А. Н. Крутов

ПРОЕКТИРОВАНИЕ И РАЗРАБОТКА БАЗ ДАННЫХ

Самара
2013

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«САМАРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Кафедра безопасности информационных систем

А.Н. Крутов

ПРОЕКТИРОВАНИЕ И РАЗРАБОТКА БАЗ ДАННЫХ

Утверждено редакционно-издательским советом университета
в качестве электронного учебного пособия
для студентов механико-математического факультета
специальности 090103.65 – "Организация и технология
защиты информации"

Самара
Издательство «Самарский университет»
2013

УДК 681.3.07
ББК 32.973.26
К 64

Рецензенты: канд. техн. наук А. В. Костров,
канд. техн. наук С. Е. Агафонова

Крутов, А.Н.

К 64 Проектирование и разработка баз данных : [Электронный ресурс] : учебное пособие / А. Н. Крутов. Электрон. учебное пособие. – Самара: Издательство «Самарский университет, 2013. – 100 с. – Режим доступа: http://weblib.samsu.ru/localsrc/ssupress/main.php, ограниченный. – Загл. с экрана.

Основой данного учебного пособия являются материалы лекций, прочитанных автором по курсу «Системы управления базами данных» для специальности 090103.65 – «Организация и технология защиты информации».

В пособие кратко рассматриваются теоретические и практические аспекты, связанные с разработкой баз данных для задач различных предметных областей.

Предназначено для студентов механико-математического факультета Самарского государственного университета специальности «Организация и технология защиты информации».

УДК 681.3.07
ББК 32.973.26

СОДЕРЖАНИЕ

Введение 4

1. Файловые информационные системы 5

2. Понятие базы данных и СУБД 7
Компоненты среды СУБД 9
Преимущества и недостатки СУБД 11
Архитектура многопользовательских СУБД 12

3. Модель «сущность-связь» 15
Элементы модели «сущность-связь» 15
Диаграммы "сущность-связь" классической модели 17
Диаграммы "сущность-связь" в стиле UML 19
Диаграммы "сущность-связь" в стиле Erwin 21

4. Реляционная модель и нормализация 25
Типы аномалий 25
Классы отношений 28
Нормальные формы 29
Денормализация 43

5. Жизненный цикл приложения баз данных 48

6. Основы языка SQL 57
История языка SQL 57
Использование SQL для извлечения данных 59
Изменение содержимого базы данных 75
Определение данных 77

7. Защита баз данных 81
Некомпьютерные средства защиты 82
Компьютерные средства защиты 84
Системы защиты от несанкционированного доступа 89

8. Задачи для лабораторных работ 98

Заключение 99

Библиографический список 99

ВВЕДЕНИЕ

Хотим мы этого или не хотим, но базы данных стали неотъемлемой частью нашей повседневной жизни. За примерами использования баз данных не надо далеко ходить. Достаточно вспомнить, как мы расплачиваемся за купленные товары в супермаркете или работаем с электронным каталогом библиотеки. В настоящее время знание технологий в базах данных является практически необходимым условием успешного карьерного роста в сфере IT-услуг. Если студент после окончания высшего учебного заведения захочет профессионально заняться программированием, то с вероятностью 95% можно будет сказать, что его работа так или иначе будет связана с базами данных.

Многие студенты находят этот предмет интересным и увлекательным, хотя порой он может быть трудным[1]. Проектирование и разработка баз данных требуют одновременно творческих и инженерных навыков. Так, проектирование логической схемы базы данных, по сути, является искусством, т.к. требует однозначной интерпретации зачастую противоречивых требований пользователей. Задача по адаптации логической схемы базы данных под конкретную систему управления базами данных (СУБД), т.е. создание физической схемы, является чисто инженерной задачей, связанной с необходимостью обеспечить наиболее высокую производительность приложений, работающих с ней.

К сожалению, как показывает практика, студенты поначалу поверхностно относятся к изучению принципов проектирования баз данных. Для 90 % из них изучение теории по этому вопросу в стенах университета заканчивается после создания своего первого приложения с базами данных для себя или под заказ. С этого момента все внимание переключается на изучение аспектов конкретной СУБД. Через это в свое время пошел и автор данного пособия.

В настоящем пособии не описываются принципы разработки приложений с базами данных. Для этого есть соответствующие монографии, с помощью которых это сделать проще и быстрее. Основная цель данного пособия – научить студентов проектированию и анализу своих или чужих схем баз данных. Для этого в пособии приводятся теоретические сведения, дополненные практическими примерами автора. Причем приводятся примеры, как удачного проектирования, так и откровенно неправильные. Большое внимание в пособие уделяется вопросу построения системы защиты базы данных от несанкционированного доступа. Все вышеизложенное прекрасно согласуется со спецификой специальности 090103.65–«Организация и технология защиты информации».

При подготовке данного методического пособия активно использовалась литература, приведенная в библиографическом списке. Данный список намеренно не содержит слишком большое количество монографий, чтобы акцентировать внимание студентов на самых главных из них.

---

[1] К ним, к сожалению, не относился автор данного пособия, для которого, по началу, базы данных показались очень скучным делом.

1. ФАЙЛОВЫЕ ИНФОРМАЦИОННЫЕ СИСТЕМЫ

Стало почти традицией то, что любое пособие по базам данных начинается с обзора их предшественниц. Так в монографии Томаса Конолли [1], их называют файловыми системами[1]. Для того, чтобы не было лишней путаницы, в настоящем пособие используется термин **файловые информационные системы**.

Существует несколько причин, по которым с ними следует познакомиться.

- Понимание проблем, присущих файловым информационным системам, может предотвратить их повторение в СУБД.
- Знать принципы работы файловых информационных систем не только очень полезно, но и необходимо при выполнении перехода от файловой информационной системы к системе баз данных.

Файловые информационные системы – набор программ, которые выполняют для пользователей некоторые операции, например создание отчетов. Каждая программа определяет свои собственные данные и управляет ими.

Не следует считать, что файловые информационные системы – это пережиток прошлого и в настоящее время не используются. Если требуется написать простую программу по регистрации списка студентов с возможностью вывода отчетов по какой-либо группе, то проще всего не связываться с базами данных вообще, а использовать, например, тип file of record языка Pascal:

```
TStudent=record
  TestNumber:int64;     // Номер зачетной книжки
  FIO: string[150];     // ФИО студента
  GroupNumber:integer; // Номер группы
  Address:string[100]; // Адрес
  Phone:string[20];     // Телефон
end;

StudentFile=file of TStudent;
```

Таким образом, с помощью данного типа можно организовать запись и чтение разнотипной информации. Для ускорения доступа к нужным записям можно обеспечить хранение списка студентов в упорядоченном виде и использовать бинарный поиск.

[1] Типичный пример неудачного перевода, стоивший большому количеству студентов своевременного зачета по курсу СУБД, для которых посещение лекций было непозволительной роскошью. Естественно, что, услышав знакомое сочетание слов, они начинали рассказывать про FAT и NTFS, даже не подумав, какое отношение они имеют к базам данных.

Файловые информационные системы были первой попыткой компьютеризировать известные всем ручные картотеки. Они вполне годятся для работы с большим количеством объектов, которые нужно только хранить и извлекать. Однако они совершенно не подходят для тех случаев, когда нужно установить перекрестные связи или выполнить обработку сведений.

Ограничения, присущие файловым информационным системам

- Разделение и изоляция данных.
- Дублирование данных.
- Зависимость от данных.
- Несовместимость файлов.
- Фиксированные запросы/быстрое увеличение количества приложений.

2. ПОНЯТИЕ БАЗЫ ДАННЫХ И СУБД

База данных – совместно используемый набор логически связанных данных (и описание этих данных), предназначенный для удовлетворения информационных потребностей организации.

Важно понять, что в этом определении прячется «между строк».

Итак, база данных – это **единое хранилище данных**, которое однократно определяется, а затем используется одновременно многими пользователями из различных подразделений. Т.е. при соответствующей разработке базы данных имеет место **минимальная доля избыточности**. Причем база данных хранит не только **рабочие данные** этой организации, но и их **описания**. В совокупности, описание данных называется **системным каталогом**, или **словарем данных**, а сами элементы описания принято называть **метаданными**.

Кроме этого, как следует из определения, данные являются **"логически связанными"**. Это значит, что при анализе задачи из какой-либо предметной области необходимо выделить сущности, атрибуты и связи. Сначала разберемся с определениями.

Сущность – отдельный тип объекта организации, который нужно представить в базе данных.

Атрибут – свойство, которое описывает некоторую характеристику описываемого объекта.

Связь – это то, что объединяет несколько сущностей.

Невооруженным глазом видно, что данные определения выглядят какими-то туманными, особенно последнее. К счастью, понять смысл этих определений достаточно просто на примерах. Например, мы разрабатываем систему для учета успеваемости студентов ВУЗа. Тогда примерами сущностей для данной задачи будут «Студенты», «Преподаватели» и «Предметы». Примерами атрибутов будут фамилия студента, номер его зачетной книжки, наименование предмета[1]. Т.к. студенты посещают предметы, которые ведут различные преподаватели, то можно утверждать, что между сущностями «Студенты», «Преподаватели» и «Предметы» существуют связи. Природу этих связей рассмотрим далее в соответствующем разделе, посвященном описанию модели "сущность-связь".

Теперь введем определение СУБД.

Система управления базами данных (СУБД) - это программное обеспечение, с помощью которого пользователи могут определять, создавать и поддерживать базу данных, а также осуществлять к ней контролируемый доступ.

[1] Т.е. атрибутом будет являться «Наименование предмета», но никак не «Алгебра», «Физика» и т.п.

Ниже рассмотрим данное определение и возможности СУБД более подробно.

Определение базы данных обычно осуществляется с помощью языка определения данных (DDL - Data Definition Language). Примером команды языка DDL, в частности, является команда create table, с помощью которой создаются таблицы средствами языка SQL.

Вставка, редактирование, удаление и извлечение информации из базы данных обычно осуществляется с помощью языка управления данными (DML – Data Manipulation Language). Существует две разновидности языков DML – **процедурные** и **непроцедурные** языки. Они отличаются между собой способом извлечения данных. Основное отличие заключается в том, что процедурные языки обычно обрабатывают информацию в базе данных последовательно, запись за записью, а непроцедурные оперируют сразу целыми наборами записей.

В качестве примера команды процедурного DML можно привести команду SCAN из СУБД FoxPro. Она, по своей сути представляет собой цикл “for” для базы данных.

Наиболее распространенным типом непроцедурного языка является язык структурированных запросов SQL, который в настоящее время определяется специальным стандартом и фактически является обязательным языком для любых реляционных СУБД[1].

Кроме этого, СУБД предоставляет контролируемый доступ к базе данных с помощью следующих средств:

- системы обеспечения безопасности, предотвращающей несанкционированный доступ к базе данных со стороны пользователей;

В настоящее время в мощных СУБД появились защиты от несанкционированного доступа (НСД) отдельных пользователей или групп. Однако в большинстве случаев защита осуществляется на уровне объектов базы данных целиком, а не отдельных частей. Только в некоторых СУБД, таких, как Oracle имеются средства так называемой горизонтальной защиты[2], но чтобы ими по-настоящему можно было пользоваться, требуется написание отдельного приложения для настройки.

- системы поддержки целостности данных, обеспечивающей непротиворечивое состояние хранимых данных;

[1] Несмотря на то, что существует стандарт для языка SQL, его реализация в различных СУБД порой расходится с требованиями стандарта. Поэтому для более тщательного изучения возможностей конкретной СУБД приходится разбираться с «ее» SQL-м подробнее.

[2] Имеется в виду fine grained access control.

К данным средствам относятся ограничения для доменов и атрибутов, устанавливающие допустимые значения полей, средства поддержки целостности сущностей, а также ссылочной целостности.

- системы управления параллельной работой приложений, контролирующей процессы их совместного доступа к базе данных;

Самые первые средства управления параллельной работой нескольких пользователей – это механизмы блокировок записей. Позднее к ним добавились механизмы транзакций, которые, свою очередь претерпели ряд этапов своего развития[1].

- системы восстановления, позволяющей воссоздать базу данных до предыдущего непротиворечивого состояния, нарушенного в результате сбоя аппаратного или программного обеспечения.

Не во всех СУБД средства восстановления есть[2], поэтому в ряде случаев приходится разрабатывать специальные приложения для устранения этих недостатков.

Компоненты среды СУБД

В монографии [1] приводится довольно удачное визуальное изображение среды СУБД, в котором данные, хранящиеся в системе, являются своеобразным мостиком между компьютером и человеком (рис. **1**)

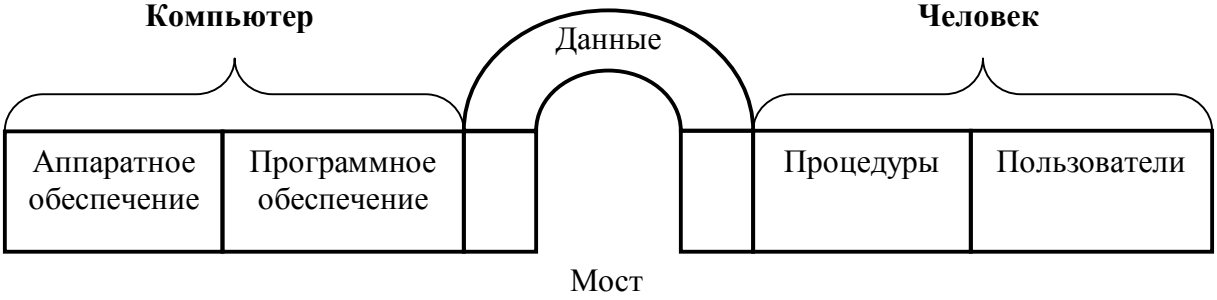

Рис. 1. Компоненты среды СУБД

Как показано на рис. **1**, в среде СУБД можно выделить пять основных компонентов: аппаратное обеспечение, программное обеспечение, данные, процедуры и пользователи.

[1] К ним, в частности, относятся появление точек сохранения (savepoints), а также автономных транзакций.
[2] Например, их нет в СУБД FoxPro 2.6.

Аппаратное обеспечение

Для работы СУБД и приложений необходимо некоторое аппаратное обеспечение. Оно может варьироваться в очень широких пределах. Используемое аппаратное обеспечение зависит от предполагаемой нагрузки на систему и типа используемой СУБД. Так, если для нормальной работы СУБД FoxPro будет достаточно старого 486-го компьютера, то только для установки СУБД Oracle 9i потребуется компьютер с процессором Pentium 300 МГц, с жестким диском не менее 4,75 ГБ[1].

Программное обеспечение

Данный компонент охватывает программное обеспечение самой СУБД и прикладных программ, вместе с операционной системой. Обычно приложения создаются на языках третьего поколения, таких как Object Pascal или C++ со вставками SQL-операторов[2]. Впрочем, СУБД может иметь свои собственные инструменты четвертого поколения, предназначенные для быстрой разработки приложений с использованием встроенных генераторов отчетов, форм, графических изображений и даже полномасштабных приложений[3].

Данные

Самым важным компонентом среды СУБД, безусловно, являются данные. База данных содержит как рабочие данные, так и метаданные. Структура базы данных называется **схемой**.

