UPDATE 246
- Where 224
- WHERE 235
- WHILE 284
- арифметические операторы 213
- битовая последовательность 203
- внешнее соединение 237
- внутреннее соединение 237
- встроенные функции 216
- конкатенация строк 216
- логические операторы 214
- массив 206
- операторы сравнения 215
- операция присваивания 213
- переменные 278
- преобразование типов данных 210
- приближённые числовые типы данных 201
- проверка неопределённости NULL 215
- символьные строки 202

составной оператор 281
 тип данных дата и время 204
 точные числовые типы данных 200
 условный оператор 281
 циклы 284

Т

The Lambda Architecture 462

Х

XML

атрибут документа 363
 документ 361
 корректность документа 361
 пространство имен 364, 374
 тип данных 377
 элемент документа 362

XML-схема

типы данных 377
 элементы схемы 376

А

Абстракция 430
 Авторизация 184
 Администратор базы данных 60
 Администратор данных 59
 Аномалия вставки 133
 Аномалия редактирования 134
 Аномалия удаления 133
 Архитектура
 клиент-сервер 50, 389
 распределенная система 54, 412
 файл-сервер 49
 Архитектура ANSI/X3/SPARC 33
 Атрибут 67
 Аудит событий безопасности 189
 Аутентификация 184

Б

База данных 23
 Безопасность данных 176

Большие данные 460

В

Восстановление данных 187
 Выбор СУБД 104

Д

Данные 16
 Двухуровневая модель хранения данных 151
 Децентрализованные БД 412
 Добыча данных 468
 Документ-ориентированная модель 446
 Домен 69, 255

Ж

Жизненный цикл БД 94
 Журнал транзакций 159, 323

З

Защищенная база данных 176

И

Идентификация 184
 Избирательный доступ 185
 Индекс 266
 В-дерево 168
 битовый 172
 внешнего ключа 163
 избирательность 173
 первичного ключа 163
 пользовательский 163
 хеширование 164
 Инкапсуляция 431
 Интерактивная аналитическая обработка 350
 Интерфейс 392
 Интерфейс клиентского ПО 406
 Информационная система 90

К

Клиент
 толстый 53, 395

тонкий 52, 395

Клиент-серверные СУБД

 модели распределения функций 394

Ключ

 внешний 77

 первичный 77, 140

 потенциальный 140

Ключ-значение 447

Компьютерные меры защиты БД 183

Криптографическая защита 185

Курсор 302

Л

Лямбда-архитектура 462

М

Мандатный доступ 185

Масштабируемость

 вертикальная 453

 горизонтальная 453

Метод объекта 428

Многоуровневые БД 53

Модель взаимодействия открытых систем 390

Модель данных

 документ-ориентированная 448, 38

 иерархическая 27

 объектно-ориентированная 425, 37

 реляционная 65, 34

 сетевая 30

Модульность 431

Н

Наследование 432

Нормализация

 1NF 137

 2NF 141

 3NF 142

 4NF 146

 5NF 147

 NF Бойса-Кодда 145

 доменно-ключевая НФ 149

О

Объектно-ориентированные БД 425

Объектный язык запросов 437

Организационные меры защиты БД 182

Отладка 109

Отношение 73

П

Параллельность 326

Полиморфизм 432

Пользователь БД 62

Пояснительная записка 475

Представление данных

 блоки 157

 записи 156

 поля 154

 файлы 157

Представление объектов в памяти 158

Представление реляционных данных в памяти 153

Представления 269

Прикладной программист 61

Проблема совместного доступа

 неактуальные чтения 325

 неповторяемые чтения 326

 потерянные обновления 325

 чтение строк-фантомов 326

Программная документация 471

Проектирование БД 100

Протокол 392

Р

Разработчик БД 61

Распределение 419

Распределенная БД 414

Распределенная обработка 452

Распределенная файловая система 464

Регулярные выражения 311

 REGEXP 311

 запрос SQL 319

 метасимволы 312

 модификаторы повторений 315

Резервная копия 188
Реляционная алгебра 81
Реляционная модель 34
Реляционная таблица 73
Репликация 419, 455
Ролевой доступ 343
Руководство оператора 478

С

Связь 71
Сегментирование 454
Сервер приложений 53, 395
Система управления базами данных 41
Система управления распределенной БД 414
Системный каталог 22, 48
Системы, основанные на файлах 17
СУБД 41
 компоненты 45
 функции 42
СУРБД 55, 414
Сущность 67

Т

Таблица 256
Тестирование 108
Техническое задание 473
Технология доступа к данным
 ADO.NET 400
 FireDAC 402
 JDBC 404
 ODBC 399
Технология программирования 89
Тип данных 69
Тип сущности 67
Транзакции
 взаимная блокировка 329
 детализация уровня блокировок 333
 метод блокировок 328
 метод временных меток 329
 метод двухфазной блокировки 330
 многоверсионность 331

 неявное управление 335
 оптимистический подход 330
 пессимистический подход 328
 состояния 324
 типы транзакций 322
 требования ACID 323
 уровни изоляции 334
 явное управление 335
Триггер 294

У

Угрозы БД 178

Ф

Фрагментация 417
Функциональная зависимость атрибутов
 многозначная 138
 полная 138
 транзитивная 138
 частичная 138

Х

Хранилище данных 353
Хранимая процедура 288

Ц

Целостность
 данных 77
 доменов 78
 корпоративная 81
 ссылочная 80
 сущностей 79
Централизованные БД 52, 393
Цикл преобразования информации 91

Э

Эксплуатационные документы 476

Книги издательства «ДМК Пресс» можно заказать в торгово-издательском холдинге «Планета Альянс» наложенным платежом, выслав открытку или письмо по почтовому адресу: **115487, г. Москва, 2-й Нагатинский пр-д, д. 6А.**

При оформлении заказа следует указать адрес (полностью), по которому должны быть высланы книги; фамилию, имя и отчество получателя. Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в интернет-магазине: **www.a-planeta.ru.**

Оптовые закупки: тел. **+7(499) 782-38-89.**

Электронный адрес: **books@aliants-kniga.ru.**

Осипов Дмитрий Леонидович

Технологии проектирования баз данных

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com

Корректор *Синяева Г. И.*
Верстка *Паранская Н. В.*
Дизайн обложки *Мовчан А. Г.*

Формат 70×100¹/₁₆. Печать цифровая.
Усл. печ. л. 40,46. Тираж 200 экз.

Веб-сайт издательства: www.dmkpress.com