Процедуры

К процедурам относятся руководства пользователям и обслуживающему персоналу, содержащие подробное описание процесса использования и сопровождения информационной системы, включающие описание различных действий, например:

- запуск и останов СУБД;
- регистрация в СУБД;
- использование отдельного инструмента СУБД или приложения;
- создание резервных копий СУБД;
- обработка сбоев аппаратного и программного обеспечения.

Пользователи

Внутри данной категории можно выделить четыре различные группы: администраторы данных и баз данных, разработчики баз данных, прикладные программисты и конечные пользователи.

---

[1] Это требования по установке. Для обеспечения приемлемого быстродействия данной СУБД потребуется компьютер намного мощнее.

[2] Сам язык SQL относят к языку программирования четвертого поколения.

[3] В качестве примера можно привести среду разработки Oracle Developer.

Администратор данных отвечает за управление данными, включая планирование базы данных, разработку и сопровождение стандартов, бизнес правил и деловых процедур, а также за логическое проектирование базы данных.

Администратор базы данных отвечает за физическую реализацию базы данных. Его обязанности носят технический характер и для их выполнения необходимы знания по конкретной СУБД и ее системного окружения.

Разработчики баз данных. В проектировании больших баз данных участвуют два разных типа разработчиков: разработчики **логической базы данных** и разработчики **физической базы данных**. В небольших командах разработчиков созданием логических и физических структур баз данных занимаются одни и те же люди[1].

Прикладные программисты. После создания структуры базы данных приступают к разработке приложений, предоставляющих пользователям необходимые им функциональные возможности. Эту работу выполняют прикладные программисты. Как правило, каждая программа содержит некоторые операторы, требующие от СУБД выполнения определенных действий с базой данных – например, таких как поиск, вставка, редактирование или удаление данных.

Пользователи являются клиентами базы данных. Она проектируется, создается, и поддерживается для того, чтобы обслуживать их информационные потребности.

Преимущества и недостатки СУБД

СУБД обладают как многообещающими потенциальными преимуществами, так и недостатками[2], которые кратко рассматриваются в данном разделе.

Преимущества

- Контроль над избыточностью данных.
- Непротиворечивость данных.
- Совместное использование данных.
- Поддержка целостности данных.
- Повышенная безопасность.
- Применение стандартов.
- Повышение эффективности с ростом масштабов системы.
- Развитые службы резервного копирования и восстановления.

[1] Автору стыдно признаться, но за свою практику ему ни разу не приходилось видеть отдельно разработчика логической структуры и разработчика физической структуры базы данных. Но некоторые особо продвинутые студенты убеждали его, что такие люди все же есть.

[2] Данные недостатки можно считать особенностями СУБД.

Недостатки

- Сложность.
- Размер.
- Стоимость СУБД.
- Дополнительные затраты на аппаратное обеспечение.
- Более серьезные последствия при выходе системы из строя.

Архитектура многопользовательских СУБД

В данном разделе описываются различные типовые архитектурные решения, используемые при реализации многопользовательских СУБД. К ним относятся схемы обычной телеобработки, файловый сервер и технология "клиент/сервер".

Телеобработка

Традиционной архитектурой многопользовательских систем считалась схема «телеобработки», при которой один компьютер был соединен с несколькими терминалами так, как показано на рисунке **2**.

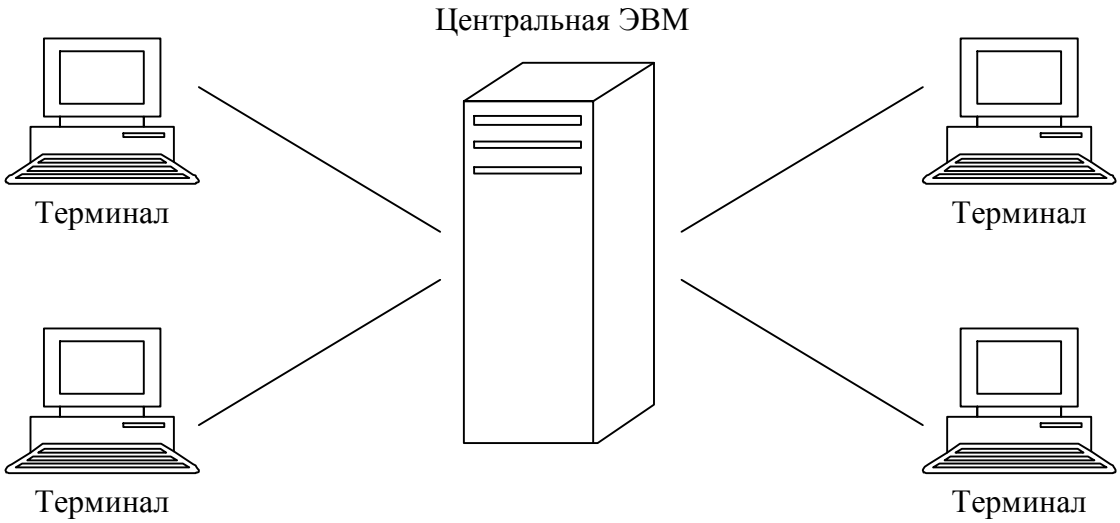

Рис. 2. Топология телеобработки

При этом вся обработка выполнялась в рамках единственного компьютера, а присоединенные к нему пользовательские терминалы были типичными "неинтеллектуальными" устройствами, не способными функционировать самостоятельно. С центральным процессором терминалы были связаны с помощью системы кабелей, по которым они посылали сообщения пользовательским приложениям. В свою очередь, пользовательские приложения обращались к необходимым службам СУБД. Таким же образом сообщения возвращались назад на пользовательский терминал. При

такой архитектуре основная и чрезвычайно большая нагрузка возлагалась на центральный компьютер, который должен был выполнять не только действия прикладных программ и СУБД, но и значительную работу по обслуживанию машин клиентов (например, форматирование данных, выводимых на экраны терминалов)[1].

Файловый сервер

В среде файлового сервера обработка данных распределена в сети, обычно представляющей собой локальную вычислительную сеть. Файловый сервер содержит файлы, необходимые для работы приложений и самой СУБД. Однако пользовательские приложения и сама СУБД размещены и функционируют на отдельных рабочих станциях, и обращаются к файловому серверу только по мере необходимости получения доступа к нужным им файлами – как показано на рис. **3.** Таким образом, файловый сервер функционирует просто как совместно используемый жесткий диск. СУБД на каждой рабочей станции посылает запросы файловому серверу по всем необходимым ей данным, которые хранятся на диске файл-сервера[2]. Такой подход характеризуется значительным сетевым трафиком, что может привести к снижению производительности всей системы в целом.

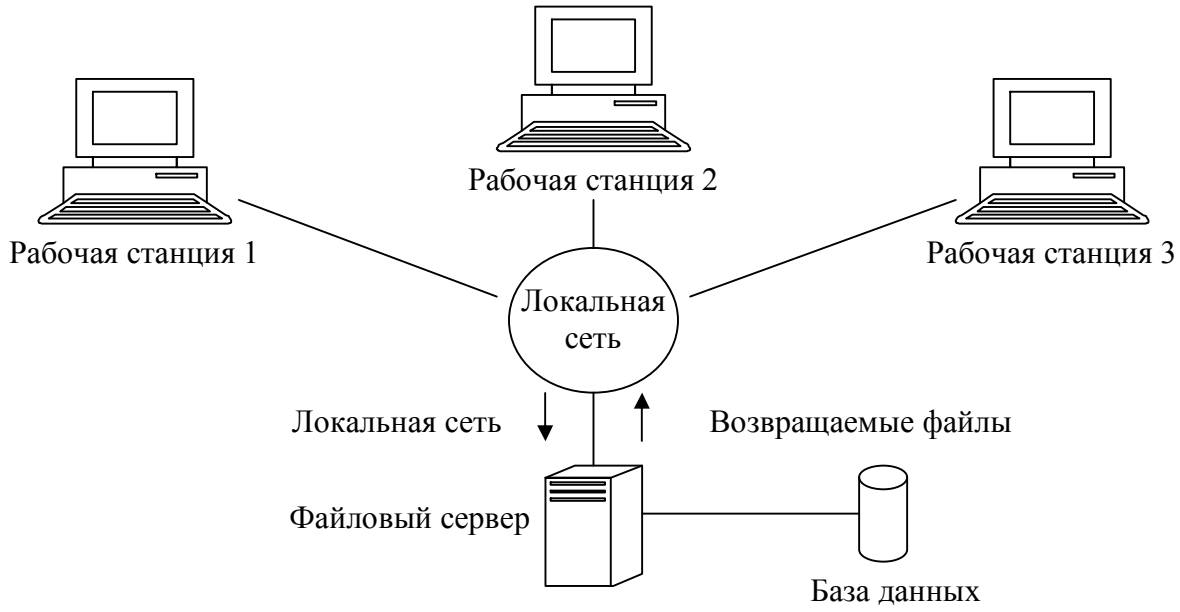

Рис. 3. Архитектура с использованием файлового сервера

[1] Ошибкой было бы считать то, что данная топология в настоящее время нигде не используется. В разнородных сетях широко используются т.н. «тонкие» клиенты, например, Citrix ICA Client. С их помощью, например, можно организовать сетевую работу пользователей Macintosh с приложениями под Windows. При этом работа пользователей осуществляется через сервер приложений в режиме терминальных сессий.

[2] СУБД, «идеально» подходящие под данную топологию – FoxPro, DBF, Paradox.

Технология "клиент/сервер"

Технология «клиент/сервер» была разработана с целью устранения недостатков, имеющихся двух подходах, описанных ранее. «Клиент/сервер» означает такой способ взаимодействия программных компонентов, при котором они образуют единую систему. Как видно из самого названия, существует некий клиентский процесс, требующий определенных ресурсов, а также серверный процесс, который эти ресурсы предоставляет. На рис. **4** показана архитектура типа "клиент/сервер".

В контексте базы данных клиент управляет пользовательским интерфейсом и логикой приложения. Он принимает от пользователя запрос, проверяет синтаксис и генерирует запрос к базе данных на понятном ей языке. Затем он передает сообщение серверу, ожидает поступления ответа и форматирует полученные данные для представления их пользователю. Сервер принимает и обрабатывает запросы к базе данных, а затем передает полученные результаты обратно клиенту. Такая обработка включает проверку полномочий клиента, обеспечение требований целостности, поддержку системного каталога, а также выполнение запроса и обновление данных. Помимо этого, поддерживается управление параллельностью и восстановлением[1]. Данный тип архитектуры обладает следующими преимуществами:

- Повышение общей производительности системы.
- Сокращение коммуникационных расходов.
- Повышение уровня непротиворечивости данных.

Рис. 4. Общая схема построения систем с архитектурой "клиент/сервер"

[1] К СУБД технологии клиент-сервер относятся Interbase, DB2, Oracle, MS SQL и многие другие.

3. МОДЕЛЬ «СУЩНОСТЬ-СВЯЗЬ»

В данной главе описывается и иллюстрируется использование модели «сущность-связь», введенной Питером Ченом в 1976 г. Модель «сущность-связь» вошла в состав множества CASE-инструментов, которые также внесли свой вклад в ее эволюцию. На сегодняшний день не существует единого общепринятого стандарта для модели «сущность-связь», зато есть набор общих конструкций, которые лежат в основе большинства вариантов этой модели. Рассматриваются диаграммы классической модели «сущность-связь», предложенной П. Ченом, диаграммы «сущность-связь» в нотации UML, а также построение диаграммы «сущность-связь» средствами программы Erwin.

Элементы модели «сущность-связь»

Ключевыми элементами модели «сущность-связь» являются **сущности, атрибуты, идентификаторы** и **связи**. О некоторых понятиях в пособие уже упоминалось.

Сущность – это некоторый объект, идентифицируемый в рабочей среде пользователя, нечто такое, за чем пользователь хотел бы наблюдать. Сущности одного и того же типа группируются в **классы сущностей**[1].

Атрибуты, как уже упоминалось – это свойства, которые описывают характеристики сущности. В модели «сущность-связь» предполагается, что все экземпляры одного класса сущностей имеют одинаковые атрибуты.

В модели «сущность-связь» атрибуты бывают обычными, композитными и многозначными. В качестве примера композитного атрибута можно привести атрибут «Адрес», состоящий из группы атрибутов {Индекс, Населенный пункт, Улица}. Примером многозначного атрибута может служить атрибут «Телефон клиента» сущности «Клиент», который может содержать сразу несколько номеров телефонов данного клиента. В большинстве реализаций модели «сущность-связь» требуется, чтобы многозначные атрибуты преобразовывались в сущности.

Идентификаторы – это атрибуты экземпляров сущностей, с помощью которых эти экземпляры именуются или идентифицируются.

Например, экземпляры сущности класса «Студент» могут идентифицироваться по атрибуту «Номер зачетной книжки». Идентификатор экзем-

[1] Аналогия здесь абсолютно такая же, как различие класса и объекта (осталось вспомнить курс методов программирования). Класс – это множество объектов определенного типа, а объект – это экземпляр определенного класса.

пляра сущности состоит из одного или более атрибутов сущности. Он может быть **уникальным** либо **неуникальным**. Если идентификатор является уникальным, его значение будет указывать на один и только один экземпляр сущности. Если идентификатор является неуникальным, его значение будет указывать на некоторое множество экземпляров. Так, идентификатор «Номер зачетной книжки» является уникальным идентификатором, а «Имя студента», очевидно, нет. Идентификаторы, состоящие из нескольких атрибутов, называются **композитными идентификаторами**.

Взаимоотношения сущностей выражаются **связями**. Модель "сущность-связь" включает в себя **классы связей** и **экземпляры связей**.

Классы связей – это взаимоотношения между классами сущностей

Экземпляры связи – взаимоотношения между экземплярами сущностей.

Класс связей может затрагивать несколько классов сущностей. Число классов сущностей, участвующих в связи, называется **степенью связи**. На рисунке **5**(а) представлена связь «Студент-телефон», имеющая степень 2, в то время, как на рисунке **5**(б) степень связи «Родитель» равна трем.

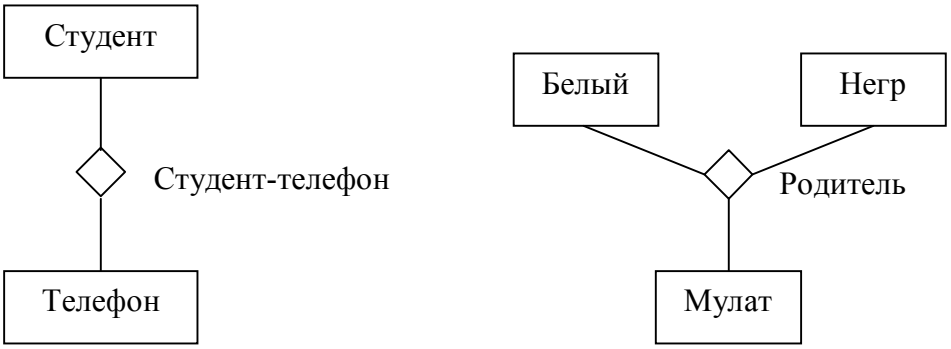

Рис. 5. Различные степени связей

Связи степени 2 весьма распространены, часто их еще называют **бинарными** связями[1].

Три типа бинарных связей

Традиционно выделяют три типа бинарных связей: «один к одному», «один ко многим» и «многие ко многим» в зависимости от того, как экзем-

[1] На самом деле, любая n-ая связь – не более чем эффектная картинка. На практике любые связи подобного типа реализуются через бинарные связи.

пляры одного класса сущностей связаны с экземплярами другого класса сущностей.[1] На рисунке **6** показаны три типа бинарных связей.

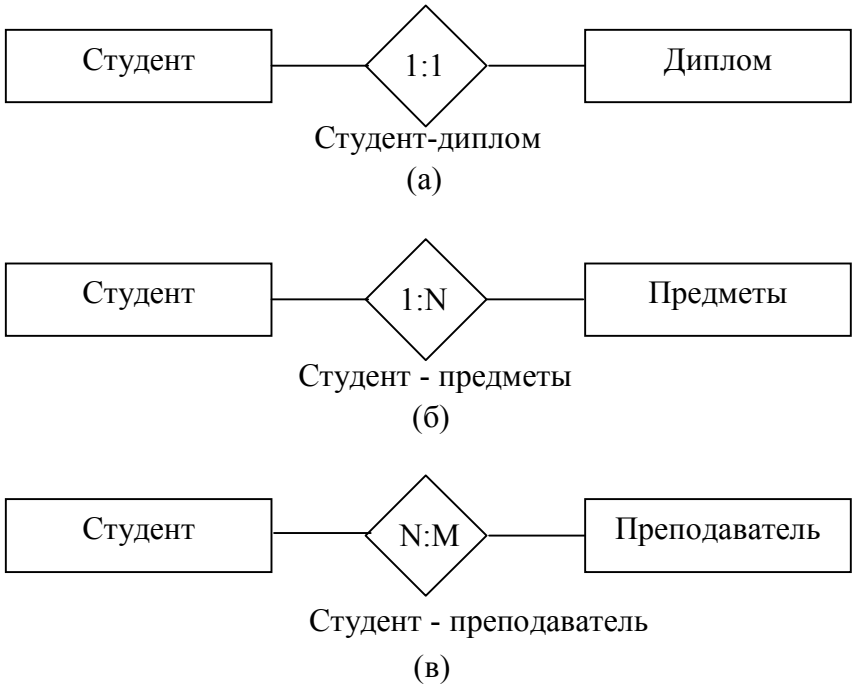

Рис. 6. Три типа бинарных связей: а – «один к одному»;
б – «один ко многим»; в – «многие ко многим»

Числа внутри ромба, символизирующего связь, обозначают максимальное количество сущностей на каждой стороне связи. Эти ограничения называются **максимальными кардинальными числами**, а совокупность из двух таких ограничений для обеих сторон связи называется **максимальной кардинальностью** связи.

Изображения типов связей, представленных на рисунке **6** не дают ответа на вопрос, каким образом физически осуществляется связь между различными классами сущностей.

Диаграммы "сущность-связь" классической модели

Изображение атрибутов

Ниже на рисунке **7** показано, каким образом в классической модели «сущность-связь» отображаются атрибуты.

Из приведенного рисунка видно, что при большом количестве атрибутов данные диаграммы могут быть достаточно громоздкими

[1] На самом деле можно выделить еще один тип связи: «многие к одному». Его легко получить из связи «один ко многим», если смотреть на нее с противоположной стороны. Однако на практике данный тип связи используется очень редко.

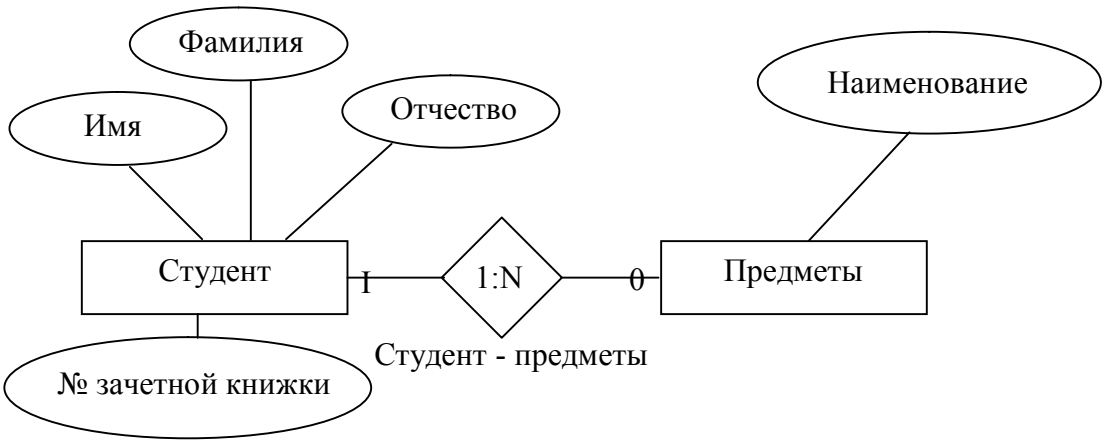

Рис. 7. Изображение атрибутов на диаграммах "сущность-связь"

Слабые сущности

В модели «сущность-связь» определен особый тип сущности, называемый **слабой сущностью**. К слабым сущностям относятся такие сущности, которые могут существовать в базе данных только в том случае, если в ней присутствует сущность некоторого другого типа. Сущность, не являющаяся слабой, называется **сильной сущностью**. На рис. **8** приведен пример, изображающий данный тип сущности. Можно выделить два типа слабой сущности: **обычная** и **идентификационно – зависимая**. В первом случае зависимость слабой сущности просто подразумевается и не более того. Во втором же случае данная зависимость жестко фиксируется тем, что идентификатор слабой сущности является составным атрибутом, включающим в себя также идентификатор сильной сущности.

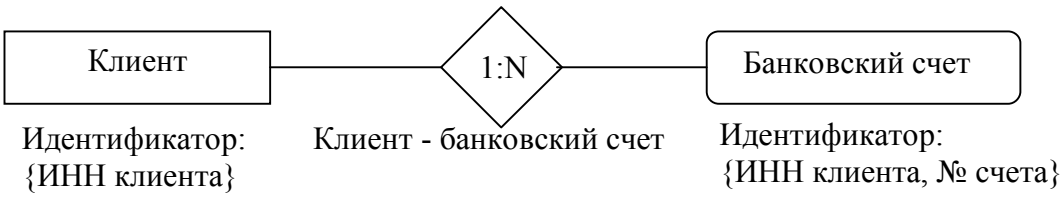

Рис. 8. Пример идентификационно-зависимой слабой сущности

Подтипы сущностей

Некоторые сущности имеют необязательные наборы атрибутов, эти сущности часто представляются с помощью **подтипов**. Рассмотрим, например, сущность «Человек» с атрибутами «Фамилия», «Имя», «Отчество». Предположим, что он может быть либо студентом, либо преподавателем, либо и тем, и другим, и что необходимо хранить некоторую дополни-

тельную информацию в зависимости от «типа человека». Пусть эта информация имеет следующее содержание:

Человек: Фамилия, Имя, Отчество, Адрес, Телефон

Студент: Номер зачетной книжки, Курс, Специальность

Преподаватель: Должность, Кафедра, Стаж работы

Самым «правильным» решением с точки зрения проектирования баз данных будет определение трех классов сущностей, как показано на рис. 9. Здесь сущности «Студент» и «Преподаватель» изображены как подтипы сущности «Человек». Из данного рисунка, к сожалению, не ясно, каким образом физически реализуются подтипы. При рассмотрении других способов изображения модели «сущность-связь»[1] этот недостаток устраняется.

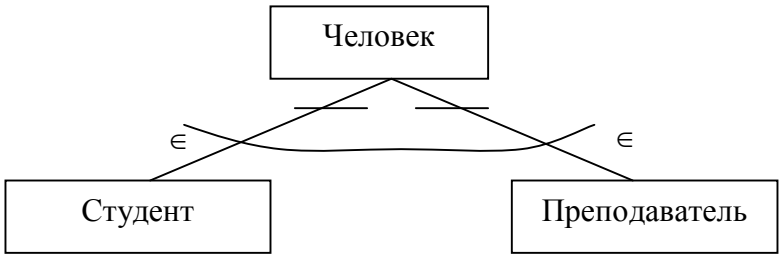

Рис. 9. Подтипы сущностей

Таким образом, диаграммы «сущность-связь» классической модели обладают довольно большой громоздкостью. Кроме этого, с их помощью порой невозможно понять физических аспектов осуществления той или иной связи между сущностями, либо их подтипами.

Диаграммы "сущность-связь" в стиле UML

Унифицированный язык моделирования (UML) – это набор структур и методик для моделирования и проектирования приложений с помощью объектно-ориентированного подхода[2]. UML – это одновременно и методология разработки систем ООП, и набор инструментов для разработки таких систем. Сфера проникновения данного языка в настоящее время настолько велика, что осуществляются попытки его использования, в том числе, и при проектировании баз данных, т.к. в этом процессе очень много общего с процессом построения объектной модели. В данном разделе рассматриваются способы представления основных элементов модели «сущность-связь» в нотации UML.

[1] Имеется в виде нотация UML и нотация Erwin.

[2] Для более подробной информации по особенностям языка UML студентам рекомендуется полистать пособие автора [3], если они этого еще не делали на курсе «Методы программирования».

Сущности и связи в UML

На рис. **10** приведено UML-представление структур. Каждая сущность представлена в виде класса[1].

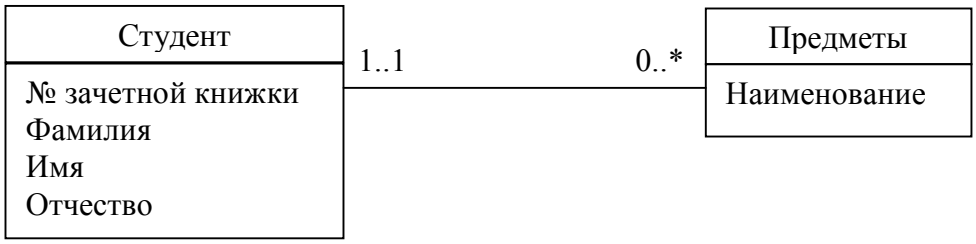

Рис. 10. Представление связей в UML

Если сравнивать данный рисунок с его аналогом из классической модели (см. рис. **7**), то можно сделать заключение, что в случае UML-представления пропадает громоздкость представления сущностей с большим количеством атрибутом.

Представление слабых сущностей

На рис. **11** изображено UML-представление слабых сущностей на примере уже рассмотренных клиентов и их банковских счетов (см. рис. **8**).

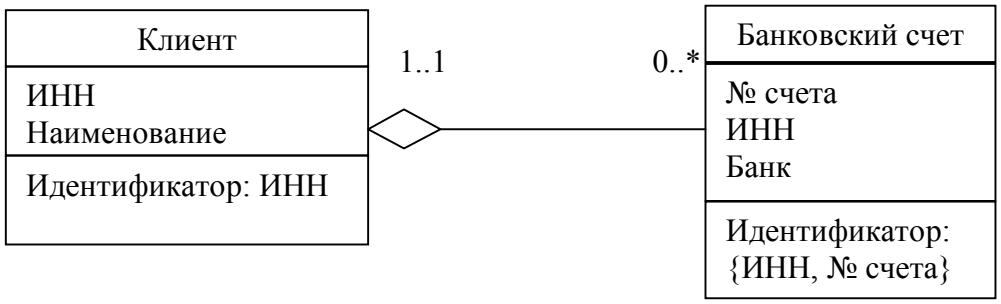

Рис. 11. Пример идентификационно-зависимой слабой сущности в нотации UML

Представление подтипов

Способ представления подтипов в UML показан на рис. **12**.

Как видно из данного рисунка, подтипы сущностей в данном случае реализуются с помощью механизма иерархии. Сущность «Человек» выступает в роли класса-предка, а сущности «Студент» и «Преподаватель» – его наследников.

[1] Имеется в виду класс языка UML.

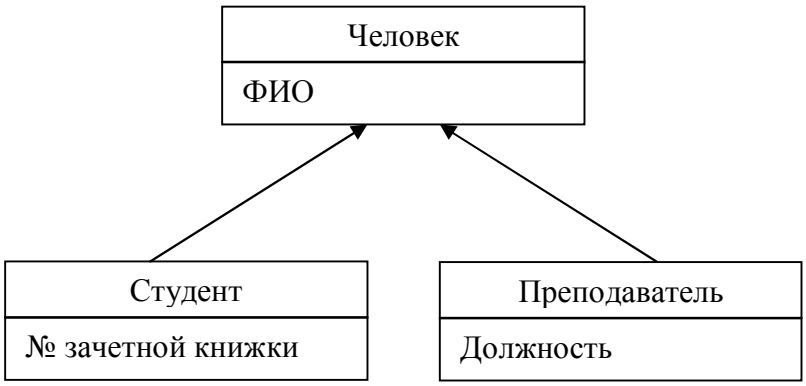

Рис. 12. Представление подтипов в UML

Таким образом, диаграммы «сущность-связь» в нотации языка UML лишены той громоздкости, которая была присуща диаграммам «сущность-связь» классической модели. Однако физические аспекты реализации той или иной связи, либо подтипа все же трудно обнаруживаемые. Следующий способ изображения модели «сущность-связь», пожалуй, наиболее удобен для разработки моделей «сущность-связь».

Диаграммы "сущность-связь" в стиле Erwin

Программа AllFusion Erwin Data Modeller, является, фактически, стандартным инструментом при разработке баз данных. С помощью него осуществляется полный комплекс работ, связанных с прямым и обратным проектированием базы данных, построением логической и физической структуры базы данных.

Сущности и связи в Erwin

На рис. **13** приведено представление структур с помощью программы Erwin.

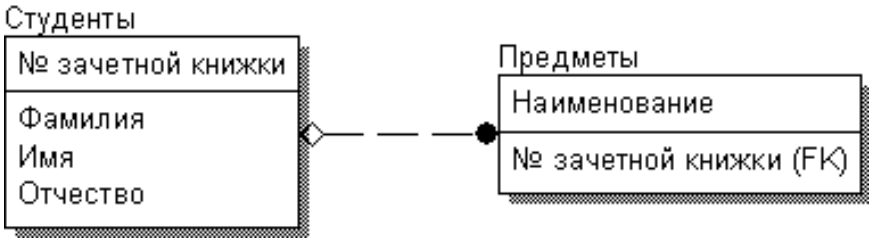

Рис. 13. Представление связей в Erwin

Все сущности в Erwin отображаются в виде прямоугольников. Наименования классов сущностей изображаются непосредственно над ними, а

уникальные идентификаторы в прямоугольнике сверху. Все остальные атрибуты перечисляются в нижнем прямоугольнике. Связи изображаются в виде линий. Жирная точка, исходящая с одного из концов связи, указывает то, какой класс сущности ссылается на другой класс сущности. Связь реализуется с помощью добавления в сущность дополнительного атрибута, который ссылается на уникальный идентификатор другой сущности. На примере рисунка **13** – это атрибут «№ зачетной книжки», который добавляется в сущность «Предметы»[1]. Таким образом, связи с помощью программы Erwin не только удобно отображаются, но и не возникает никаких вопросов по поводу физической их реализации.

Особенно хочется отметить способ реализации связи «многие ко многим». Путем добавления в сущности дополнительных атрибутов невозможно реализовать данный тип связи. Поэтому приходится между двумя таблицами вводить третью[2], с помощью которой осуществляется связь. Ниже на рисунке **14** приводится отображение связи «многие ко многим» на примере связи «студент-преподаватель».

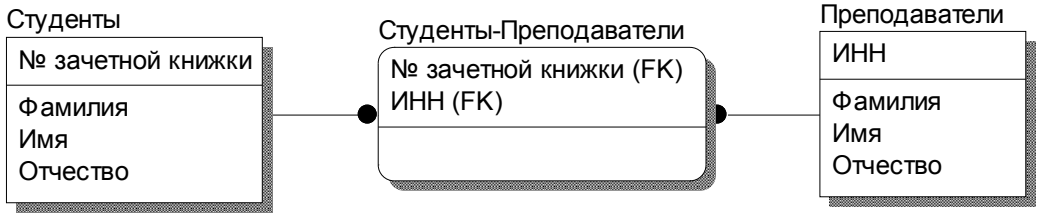

Рис. 14. Представление связи «многие ко многим»

Связь «многие ко многим», фактически, является объединением двух связей «один ко многим», идущих с разных сторон. Так, сущности «Студенты» и «Студенты-преподаватели» объединены между собой связью «один ко многим». То же самое относится и к сущностям «Преподаватели» и «Студенты-преподаватели». Непосредственной же связи между сущностями «Студенты» и «Преподаватели» нет, а связь реализуется через таблицу-связку. Как видно из приведенных рисунков, для того, чтобы на какую-либо сущность могла ссылаться другая сущность, необходимо, чтобы у нее был уникальный идентификатор.

Представление слабых сущностей

На рис. **15** изображено Erwin-представление идентификационно-зависимых слабых сущностей на примере рассмотренных ранее клиентов и их банковских счетов (см. рис. **8**, **11**).

[1] Слова в скобках (FK) означают «foreign key», т.е. внешний ключ.
[2] Так называемая таблица-связка.

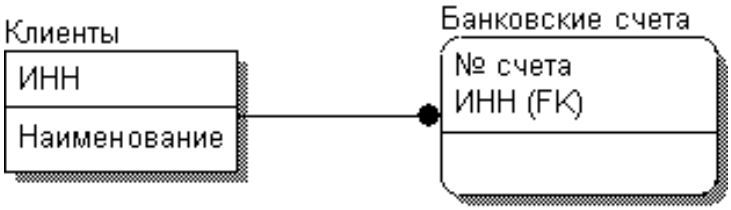

Рис. 15. Идентификационно-зависимая слабая сущность в Erwin

Представление подтипов

Способ представления подтипов в Erwin показан на рис. **16**.

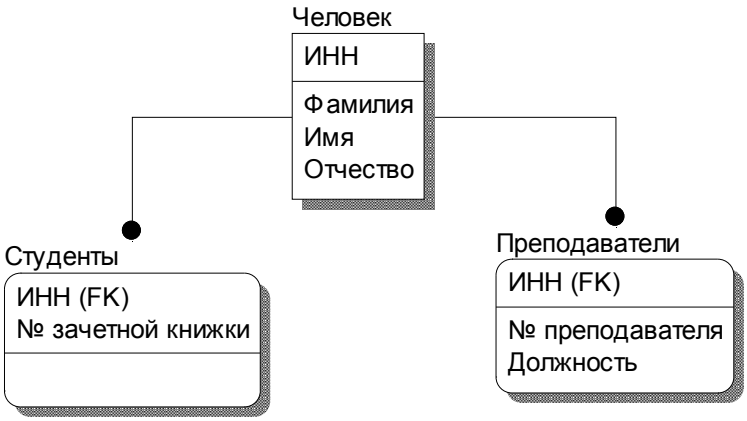

Рис. 16. Отображение подтипов в Erwin

С точки зрения автора[1], диаграммы Erwin представляют собой наиболее удобный способ отображения

Выбор идентификатора

Ранее было показано, что для того, чтобы на экземпляр какой-либо сущности мог сослаться какой-либо экземпляр другой сущности, необходимо, чтобы у первой сущности существовал уникальный идентификатор, или другими словами, первичный ключ, с помощью которого осуществляется однозначная идентификация записи в сущности.

При практической реализации какой-либо схемы базы данных имеются два пути решения:

- в каждой из сущностей среди имеющихся атрибутов попытаться найти отдельные атрибуты или их комбинации, которые можно использовать в качестве первичного ключа[2];

[1] Хотя это мнение автор никоем образом не стремится навязывать.
[2] В ряде монографий данный ключ называют «естественным ключом».

- в каждую из сущностей добавить дополнительный атрибут, который в дальнейшем использовать в качестве первичного ключа[1].

Таким образом, в первом случае в качестве первичного ключа используются уже существующие атрибуты, а во втором – новые. Казалось бы, первый способ более предпочтительный. Во-первых, экономится дисковое пространство ввиду того, что в таблицу не добавляются новые атрибуты. Во-вторых, автоматически гарантируется то, что совокупность значений атрибутов, выбранных в качестве первичного ключа, будет уникальной, что является дополнительной проверкой на корректность ввода информации пользователями. Основным недостатком данного способа является потенциальная возможность изменения у какой-либо записи первичного ключа, т.к. соответствующие атрибуты могут править человеком[2]. Данная ситуация является крайне нежелательной для реляционной базы данных, т.к. влечет необходимость смены всех соответствующих значений в сущностях, которые ссылаются на изменяемую запись. Во втором же случае этот недостаток можно полностью устранить. Для этого необходимо автоматически проставлять уникальные значения в дополнительные атрибуты и никогда в приложениях не показывать эти значения пользователям[3]. На практике используют оба способа выбора идентификаторов.

[1] Иногда его называют «суррогатным ключом».

[2] Вероятность такой смены может быть достаточно малая, однако законы Мерфи еще никто не отменял.

[3] Сторонников как первого способа выбора первичных ключей, так и второго, предостаточно. Автор относит себя ко второй группе, однако никому своего мнения не навязывает.

4. РЕЛЯЦИОННАЯ МОДЕЛЬ И НОРМАЛИЗАЦИЯ

В данном разделе объясняются фундаментальные принципы нормализации. Сначала дается определение нормализации.

Нормализация – это процесс преобразования отношения, имеющего некоторые недостатки, в отношение, которое этих недостатков не имеет. Нормализацию можно использовать как критерий для определения желательности и правильности отношений.

Данное определение является достаточно абстрактным и нуждается в дальнейшем пояснении. Во-первых, что считать недостатком отношения. Во-вторых, в чем заключается процесс нормализации. С этими вещами необходимо разобраться в первую очередь.

Не все отношения одинаковы: некоторые из них более предпочтительны, чем другие. Таблица, отвечающая минимальному определению отношения, может иметь неэффективную или неподходящую структуру. Для некоторых отношений какое-либо изменение данных может привести к нежелательным последствиям, называемыми **аномалиями**. В определении нормализации недостатками являются именно аномалии. Теперь необходимо разобраться с возможными типами аномалий.

Типы аномалий

В качестве примера рассмотрим отношение «Секция» на рисунке 17. Будем считать, что студенты могут ходить только в одну из секций. Кроме этого, исключим возможность появления однофамильцев[1], а также потребуем, чтобы плата в той или иной секции была постоянной вне зависимости от студента, который в эту секцию ходит. Исходя из всего этого, в качестве первичного ключа в сущности «Секция» можно взять атрибут «Студент». На примере данного отношения легко продемонстрировать все типы аномалий, которые чаще всего имеют место на практике.

Студент	Секция	Плата
Иванов	Плавание	100
Петров	Плавание	100
Сидоров	Бокс	50
Скворцов	Бег	30

Рис. 17. Отношение «Секция»

[1] Можно, конечно, вместо фамилии студента писать какой-либо числовой код, однако в этом случае пример потеряет наглядность.

Предположим, что мы хотим записать в таблицу тот факт, что плата в секцию «Футбол» составляет 80 единиц. однако мы не сможем ввести эти данные в отношение, пока хотя бы один из студентов не запишется в данную секцию. Это ограничение выглядит абсурдным. Никакой студент не сможет записаться в секцию, пока та не появится в базе данных. Однако, никакая секция не появится в базе данных, пока в нее не запишется какой-либо студент. Это ограничение называется **аномалией вставки**. Суть его заключается в невозможности записать в таблицу некоторый факт об одной сущности, не указав дополнительно некоторый факт о другой сущности.

Теперь представим себе ситуацию, что плата в секцию «Плавание» изменилась и вместо 100 единиц стала составлять 120[1]. В этом случае придется менять соответствующие значения сразу для двух студентов – Иванова и Петрова. Данная ситуация является источником потенциальной ошибки, т.к. в случае какого-либо сбоя или непреднамеренных действий пользователя окажется, что Иванов платит за занятия в секции «Плавание» уже 120 единиц, а Петров по-прежнему 100. Данная ситуация является одним из примеров так называемого нарушения целостности данных, т.е. информация в ней находится в противоречивом состоянии. А т.к. основную ценность базы данных составляют сами данные, получается, что эта ценность потеряна, информации из базы данных нельзя доверять. Данный тип аномалии называется **аномалией редактирования**.

Рассмотрим теперь процесс удаления информации из отношения, представленного на рисунке **17**. Допустим, что студента Сидорова отчислили, и поэтому он перестал также заниматься в спортивной секции. Естественным действием при этом является удаление соответствующей информации из отношения «Секция». Однако, если мы удалим строку о том, что студент Сидоров посещает секцию «Бокс», то мы автоматически потеряем также информацию о том, что, во-первых, существует секция «Бокс», а во-вторых, информацию о том, что плата в этой секции составляет 50 единиц. Данная ситуация называется **аномалией удаления** – то есть, удаляя факты, относящиеся к одной сущности, мы непроизвольно удаляем факты, относящиеся к другой сущности. Выполнив одну операцию удаления, мы теряем информацию сразу о двух сущностях.

Для того, чтобы устранить из приведенного выше отношения все приведенные выше типы аномалий, необходимо разделить отношение «Секция» на два отношения, каждое из которых будет содержать информацию на определенную тему. В первое отношение мы поместим атрибуты «Студент» и «Секция» (назовем это отношение «Студент-Секция»), а в другое – атрибуты «Секция» и «Плата» (назовем это отношение «Секция-Плата»). Соответствующие отношения приведены на рисунке **18**.

[1] Типичная ситуация, не правда ли?

Студент	Секция
Иванов	Плавание
Петров	Плавание
Сидоров	Бокс
Скворцов	Бег

Секция	Плата
Плавание	100
Бокс	50
Бег	30

Рис. 18. Отношения «Студент-Секция» и «Секция-Плата»

Для первой сущности в качестве первичного ключа по прежнему используется атрибут «Студент». Уникальным идентификатором второй сущности является наименование секции.

При новой организации данных не составляет проблемы добавить информацию о том, что появилась новая секция «Футбол» со стоимостью занятий 80 единиц. Для этого не придется принудительно «записывать» в секцию какого-либо студента, как это было в первом случае. Достаточно будет просто добавить запись в сущность «Секция-Плата». Когда студенты начнут записываться в эту секцию, в таблице «Студент-Секция» начнут появляться новые записи. Таким образом, аномалия вставки в данном случае отсутствует.

Никаких проблем не возникнет также и при необходимости коррекции платы за занятия в той или иной секции. В данном случае изменению всякий раз будет подвергаться только одна из записей сущности «Секция-Плата», а не несколько сразу, как это было раньше. Т.е. аномалии редактирования в данном случае также нет.

Теперь, если мы удалим запись о студенте Сидорове из таблицы «Студент-Секция», мы уже не потеряем никакой информации по поводу секции «Бокс», т.к. соответствующая запись в сущности «Секция-Плата» останется неизменной. Т.е. аномалия удаления в данном случае также отсутствует.

Разбиение одного отношения на два, однако, имеет один недостаток. Предположим, что студент хочет записаться в несуществующую секцию. Например, студент Федоров хочет записаться в секцию дельтапланеризма. Мы можем вставить соответствующую строку в отношение «Студент-Секция» (строка будет содержать запись (Федоров, Дельтапланеризм)), но следует ли это делать? Стоит ли разрешать студенту записываться в секцию, которая отсутствует в отношении «Секция-Плата»? Другими словами, должна ли система каким-либо образом препятствовать добавлению строк в таблицу «Студент-Секция», если название соответствующей секции отсутствует в таблице «Секция-Плата»? Ответ на этот вопрос должен содержаться в требованиях пользователей. Если такое действие должно быть запрещено, данное ограничение необходимо будет внести в схему базы данных. При физическом проектировании базы данных это ограничение необходимо возложить на СУБД, если используемая СУБД предоставляет

такую возможность[1]. Если нет, то соблюдение данного ограничения должно обеспечиваться на уровне приложения вручную, что, конечно, не способствует повышению надежности хранения данных. Данные ограничения называются ограничениями **ссылочной целостности**.

Суть нормализации

Аномалии, присутствующие в отношении на рис. 17, можно интуитивно описать следующим образом: проблемы возникают из-за того, что отношение «Секция» содержит факты, относящиеся к двум различным темам:

1. Кто из студентов какую секцию посещает.
2. Какова плата за абонемент в каждой из секций.

Когда мы добавляем новую строку, нам приходится добавлять информацию, затрагивающую две различные темы. В случае, когда мы удаляем строку, мы вынуждены удалять данные, относящиеся к двум темам сразу. Каждое нормализованное отношение имеет одну-единственную тему. Любое отношение, содержащее две или более темы, следует разбить на два или более отношения, каждое из которых будет содержать одну тему. Этот процесс составляет суть нормализации. Когда мы обнаруживаем отношение с аномалиями модификации, мы устраняем эти аномалии, разбивая отношение на два или более новых отношения, каждое из которых содержит факты, относящиеся к одной теме.

Однако не стоит забывать, что всякий раз, когда мы разбиваем отношение, мы порождаем ограничение ссылочной целостности. Поэтому следует обязательно проверять наличие таких ограничений каждый раз при разбиении отношения на два или более новых.

Классы отношений

Отношения можно классифицировать по типам аномалий, которым они подвержены. В 1970-х годах теоретики реляционных баз данных постепенно сокращали количество этих типов. Кто-то находил аномалию, классифицировал ее и думал, как предотвратить ее возникновение. Каждый раз, когда это происходило, критерии построения отношений совершенствовались. Эти классы отношений и способы предотвращения аномалий называются **нормальными формами**. В зависимости от своей структуры, отношение может быть в первой, во второй или какой-либо другой нормальной форме.

В 70-х годах были даны определения первой, второй и третьей нормальных форм (1НФ, 2НФ и 3НФ). Позднее была введена нормальная форма Бойса-Кодда (НФБК), а затем были определены четвертая и пятая нор-

[1] К сожалению, не все СУБД поддерживают ограничения ссылочной целостности, например, в FoxPro2.6 такой возможности нет.

мальные формы. Как показано на рис. **19**, эти нормальные формы являются вложенными.

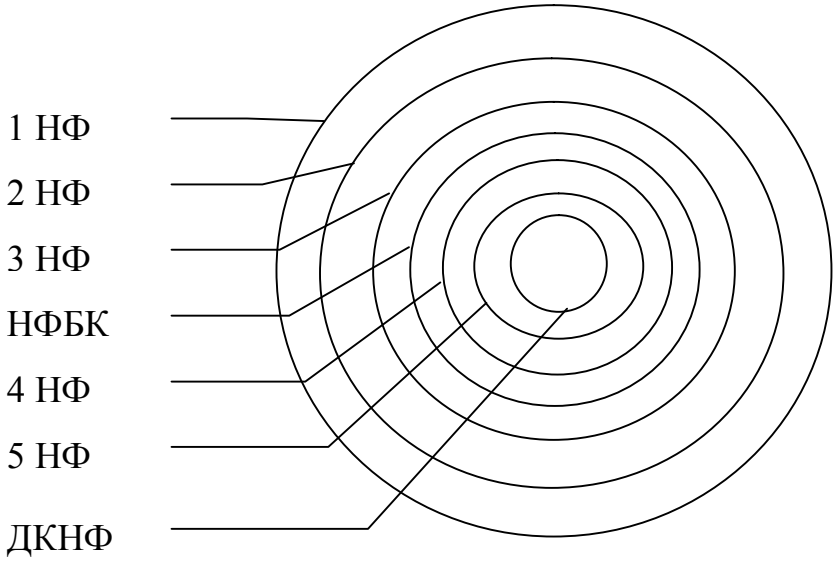

Рис. 19. Взаимосвязь нормальных форм

То есть отношение во второй нормальной форме является также отношением в первой нормальной форме, а отношение в 5НФ (пятая нормальная форма) находится одновременно в 4НФ, НФБК, 3НФ, 2НФ и 1НФ.

Эти нормальные формы помогали, но у них было и серьезное ограничение. Не было теории, гарантирующей, что какая-либо из этих форм устранит все аномалии: каждая форма могла устранить только определенные их виды. Эта ситуация разрешилась в 1981 г., когда Р. Фагин ввел новую нормальную форму, которую он назвал **доменно-ключевой нормальной формой**, или **ДКНФ**. В своей работе он показал, что отношение в ДКНФ свободно от всех аномалий, независимо от их типа. Также он показал, что любое отношение, свободное от аномалий, должно находиться в ДКНФ. Вся сложность заключается в том, что не существует методики приведения произвольного отношения к ДКНФ.

Нормальные формы

Первая нормальная форма (1НФ)

Для того, чтобы таблица находилась в первой нормальной форме, необходимо, чтобы выполнялись следующие условия:

1. Ячейки таблицы должны содержать одиночные значения и в качестве значений не допускаются ни повторяющиеся группы, ни массивы.
2. Все записи в одном столбце (атрибуте) должны иметь один и тот же тип.
3. Каждый столбец должен иметь уникальное имя, порядок следования столбцов в таблице несуществен.

4. В таблице не может быть двух одинаковых строк, порядок следования строк несуществен.

Для того, чтобы в таблице никогда не появилось двух одинаковых строк, необходимо, чтобы в таблице был уникальный идентификатор.

Вторая нормальная форма (2НФ)

Чтобы понять, что такое вторая нормальная форма, еще раз рассмотрим отношение «Секция», представленное на рисунке **17**. Однако в данном случае немного поменяем правила игры. Пусть один и тот же студент может ходить в несколько секций сразу (см. рис. **20**). В этом случае в качестве первичного ключа атрибут «Студент» в одиночестве уже выступать не может и поэтому в качестве уникального идентификатора используется составной ключ «Студент, Секция».

Студент	Секция	Плата
Иванов	Плавание	100
Иванов	Бокс	30
Петров	Плавание	100
Сидоров	Бокс	50
Скворцов	Бег	30
Скворцов	Гимнастика	200

Рис. 20. Отношение «Секция»
(один студент может ходить в несколько секций)

Из-за несколько изменившихся правил игры никак не изменились ни аномалии, ни их типы, поэтому все рассуждения, приводимые выше в равной степени справедливы и для данного примера.

Проблема с этим отношением заключается в том, что оно содержит зависимость, затрагивающую только часть ключа. Так, атрибут «Плата» зависит от атрибута «Секция» и никак не зависит от атрибута «Студент».

Определение. Говорят, что если по значению атрибута A в таблице можно однозначно определить значение атрибута B, то атрибут A является **детерминантом** для атрибута B. Данная зависимость обозначается как A⇒B.

Аномалий модификации не возникло бы, если атрибут «Плата» зависела бы от всего ключа. Чтобы устранить эти аномалии, мы должны разделить отношение на два отношения так, как это было представлено на рисунке **18**.

Данный пример приводит нас к **определению второй нормальной формы**: отношение находится во второй нормальной форме, если все его неключевые атрибуты зависят от всего ключа целиком. В соответствии с

этим определением, если отношение имеет в качестве ключа одиночный атрибут, то оно автоматически находится во второй нормальной форме. Поскольку в данном случае ключ является одиночным атрибутом, то по умолчанию каждый неключевой атрибут зависит от всего ключа, и частичных зависимостей быть не может. Таким образом, отношение «Секция», представленное на рисунке **20** определению второй нормальной формы не удовлетворяет. Отношения «Студент-Секция» и «Секция-Плата», представленные на рисунке **18**, которые получились в процессе нормализации данного отношения, определению второй нормальной уже удовлетворяют. Действительно, в отношении «Студент-Секция» неключевых атрибутов нет, а в отношении «Секция-Плата» ключевым атрибутом является одиночный атрибут.

Третья нормальная форма (3НФ)

Отношения во второй нормальной форме также могут иметь аномалии. В качестве примера рассмотрим отношение «Проживание» (см. рис. **21**).

Студент	Общежитие	Плата
Иванов	Общежитие № 1	200
Петров	Общежитие № 1	200
Сидоров	Общежитие № 2	250
Скворцов	Общежитие № 3	300

Рис. 21. Отношение «Проживание»

Естественно, один и тот же студент не может одновременно проживать сразу в нескольких общежитиях. При таком условии первичным ключом отношения вполне может выступать атрибут «Студент». В данном отношении имеется зависимость между атрибутами вида «Общежитие»⇒ «Плата», т.к. плата проживания в общежитии не зависит от того, какой студент в этом общежитии живет. Эти зависимости возникают потому, что каждый студент живет только в одном общежитии, и каждое общежитие взимает со всех проживающих в нем студентов одинаковую плату[1]. Данное отношение находится во второй нормальной форме, т.к. у него есть первичный ключ и он состоит из одиночного атрибута. Однако данное отношение не лишено аномалий вставки, редактирования и удаления[2]. Осталось разобраться с причиной возникновения аномалий в данном случае. Поскольку атрибут «Студент» является первичным ключом, то он также и детерминантом для всех неключевых атрибутов, в том числе и для атрибута «Общежитие». Однако в

[1] Данные условия берутся в качестве требований к разработке.
[2] Читателю предлагается самостоятельно провести рассуждения, которые уже приводились выше, чтобы в этом убедиться.

этом отношении имеется зависимость «Общежитие»⇒«Плата». Таким образом, атрибут через «Студент» косвенным образом определяется атрибут «Плата». Такая структура функциональных зависимостей называется **транзитивной зависимостью**, поскольку атрибут «Студент» определяет атрибут «Плата» через атрибут «Общежитие».

Определение. Если для атрибутов A, B и C некоторого отношения существуют зависимости A⇒B и B⇒C, то говорят, что атрибут C **транзитивно зависит** от атрибута A через атрибут B (при условии, что атрибут A функционально не зависит от атрибута B, ни от атрибута C).

В данном случае наличие аномалий объясняется именно наличием транзитивной зависимости в отношении. Для того, чтобы удалить аномалии из отношения во второй нормальной форме, необходимо устранить транзитивную зависимость. Таким образом, можно дать следующее **определение третьей нормальной формы**: отношение находится в третьей нормальной форме, если оно находится во второй нормальной форме и не имеет транзитивных зависимостей.

Отношение «Проживание» можно разделить на два отношения. На рисунке **22** это отношения «Студент-Общежитие» и «Общежитие-Плата».

Студент	Общежитие
Иванов	Общежитие № 1
Петров	Общежитие № 1
Сидоров	Общежитие № 2
Скворцов	Общежитие № 3

Общежитие	Плата
Общежитие № 1	200
Общежитие № 2	250
Общежитие № 3	300

Рис. 22. Отношения «Студент-Общежитие» и «Общежитие-Плата»

Уникальным идентификатором первой сущности будет «Студент», а второй – «Общежитие». Нетрудно убедиться, что в данном случае оба отношения будут находиться в третьей нормальной форме. Во-первых, у обеих таблиц есть первичные ключи, следовательно, они находятся в первой нормальной форме[1]. Т.к. данные ключи состоят из одиночных атрибутов, то отношение находится также и во второй нормальной форме. Для того чтобы в отношениях была транзитивная зависимость у данных таблиц, не хватает количества атрибутов[2], поэтому они находятся в третьей нормальной форме и в них отсутствуют аномалии, которые были у сущности «Проживание»[3].

[1] На самом деле необходимо проверить все четыре условия принадлежности первой нормальной форме.
[2] См. определение транзитивной зависимости.
[3] Читателю рекомендуется путем несложных рассуждений в этом убедиться.

Нормальная форма Бойса-Кодда (НФБК)

К сожалению, даже отношения в третьей нормальной форме могут иметь аномалии. Рассмотрим отношение «Консультант» (рис. 23). Пусть требования к этому отношению таковы:

1. Студент может консультироваться по одному или нескольким предметам.
2. Консультантами по одному и тому же предмету могут быть несколько преподавателей.
3. Каждый преподаватель может быть консультантом только по одному предмету.

Дополнительно к этому будем предполагать, что ни у преподавателей, ни и у студентов не может быть одинаковых фамилий.

Студент	Предмет	Преподаватель
Иванов	Алгебра	Попов
Иванов	СУБД	Крутов
Петров	Алгебра	Рудман
Сидоров	Методы вычислений	Сироченко
Скворцов	Методы вычислений	Сироченко

Рис. 23. Отношение «Консультант»

Поскольку студенты могут консультироваться сразу по нескольким предметам, то атрибут «Студент» не может использоваться в качестве первичного ключа. То же самое можно сказать и про атрибуты «Предмет» и «Преподаватель». Попробуем использовать различные комбинации атрибутов. Так, комбинация («Студент», «Предмет») определяет атрибут «Преподаватель», а комбинация («Студент», «Преподаватель») определяет атрибут «Предмет». Следовательно, любая из этих комбинаций может выступать в качестве первичного ключа. Два или более атрибута или группы атрибутов, которые могут быть ключом, называются **ключами-кандидатами**[1]. Кроме этого, так как любой из преподавателей может быть консультантом только по одному предмету, следовательно, зная конкретного преподавателя, мы можем определить предмет, по которому он консультирует. Таким образом, атрибут «Преподаватель» является детерминантом для атрибута «Предмет».

Отношение «Консультант» находится в первой нормальной форме[2]. Оно также находится и во второй нормальной форме, поскольку не имеет неключевых атрибутов. Каждый из его атрибутов является частью мини-

[1] Наличие в отношении нескольких ключей-кандидатов не говорит о том, что в отношении имеются аномалии какого-либо типа.

[2] Вспоминаем рассуждения про наличие первичного ключа и т.д.

мум одного ключа[1]. Наконец, это отношение находится в третьей нормальной форме, так как оно не имеет транзитивных зависимостей[2]. Тем не менее, несмотря на все это, отношение имеет аномалии[3].

Проблема в данном случае заключается в том, что атрибут «Преподаватель», являющийся детерминантом, не является ни первичным ключом, ни ключом-кандидатом. Данная ситуация приводят нас к **определению нормальной формы Бойса-Кодда**: отношение находится в НФБК, если каждый детерминант является ключом-кандидатом.

Согласно данному определению, отношение «Консультант» не находится в нормальной форме Бойса-Кодда. Как и в других примерах, в процессе нормализации отношение «Консультант» разобьем на два отношения, не имеющие аномалий – отношения «Студент-Преподаватель» и «Преподаватель-Предмет», представленные на рис. **24**.

Студент	Преподаватель
Иванов	Попов
Иванов	Крутов
Петров	Рудман
Сидоров	Сироченко
Скворцов	Сироченко

Преподаватель	Предмет
Попов	Алгебра
Крутов	СУБД
Рудман	Алгебра
Сироченко	МВ

Рис. 24. Отношения «Студент-Преподаватель» и «Преподаватель-Предмет»

Первичным ключом первой сущности будет совокупность атрибутов «Студент-Преподаватель», а второй – «Преподаватель». Нетрудно убедиться, что в данном случае оба отношения будут находиться в НФБК, т.к. у первой сущности все ее атрибуты входят в первичный ключ, а у второй – атрибут «Преподаватель» является детерминантом для атрибута «Предмет» и является первичным ключом[4].

Сущности «Студент-Преподаватель» и «Преподаватель-Предмет» лишены тех аномалий, которые были у сущности «Консультант», необходимые рассуждения для этого уже приводились выше[5].

Отношения в НФБК не имеют аномалий, относящихся к функциональным зависимостям, и на какой-то момент времени казалось, что вопрос с

[1] Один из ключей – первичный, другой – ключ-кандидат.

[2] Для того, чтобы в этом убедиться необходимо рассмотреть различные комбинации возможных зависимостей атрибутов отношения между собой.

[3] Внимательный читатель без труда воспроизведет все соответствующие рассуждения, приводимые ранее по этому поводу.

[4] В рассуждении сознательно опущены рассуждения по поводу того, что данные отношения находятся в том числе и в 3 НФ с тем, чтобы еще раз побудить читателя проделать эти рассуждения самостоятельно.

[5] Если читатель их забыл, то можно еще раз перечитать раздел «Типы аномалий».

аномалиями на этом исчерпан. Однако вскоре обнаружилось, что аномалии могут быть обусловлены и иными причинами, нежели функциональные зависимости и, следовательно, появились новые нормальные формы.

Четвертая нормальная форма (4НФ)

Рассмотрим отношение «Студент», представленное на рисунке 25. Она отображает связи между студентами, предметами, в которых они специализируются и секциями, в которых они занимаются[1].

Студент	Предмет	Секция
Иванов	Алгебра	Бокс
Иванов	СУБД	Бокс
Иванов	Алгебра	Плавание
Иванов	СУБД	Плавание
Петров	Методы вычислений	Бокс
Скворцов	Методы вычислений	Гимнастика

Рис. 25. Отношение «Студент»

Предположим, что студенты могут специализироваться сразу в нескольких дисциплинах и занимаются в нескольких различных секциях[2]. В этом случае первичным ключом является комбинация («Студент», «Предмет», «Секция»). Например, студент Иванов специализируется в алгебре и СУБД и, кроме этого, посещает секции бокса и плавания, а студент Петров специализируется только на методах вычислений и занимается боксом.

Прежде всего, необходимо разобраться со связью между атрибутами «Студент» и «Предмет». Это не функциональная зависимость, поскольку у студента может быть несколько дисциплин специализации. Одному и тому же значению атрибута «Студент» может соответствовать много значений атрибута «Предмет». Помимо того, одному и тому же значению атрибута «Студент» может соответствовать много значений атрибута «Секция». Такая зависимость атрибутов называется многозначной зависимостью.

Определение. Многозначной зависимостью между атрибутами A, B и C называется случай, когда для каждого значения атрибута A имеется набор значений атрибута B и набор значений атрибута C. Однако значения атрибутов B и C не зависят друг от друга.

[1] Если посмотреть на рисунок, то можно сказать, что это двойная связь «один ко многим», реализованная в рамках одной таблицы.

[2] Несколько идеализированный случай, не правда ли?

Многозначные зависимости приводят к аномалиям модификации. Однако в данном случае имеет место также избыточность данных (см. рис. 25). Студенту Иванову в таблице посвящено четыре записи, в каждой из которых указана одна из его специализаций и одна из посещаемых им секций. Если бы те же данные хранились в меньшем количестве строк (скажем, было бы две строки - одна для алгебры и бокса, а другая для СУБД и плавания), это дезориентировало бы пользователей. Получалось бы, что студент Иванов посещает секцию бокса только тогда, когда специализируется на алгебре, а плавает только тогда, когда специализируется на базах данных. Но такая интерпретация нелогична. Предметы и секции совершенно независимы друг от друга. Поэтому, чтобы избежать таких неверных заключений, необходимо хранить все сочетания предметов и секций. Предположим, что студент Иванов решил записаться еще в секцию гимнастики, и поэтому мы добавляем в таблицу строку (“Иванов”, “Алгебра”, “Гимнастика”). В данный момент из отношения можно сделать вывод, что студент Иванов занимается гимнастикой только при специализации алгеброй, но никак не СУБД. Чтобы данные имели согласованный характер, мы должны добавить столько строк, сколько имеется специализаций, и в каждой из них указать секцию гимнастики. Это **аномалия обновления**: требуется слишком много модификаций, чтобы внести одно простое изменение.

Многозначные зависимости обозначаются следующим образом: «Студент» >> «Предмет» и «Студент» >> «Секция». Это читается следующим образом: атрибут «Студент» многозначно определяет атрибут «Предмет», и атрибут «Студент» многозначно определяет атрибут «Секция».

Отношение «Студент» находится в НФБК[1]. Однако, как мы убедились, оно имеет аномалии: если студент берет дополнительный предмет специализации, мы должны добавить в отношение столько строк, в скольких секциях данный студент занимается, и в каждой из этих строк указать новый предмет. То же самое справедливо и для случая, когда студент записывается в новую секцию. Если студент отказывается от одного из своих предметов специализации, мы должны будем удалить все строки, где указан данный предмет.

Исходя из сделанных нами наблюдений, мы **определим четвертую нормальную форму** следующим образом: отношение находится в четвертой нормальной форме, если оно находится в НФБК и не имеет многозначных зависимостей.

Чтобы устранить эти аномалии, мы должны избавиться от многозначной зависимости. Для этого необходимо создать два отношения, в каждом из которых будут храниться данные только по одному многозначному атрибуту. Результирующие отношения не будут иметь аномалий. Это отно-

[1] 1НФ – поскольку существует первичный ключ, 2НФ - поскольку нет неключевых атрибутов, 3НФ – так как отсутствуют транзитивные зависимости, и НФБК – поскольку нет неключевых детерминантов.

шения «Студент-Предмет» и «Студент-Секция», приведенные на рис. **26**. Первичным ключом первой сущности будет совокупность атрибутов («Студент», «Предмет»), а второй – («Студент», «Секция»). Нетрудно убедиться, что данные отношения уже удовлетворяют четвертой нормальной форме[1].

Студент	Предмет
Иванов	Алгебра
Иванов	СУБД
Петров	МВ
Скворцов	МВ

Студент	Секция
Иванов	Бокс
Иванов	Плавание
Петров	Бокс
Скворцов	Гимнастика

Рис. 26. Отношения «Студент-Предмет» и «Студент-Секция»

Пятая нормальная форма (5НФ)

Ранее при любой декомпозиции отношений[2] на два других отношения полученные отношения обладали свойством соединения без потерь[3]. Это значит, что полученные отношения можно было снова соединить и получить прежнее отношение в исходном виде.

Однако бывают случаи, когда требуется декомпозировать отношение на более чем два отношения. В таких (достаточно редких) случаях возникает необходимость учитывать зависимость соединения, которая устраняется с помощью пятой нормальной формы (5НФ). Однако сначала необходимо ввести несколько новых терминов.

Декартово произведение – это попарная конкатенация[4] всех строк одного отношения со всеми строками другого отношения. Декартово произведение обозначается как A×B, где A – первое отношение, а B – второе.

Приведем примеры декартовых произведений. Для отношений, не имеющих одинаковых атрибутов, соответствующий пример представлен на рисунке **27**.

Как видно из приведенного рисунка, декартово произведение отношений, не имеющих одинаковых атрибутов, представляет собой полный перебор всех значений одного отношения со всеми значениями другого отношения.

[1] Таблицы всего с двумя атрибутами очень удобны: количества атрибутов не хватает ни для транзитивной зависимости, ни для многозначной.

[2] Т.е. при разбиении отношения на несколько других отношений.

[3] Т.е. при желании из данных, представленных в «мелких» отношениях можно получить данные, представленные ранее в «крупном» отношении.

[4] Другими словами, объединение.

A:

C	D
1	2
3	4

B:

E	F
5	6
7	8

A×B:

C	D	E	F
1	2	5	6
1	2	7	8
3	4	5	6
3	4	7	8

Рис. 27. Пример декартова произведения для отношений, не имеющих одинаковых атрибутов

Аналогичная ситуация имеет место также для случаев, если в отношениях имеются какие-либо общие атрибуты. Поясним это на примере таблиц, уже ранее рассматривавшихся, например, найдем декартово произведение отношений «Студент-Преподаватель» и «Преподаватель-Предмет», представленных на рис. 24. Результатом будет таблица, состоящая из атрибутов «Студент», «Преподаватель» и «Предмет»[1]. Т.к. атрибут «Преподаватель» для обоих отношений является общим, то поступают следующим образом. Идут по строкам первого отношения, в данном случае это отношение «Студент-Преподаватель», начиная с первого. Это запись (“Иванов”, “Попов”). Далее во второй таблице ищутся все записи, у которых атрибут «Преподаватель» установлен в значение “Попов”. В данном случае это только одна запись, и поэтому в декартово произведение добавится только строка (“Иванов”, “Попов”, “Предмет”)[2]. Далее необходимо перейти ко второй строке отношения «Студент-Преподаватель» и так далее. В результате получится таблица, которая по наполнению полностью совпадает с отношением «Консультант», представленным выше на рисунке 23.

Таким образом, мы обнаружили важный факт. В результате процесса нормализации большие таблицы, имеющие аномалии разбиваются на более мелкие, в которых аномалий уже нет. Однако если нам когда-нибудь потребуется восстановить большую таблицу, мы это без проблем сделаем путем нахождения декартова произведения всех отношений, которые появились в результате процесса нормализации. Однако так бывает не всегда и появление пятой нормальной формы связано именно с этим. Сначала введем вспомогательное определение.

[1] Таким образом, множество столбцов в декартовом произведении определяется как объединение множеств столбцов первого и второго отношений. Таким образом, повторяющиеся атрибуты входят в результирующее отношение только один раз.

[2] В случае, если бы таких записей было несколько, например, 3 то в декартовом произведении появилось бы три записи, которые соответствуют первой строке первого отношения и всем строкам второго отношения.

Определение. **Зависимость соединения –** это свойство декомпозиции, которое вызывает генерацию ложных строк при обратном соединении.

Во всех вышеприведенных примерах исходные отношения были декомпозированы на два отношения, однако бывают случаи, когда требуется выполнить декомпозицию без потерь с образованием более чем двух отношений. Именно в таких случаях применимы понятия зависимости соединения и пятой нормальной формы (5НФ).

Определение Отношение находится в **пятой нормальной форме (5НФ)**, если оно находится в четвертой и не имеет зависимостей соединения[1].

Для того, чтобы убедиться, что зависимость соединения может существовать на практике, рассмотрим отношение «Поставка мебели», представленное на рисунке 28. В данном отношении представлены данные по различным объектам недвижимости, в которые поставляется мебель различными поставщиками.

Недвижимость	Мебель	Поставщик
1	Кровать	Иванов
1	Стул	Петров
2	Кровать	Петров
1	Кровать	Петров
3	Тумба	Сидоров

Рис. 28. Отношение «Поставка мебели»

Пусть в качестве ограничения[2] используется условие о том, чтобы один поставщик мог поставлять только определенные типы мебели для заданного объекта недвижимости[3].

В качестве первичного ключа в данном случае можно использовать только совокупность сразу трех атрибутов («Недвижимость», «Мебель», «Поставщик»), т.к. никакая совокупность, ни одиночных атрибутов, ни парных, не является уникальной комбинацией[4]. Нетрудно убедиться, что

[1] Ввиду того, что ранее при всех проводимых процессах нормализации зависимость соединения отсутствовала, следовательно, все получавшиеся нормальные формы удовлетворяли, в том числе условию пятой нормальной формы (естественно при условии нахождения отношения в 4НФ).

[2] По-другому это называется бизнес-правилом.

[3] Т.е., например, поставщик Иванов поставляет в объект недвижимости 1 только кровати, а поставщик Петров поставляет туда же стулья и кровати, хотя он же в объект недвижимости 2 поставляет только кровати.

[4] Это видно, в частности, из представленного наполнения отношения.

данное отношение удовлетворяет четвертой нормальной форме[1]. Тем не менее, аномалии в данном отношении присутствуют[2].

Ограничение о том, чтобы один поставщик может поставлять только определенные типы мебели для заданного объекта недвижимости, создает в отношении «Поставка мебели» зависимость соединения. Однако структура этого отношения не соответствует данному ограничению, поскольку допускается добавление в нее любых строк[3].

Поскольку отношение «Поставка мебели» содержит зависимость соединения, значит, оно не находится в пятой нормальной форме. Для удаления этой зависимости соединения следует выполнить декомпозицию данного отношения. Если в приводимых ранее примерах достаточно было одну таблицу разделить на два отношения, то в данном случае этого уже становится недостаточно. Декартово произведение любых двух отношений приведет к появлению ложных строк[4]. Поэтому в данном случае только разбиение таблицы на три отношения позволит восстановить информацию, находящуюся в сущности «Поставка мебели». Ниже на рисунке **29** приводятся соответствующие отношения.

Недви-жимость	**Мебель**
1	Кровать
1	Стул
2	Кровать
3	Тумба

Недви-жимость	**Поставщик**
1	Иванов
1	Петров
2	Петров
3	Сидоров

Мебель	**Поставщик**
Кровать	Иванов
Стул	Петров
Кровать	Петров
Тумба	Сидоров

Рис. 29. Отношения «Недвижимость-Поставщик», «Недвижимость-Поставщик» и «Мебель-Поставщик»

Доменно-ключевая нормальная форма (ДКНФ)

Все описанные выше нормальные формы были выделены исследователями, выявившими аномалии у некоторых отношений, которые находились в нормальной форме более низкого порядка. Так, обнаружение аномалий модификации у отношений во второй нормальной форме привело к определению третьей нормальной формы и т. д. Хотя каждая нормальная форма решала часть проблем, найденных у предыдущей нормальной формы, никто не мог сказать, какие проблемы еще не выявлены. Каждый такой шаг являлся прогрессом на пути к представлению об оптимальной

[1] Автор очень надеется, что для читателя это действительно уже нетрудно.
[2] Имеются ввиду аномалии вставки, редактирования и удаления.
[3] Например, строку (1, “Стул”, “Иванов”), хотя уже говорилось, что в объект недвижимости “1” поставщик Иванов может поставлять только кровати. Таким образом, ограничение продекларировано, но никак на уровне самой базы данных не реализовано.
[4] Читателю настоятельно рекомендуется проверить этот факт самостоятельно.

структуре базы данных, однако никто не мог гарантировать, что не будет обнаружено еще каких-либо аномалий. Таким образом, последовательно появилось шесть нормальных форм: 1НФ, 2НФ, 3НФ, НФБК, 4НФ, 5НФ.

В 1981 г. Р. Фагин опубликовал статью, в которой он определил доменно-ключевую нормальную форму (ДКНФ). Он показал, что отношение в ДКНФ не имеет аномалий модификации и, более того, любое отношение, не имеющее аномалий модификации, должно находиться в ДКНФ. Это открытие положило конец введению новых нормальных форм, и теперь в нормальных формах более высокого порядка нет необходимости – по крайней мере, для устранения аномалий модификации.

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

Термин «**ограничение»** имеет в данном определении намеренно широкое толкование. В данном случае это любое правило, регулирующее возможные статические значения атрибутов и достаточно точное для того, чтобы можно было установить, выполняется оно или нет. Правила редактирования, ограничения взаимоотношений и структуры отношений, функциональные зависимости и многозначные зависимости являются примерами таких ограничений. Однако ограничения, относящиеся к изменениям значений данных, или ограничения, зависящие от времени, к ограничениям не относятся. Например, правило "Зарплата сотрудника за текущий период не может быть меньше, чем за предыдущий период" не подпадает под определение ограничения.

Домен – это описание допустимых значений атрибута. Он состоит из двух частей: физического описания и логического описания. Физическое описание – это множество значений, которые может принимать атрибут, а логическое описание – это смысл данного атрибута.

Говоря неформально, отношение находится в ДКНФ, если выполнение ограничений на домены и ключи приводит к выполнению всех ограничений. Более того, поскольку отношения в ДКНФ не могут иметь аномалий модификации, СУБД может предотвратить возникновение этих аномалий, реализуя выполнение ограничений на домены и ключи.

К сожалению, пока не известен алгоритм преобразования отношения к доменно-ключевой нормальной форме. Неизвестно даже, какие отношения могут быть приведены к ДКНФ. Нахождение и создание отношений в ДКНФ является более искусством, чем наукой.

Несмотря на все это, при практическом проектировании баз данных ДКНФ служит в высшей степени полезным ориентиром. Если мы сможем

вводить отношения таким образом, что все налагаемые на них ограничения являются логическими следствиями доменов и ключей, то в таких отношениях не будет аномалий модификации. Во многих случаях этот критерий может быть соблюден. Если же выполнить его на уровне СУБД не представляется возможным, необходимые ограничения должны быть встроены в логику прикладных программ, которые обрабатывают базу данных.

Пример доменно-ключевой нормальной формы

Рассмотрим отношение «Студент» на рис **30**, имеющее атрибуты «Номер студента», «Курс», «Общежитие» и «Плата»[1]. Здесь «Плата» - это сумма, которую студент платит за проживание в общежитии.

Студент (Номер студента, Курс, Общежитие, Плата)
Ключ: Номер студента
Ограничения: 1. Общежитие ⇒ Плата
2. Номер студента не должен начинаться с цифры 1

Рис. 30. Отношение «Студент»

Атрибут «Номер Студента» функционально определяет остальные три атрибута, поэтому он является ключом. Допустим, что согласно требованиям пользователей, «Общежитие» ⇒ «Плата», а «Номер Студента» не должен начинаться с единицы. Если нам удастся представить эти требования в качестве логических следствий определения доменов и ключей, то мы сможем быть уверены, что никаких аномалий модификации не возникнет. В данном примере это будет легко сделать.

Чтобы реализовать ограничение, указывающее, что номера студентов не должны начинаться с 1, мы просто включим это требование в определение домена атрибута «Номер Студента» (рис. **31**). Соблюдение ограничения домена гарантирует, что данное требование будет выполнено.

Далее нам требуется сделать функциональную зависимость «Общежитие» ⇒ «Плата» логическим следствием ключей. Если бы атрибут «Общежитие» был ключевым, то данная зависимость была бы логическим следствием ключа. Поэтому встает вопрос о том, как сделать атрибут «Общежитие» ключом. Данный атрибут не может быть ключом в отношении «Студент», так как в одном и том же общежитии живет более одного студента, однако он может быть ключом своего собственного отношения. Таким образом, мы можем ввести отношение «Общежитие-Плата» с атрибутами «Общежитие» и «Плата». Общежитие будет являться ключом этого

[1] Данное отношение несколько напоминает уже приводившееся ранее отношение.

отношения. Введя данное отношение, мы сможем удалить атрибут «Плата» из отношения «Студент». Окончательные определения доменов и отношений для этого примера представлены на рис. **31**.

Определения доменов

Номер студента	CDDD, где C – цифра, не равная 1; D – десятичная цифра
Курс	{“1”, “2” , “3” , “4” , “5”}
Общежитие	CHAR(10)
Плата	DECIMAL(4)

Определение отношений и ключей

Студент (Номер студента, Курс, Общежитие)
Ключ: Номер студента

Общежитие-Плата (Общежитие, Плата)
Ключ: Общежитие

Рис. 31. Пример отношений, находящихся в ДКНФ

Этот же результат мы получили, преобразовав отношение из второй нормальной формы в третью для удаления транзитивных зависимостей. В этом случае, однако, процесс был проще, а результат является более надежным. Процесс был проще потому, что нам не требовалось знать, что мы устраняем транзитивную зависимость. Все, что нам было нужно – это найти действенные способы превращения всех ограничений в логические следствия определений доменов и ключей. Результат является более надежным потому, что при преобразовании отношения в третью нормальную форму мы знали только то, что количество аномалий в нем стало меньше по сравнению со второй нормальной формой. Приведя отношение к ДКНФ, мы точно знаем, что получившиеся отношения не будут иметь никаких аномалий модификации.

Денормализация

Как уже говорилось, нормализованные отношения свободны от аномалий модификации, и по этой причине они являются более предпочтительными, чем ненормализованные отношения. Но если судить с других позиций, иногда нормализация не стоит того, чтобы ее проводить.

Рассмотрим отношение «Клиент», состоящее из атрибутов («Номер Клиента», «Имя Клиента», «Город», «Индекс»), где ключом является «Номер Клиента». Это отношение не находится в ДКНФ, поскольку в нем имеется функциональная зависимость «Индекс» ⇒ «Город», которая не следует из ключа - атрибута «Номер Клиента». Следовательно, мы имеем ограничение, не являющееся следствием определения ключей.

Данное отношение может быть преобразовано в два отношения следующего вида: «Клиент» («Номер Клиента», «Имя Клиента», «Индекс»), где ключом является «Номер Клиента», и «Индексы» («Индекс», «Город»), где ключом является «Индекс». Эти два отношения находятся в доменно-ключевой нормальной форме, но они, скорее всего, не представляют собой более удачного решения по сравнению с исходной таблицей. Возможно, что ненормализованная таблица лучше, поскольку ее будет легче обрабатывать, а неудобства, связанные с дублированием данных о городе и штате, не столь существенны.

В общем, иногда отношения намеренно оставляют в ненормализованном виде либо нормализуют, а затем денормализуют. Зачастую это делается для повышения производительности. Всегда, когда необходимо комбинировать данные из двух различных таблиц, СУБД должна выполнять дополнительную работу. В большинстве случаев для этого требуется как минимум две операции чтения вместо одной.

Преднамеренная избыточность

Одним из преимуществ нормализованных отношений является то, что в них минимизируется дублирование данных (только значения ключей появляются более чем в одном отношении). Но в целях повышения производительности иногда уместным является умышленное дублирование данных. Рассмотрим, например, приложение, обрабатывающее заказы, которое обращается к таблице «Товар», имеющей следующие столбцы («Номер товара», «Наименование» , «Цвет», … , «Описание», «Фотография»).

Предположим, что атрибут «Номер Товара» является первичным ключом, и что таблица находится в ДКНФ. Предположим также, что атрибут «Описание» - это потенциально длинное текстовое поле, а «Фотография» - это столбец двоичных данных длиной минимум 256 Кбайт.

Приложению, обрабатывающему заказы, необходимо будет обратиться к этой таблице, чтобы получить значения атрибутов «Наименование», «Цвет», «Цена» и «Количество». Допустим, что приложению не требуется значения атрибутов «Описание» или «Фотография». В зависимости от характеристик используемой СУБД, возможно, что присутствие этих двух больших столбцов значительно замедлит обработку. В этой ситуации проектировщики базы данных могут решить продублировать некоторые данные во второй таблице, которая будет содержать только ту информацию, которая

требуется для обработки заказа. К примеру, они могут создать таблицу под названием «Заказ товара» с атрибутами, относящимся к сущности «Товар» – («Номер Товара», «Наименование», «Цвет», «Цена», «Количество»), которая будет использоваться только приложением, обрабатывающим заказы.

В этом случае проектировщики создают потенциальный источник серьезных проблем, связанных с целостностью данных. Им придется разработать процедуры как программного, так и ручного контроля, чтобы гарантировать, что таких проблем не возникнет. Поэтому к подобному решению они прибегнут только в том случае, если, по их мнению, повышение производительности окупит расходы, связанные с дополнительным контролем, и риск возникновения проблем с целостностью данных.

Еще одна причина для введения преднамеренной избыточности – это создание таблиц, предназначенных исключительно для создания отчетов и поддержки принятия решений[1].

Пример денормализованной структуры базы данных

В данном разделе приводится пример структуры денормализованной базы данных. Если рассматривать эту структуру с точки зрения именно проектирования, то это пример того, как «делать не надо», т.к. некоторые сущности в ней не удовлетворяют даже первой нормальной форме. Тем не менее, программа, работавшая с этой базой данных, успешно эксплуатировалась в книготорговой фирме в течение семи лет[2].

Итак, организация занималась оптово-розничной продажей книг. Операторами, находящимися в торговом зале, регистрировались заявки клиентов, далее по которым со склада производился отпуск продукции. Клиентам позволялось покупать книги в кредит и потом за них расплачиваться. Компьютеров в организации было всего три[3], объединены они были в одно-ранговую сеть с помощью коаксиального кабеля. Для пользователей было важно, чтобы заказы формировались как можно быстрее и поэтому требования к быстродействию были весьма жесткие.

Для реализации программы была выбрана СУБД FoxPro 2.6 для Windows, которая хорошо годилась именно для небольшого количества одновременно работающих пользователей. Ввиду того, что в FoxPro 2.6 очень многие средства не были реализованы на уровне базы данных[4], при-

---

[1] На практике очень часто для подобных отчетов устанавливают временные критерии для вывода информации. Как правило, данные отчеты генерируются по окончании финансового года или другого отчетного периода и сразу в больших количествах. Именно поэтому очень часто для их реализации используется так или иначе процесс денормализации.

[2] На самом деле структура базы данных была более сложная, однако основополагающие сущности на диаграмме приведены.

[3] Один компьютер был 486-й, два других – Pentium I.

[4] К ним относятся ссылочная целостность, резервное копирование и т.п.

шлось ряд вспомогательных процедур реализовывать на уровне приложения. В результате после определенного периода времени[1] появилась схема базы[2], представленная на рисунке **32**.

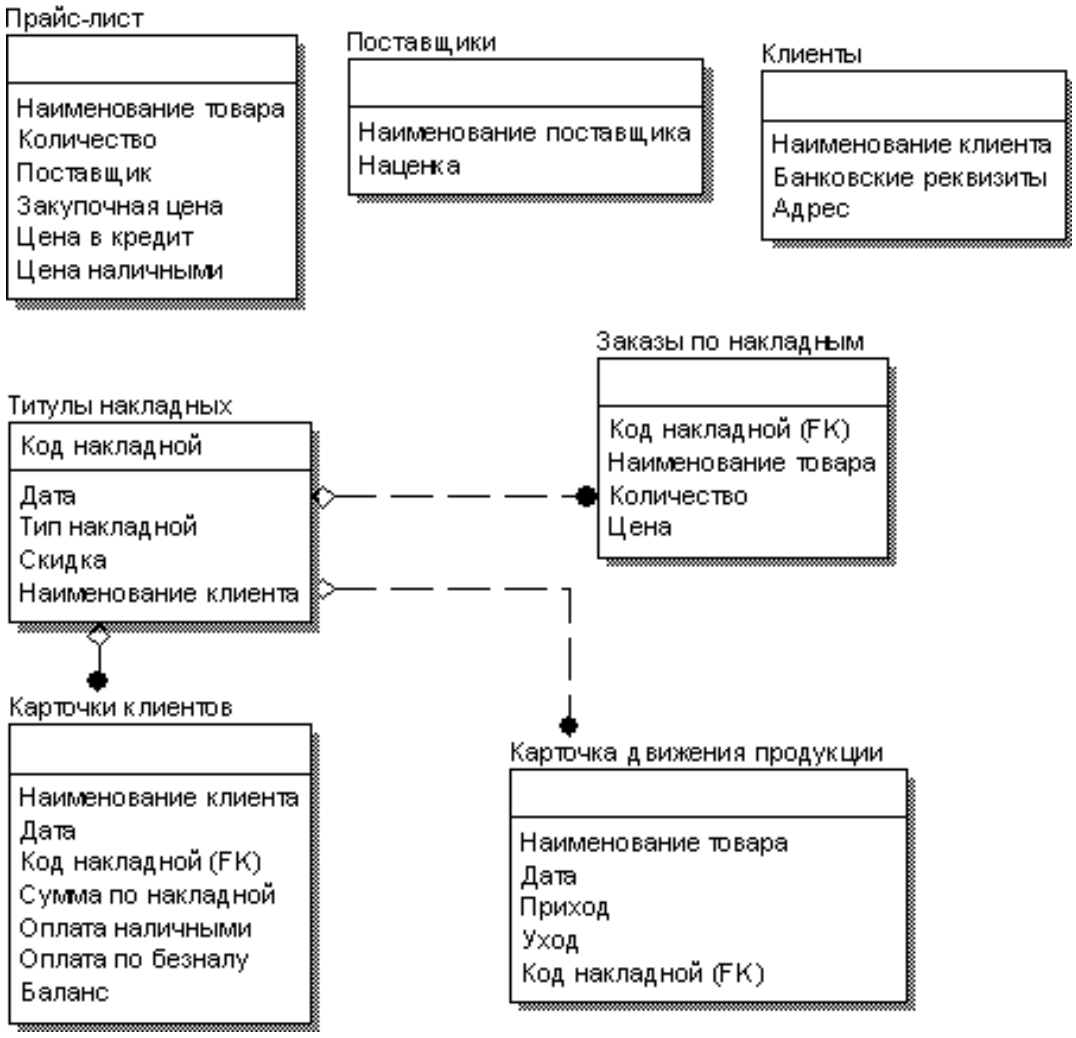

Рис. 32. Пример денормализованной структуры базы данных

При создании заказа сначала добавляется запись в таблицу «Титулы накладных», а конкретное наполнение заказа вносится в таблицу «Заказы по накладным». При подборе наименований продукции осуществляется выборка из таблицы «Прайс-лист». Напрямую таблица «Прайс-лист» ни с кем не связана, тем не менее, данные из этой таблицы заносятся в таблицы «Заказы по накладным» и «Карточка движения продукции». Это сделано сознательно, т.к. ассортимент продукции в организации очень часто обновлялся и поэтому нельзя было запрещать удалять записи из таблицы «Прайс-лист». При наличии связей между различными сущностями при

[1] Процесс создания структуры базы данных – итерационный процесс.

[2] На самом деле в тот период никаких схем баз данных не создавалось, и база данных делалась больше интуитивно, чем с помощью каких-либо теоретических знаний в области проектирования.

удалении какой-либо записи из таблицы «Прайс-лист» терялась бы информация о соответствующем наименовании продукции, которая была куплена по какой-либо накладной. Поэтому в данном случае имела место сознательная денормализация структуры базы данных с тем, чтобы реализовать требования пользователей и одновременно с этим достигнуть высокого уровня быстродействия. Впоследствии пользователи захотели анализировать информацию о том, какая продукция пользуется большим или меньшим спросом. Для того, чтобы соответствующая информация выводилась как можно быстрее, была создана таблица «Карточка движения продукции», информация в которую заносилась при формировании заказов пользователей и при закупках продукции у поставщиков.

Таким образом, на практике[1] очень часто проектируют базы данных с учетом различных критериев, таких, как быстродействие системы, а не только согласованность данных. При создании же информационных систем для крупных предприятий, как правило, основным критерием все же является надежность хранения данных и их непротиворечивость[2].

[1] Особенно это имеет место при проектировании приложений для небольших организаций, в которых есть ограничения на аппаратное обеспечение и требования по выбору определенной СУБД.

[2] При создании серьезных информационных систем вопросам нормализации уделяют достаточно большое внимание. Вопросы быстродействия же решаются с помощью вспомогательных средств, таких как индексы, партиции и т.п.

5. ЖИЗНЕННЫЙ ЦИКЛ ПРИЛОЖЕНИЯ БАЗ ДАННЫХ

В настоящее время ключевая роль в достижении успеха большинства компьютеризованных систем принадлежит не используемому оборудованию, а программному обеспечению. В последние десятилетия прикладные программы проделали путь от маленьких и сравнительно простых приложений из нескольких строк кода до очень больших и сложных приложений, состоящих из нескольких миллионов строк. Усилия и ресурсы, затрачиваемые на сопровождение программного обеспечения, возрастали угрожающими темпами. Все это привело к ситуации, которая известна под названием "кризис программного обеспечения".

Неудачи при создании программного обеспечения были вызваны:

- отсутствием полной спецификации всех требований[1];
- отсутствием приемлемой методологии разработки;
- недостаточной степенью разделения общего глобального проекта на отдельные компоненты, поддающиеся эффективному контролю и управлению.

Для разрешения этих проблем был предложен структурный подход к разработке программного обеспечения, называемый жизненным циклом разработки программного обеспечения

Ниже на рисунке **33** представлена схема жизненного цикла приложения баз данных. Далее кратко рассматриваются ее основные этапы.

Планирование разработки базы данных

На данном этапе осуществляются подготовительные действия, позволяющие с максимально возможной эффективностью реализовать этапы жизненного цикла приложения баз данных.

Как и в случае создания прочего программного обеспечения, планирование разработки базы данных состоит в определении трех основных компонентов: требуемого объема работы, необходимых ресурсов и общей стоимости проекта. Планирование разработки базы данных должно быть связано с общей стратегией построения информационной системы организации[2].

[1] Т.е. неспособностью пользователей четко сформулировать все требования к проектируемой системе. Кроме этого, за время разработки, требования способны меняться кардинальным образом, что также сказывается отрицательным образом на времени создания информационной системы.

[2] В стратегию могут входить требования к выбору СУБД для информационных систем, правила интеграции различных информационных систем, способы обмена информацией и т.п.

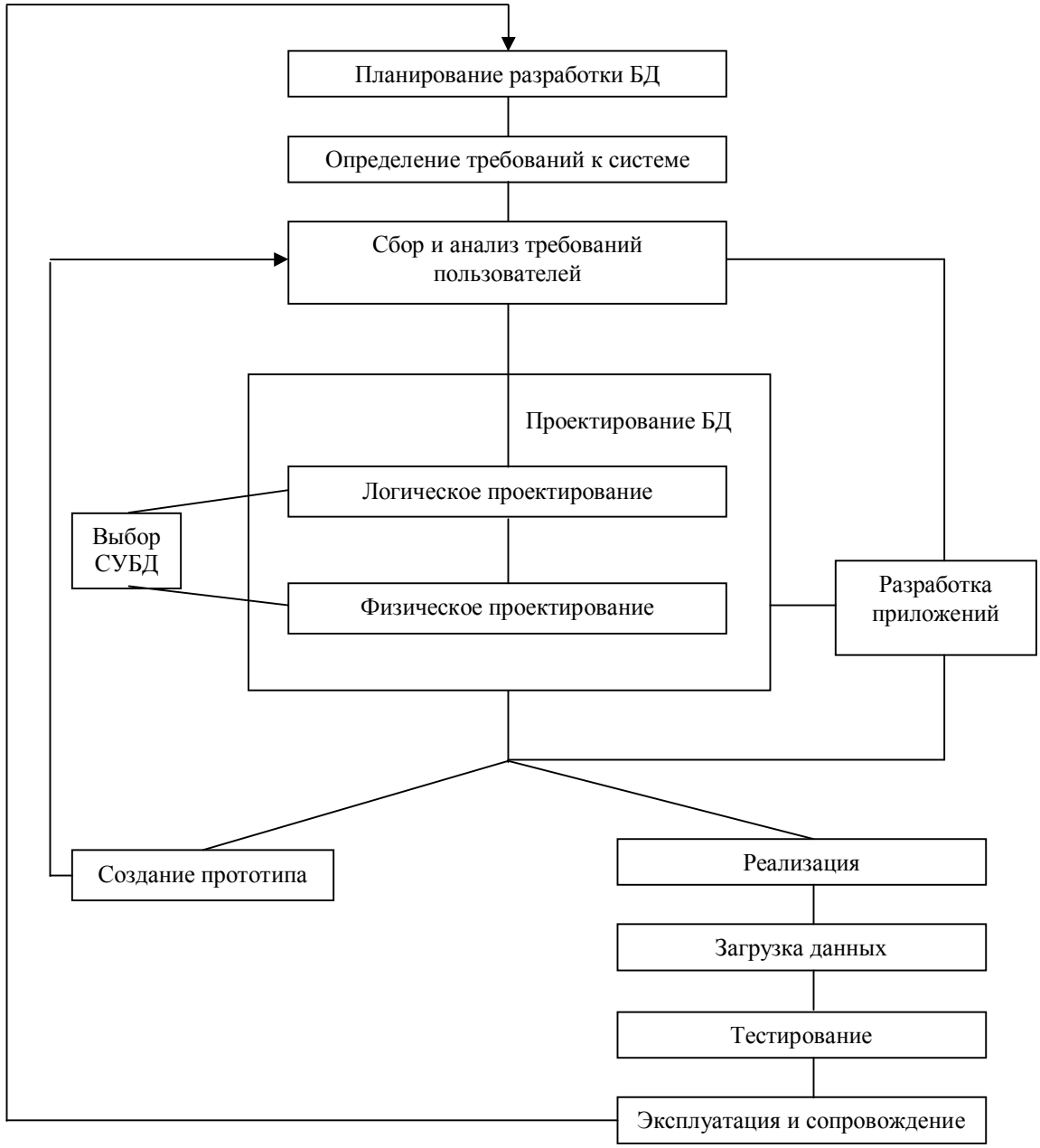

Рис. 33. Жизненный цикл приложения баз данных

Определение требований к системе

В данном случае определяется диапазон действия и границ приложения базы данных, состава его пользователей и областей применения.

Сбор и анализ требований пользователей

Данный этап является очень важным, т.к. формулирование требований пользователей, их систематизация является весьма сложной и ответственной задачей, влияющей на всю последующую разработку. На данном этапе происходит сбор и анализ информации о той части организации, ра-

бота которой будет поддерживаться с помощью создаваемого приложения базы данных, а также использование этой информации для определения требований пользователей к создаваемой системе. Результатом выполнения работ по данному этапу является техническое задание на разработку[1].

Проектирование базы данных

По результатам требований пользователей на данном этапе осуществляется проектирование базы данных, предназначенной для поддержки функционирования предприятия и способствующей достижению его целей.

К основным целям этапа проектирования относятся:

- представление данных и связей между ними, необходимых для всех основных областей применения данного приложения и любых существующих групп его пользователей;
- создание модели данных, способной поддерживать выполнение любых требуемых транзакций[2] обработки данных;
- разработка предварительного варианта проекта, структура которого позволяет удовлетворить все основные требования, предъявляемые к производительности системы.

К сожалению, эти цели легко достижимы далеко не всегда, и в некоторых случаях приходится идти на компромисс - например, для достижения приемлемого уровня производительности системы.

Существует два основных подхода к проектированию систем баз данных: "**нисходящий**" и "**восходящий**"[3].

При **восходящем** подходе работа, начинается с самого нижнего уровня – уровня определения атрибутов (т.е. свойств сущностей), которые на основе анализа существующих между ними связей группируются в отношения, представляющие типы сущностей и связи между ними. Восходящий подход лучше всего подходит для проектирования простых баз данных с относительно небольшим количеством атрибутов. Однако использование этого подхода существенно усложняется при проектировании баз данных с большим количеством атрибутов, установить среди которых все существующие функциональные зависимости довольно затруднительно. Кроме того, на начальных стадиях формулирования требований к данным для крупных информационных систем может быть весьма затруднитель-

[1] Техническое задание может писаться самим заказчиком, но, как правило, оно составляется фирмой-разработчиком и согласовывается с организацией-заказчиком системы.
[2] Т.е. чтобы все требуемые бизнес-процессы, осуществляемые в организации, производились с помощью инструментария информационной системы.
[3] Точно также существует «нисходящий» и «восходящий» стиль программирования.

ным установить все атрибуты, которые должны быть включены в модель данных.

Более подходящей стратегией проектирования сложных баз данных является использование **нисходящего** подхода. Этот подход начинается с разработки моделей данных, которые содержат несколько высокоуровневых сущностей и связей, затем работа продолжается в виде серии нисходящих уточнений низкоуровневых сущностей, связей и относящихся к ним атрибутов. Нисходящий подход демонстрируется в концепции модели "сущность-связь" уже рассматриваемый ранее.

Проектирование базы данных состоит из двух основных фаз: логического и физического проектирования[1].

Логическое проектирование базы данных

Под **логическим проектированием** базы данных понимается процесс создания модели используемой на предприятии информации с учетом выбранной модели организации данных, но независимо от типа целевой СУБД и других физических аспектов реализации.

В процессе разработки логическая модель данных постоянно тестируется и проверяется на соответствие требованиям пользователей. Для проверки корректности логической модели данных используется метод нормализации. Нормализация гарантирует, что выведенные из существующей модели данных отношения не будут обладать избыточностью данных, способной вызвать аномалии обновления после их физической реализации.

Построенная логическая модель данных является источником информации для этапа физического проектирования и обеспечивает разработчика физической базы данных средствами нахождения компромиссов, необходимых для достижения поставленных целей, что очень важно для эффективного проектирования. Логическая модель данных также играет важную роль на этапе эксплуатации и сопровождения уже готовой системы. При правильно организованном сопровождении поддерживаемая в актуальном состоянии модель данных позволяет точно и наглядно представить любые вносимые в базу данных изменения, а также оценить их влияние на прикладные программы и использование данных, уже имеющихся в базе.

Если рассматривать логическое проектирование более подробно, то можно выделить следующие основные его этапы:

1. Определение типов сущностей.
2. Определение типов связей.
3. Определение атрибутов и связывание их с типами сущностей и связей.
4. Определение доменов атрибутов.
5. Определение атрибутов, являющихся потенциальными и первичными ключами.

[1] В некоторой литературе можно встретить еще и концептуальное проектирование, которое, по сути, является предварительной стадией логического проектирования.

6. Специализация или генерализация типов сущностей (необязательный этап).
7. Создание диаграммы "сущность-связь".
8. Проверка модели с помощью правил нормализации.
9. Проверка модели в отношении транзакций пользователей.
10. Определение требований поддержки целостности данных.
11. Проверка возможностей расширения модели в будущем.
12. Обсуждение логической модели данных с пользователями.

Физическое проектирование базы данных

Под **физическим проектированием** базы данных понимается процесс создания описания реализации базы данных на вторичных запоминающих устройствах с указанием структур хранения и методов доступа, используемых для организации эффективной обработки данных.

На данной фазе построения базы данных проектировщик принимает решение о способах реализации разрабатываемой базы данных. Во время предыдущей фазы проектирования была определена логическая структура базы данных (т.е. набор ее сущностей, связей и атрибутов). Однако, приступая к физическому проектированию, прежде всего, необходимо выбрать конкретную целевую СУБД. Поэтому физическое проектирование неразрывно связано с конкретной СУБД. Между логическим и физическим проектированием существует постоянная обратная связь, так как решения, принимаемые на этапе физического проектирования с целью повышения производительности системы, способны повлиять на структуру логической модели данных.

Для выбора целевой СУБД необходимая информация о таких общих требованиях к системе, как производительность, простота реорганизации, уровень защищенности и ограничения целостности данных.

При более подробном рассмотрении стадии физического проектирования можно выделить следующие виды деятельности:

1. Выбор целевой СУБД.
2. Проектирование основных таблиц в среде целевой СУБД.
3. Реализация бизнес правил предприятия в среде целевой СУБД.
4. Анализ транзакций.
5. Выбор файловой структуры.
6. Определение вторичных индексов.
7. Анализ необходимости введения контролируемой избыточности данных.
8. Определение требований к дисковой памяти.
9. Разработка пользовательских представлений.
10. Определение прав доступа.
11. Организация мониторинга и настройка функционирования системы.

Разработка приложений

На рис. **33** показано, что в жизненном цикле системы проектирование базы данных и разработка приложений выполняется параллельно. В большинстве случаев проектирование приложений нельзя завершить до окончания проектирования базы данных. С другой стороны, база данных предназначена для поддержки приложений, а потому между фазами проектирования базы данных и проектирования приложений для этой базы данных должен постоянно происходить обмен информацией.

Необходимо убедиться, что все функциональные возможности, предусмотренные в спецификациях требований пользователей, обеспечиваются интерфейсом пользователя соответствующих приложений. Помимо проектирования способов, с помощью которых пользователь сможет получить доступ к необходимым ему функциональным возможностям, следует также разработать соответствующий пользовательский интерфейс приложения. Он должен предоставлять пользователю необходимую информацию самым удобным для него образом. При всей своей важности, проектирование интерфейсов пользователя порою просто игнорируется или же оставляется на самые поздние этапы разработки[1]. Однако следует признать, что пользовательские интерфейсы являются одним из важнейших компонентов системы. Если интерфейс легко осваивается персоналом, прост в использовании, интуитивно понятен и устойчив к ошибкам, то пользователи легко научатся извлекать пользу из представленной в нем информации. В то же время, если интерфейс лишен указанных качеств, то работа с такой системой неизбежно будет сопровождаться теми или иными проблемами.

Создание прототипов

На различных этапах процесса проектирования системы имеется возможность либо полной реализации приложения базы данных, либо создания его прототипа.

Прототип – это рабочая модель, которая обычно обладает лишь частью требуемых возможностей и не обеспечивает всей функциональности готовой системы.

Прототип приложения базы данных создается для того, чтобы дать пользователям возможность опробовать его в работе и определить, какие из функциональных средств системы отвечают своему назначению, а какие

[1] Это связано с тем, что в большинстве случаев проектированием интерфейса занимаются сами программисты, слабо знакомые с правилами его построения, либо не считающие эту деятельность творческим делом. Проектированием интерфейса должны заниматься дизайнеры, однако для небольших команд разработчиков иметь таких людей в своем штате – непозволительная роскошь.

нет. В последнем случае пользователям предлагается указать (если это возможно), какие улучшения или даже совершенно новые функции желательно реализовать в приложении. Таким образом, прототип представляет собой инструмент, позволяющий в значительной степени прояснить требования пользователей как для самих себя, так и для разработчиков системы, а также оценить гибкость разработанного проекта базы данных[1].

Реализация

В результате выполнения всех этапов проектирования (которые могут включать или не включать создание прототипов) будет подготовлено все, что необходимо для реализации базы данных и прикладных программ. Реализация базы данных осуществляется посредством создания ее описания на языке определения данных (DDL) в среде целевой СУБД. Команды DDL-языка компилируются и используются для создания схем и пустых файлов базы данных. На этом же этапе определяются и все специфические пользовательские представления[2].

Конвертирование и загрузка данных

Данный этап выполняется, как правило, в том случае, когда вводится в опытную эксплуатацию новая версия информационной системы. В настоящее время практически любая СУБД имеет утилиту загрузки уже существующих файлов в новую базу данных. Этой утилите обычно требуется предоставить спецификацию файла-источника и целевой базы данных, после чего она автоматически преобразует данные в нужный формат файлов новой базы данных. Если только это возможно, разработчику следует преобразовать все имеющиеся приложения старой системы для использования их в новой системе[3]. Всякий раз, когда требуется выполнить преобразование и загрузку данных, этот процесс следует тщательно планировать с целью обеспечения гладкого перехода системы в состояние полной готовности.

Тестирование

Прежде чем использовать новую систему на практике, ее следует тщательно протестировать. Этого можно добиться путем разработки продуманной стратегии тестирования с использованием реальных данных, кото-

[1] В случае, когда планируется создание новой версии информационной системы с новым интерфейсом пользователя, использование прототипа весьма приветствуется.

[2] Про представления более подробно рассказывается в соответствующем разделе ниже.

[3] К сожалению, воспользоваться средствами автоматического преобразования становится проблематичным, если структура базы данных новой версии системы серьезно отличается от структуры старой версии. В этом случае для заполнения базы данными придется разрабатывать дополнительные программы.

рая должна быть построена таким образом, чтобы весь процесс тестирования выполнялся строго последовательно и методически правильно[1].

Стратегии тестирования

Для оценки законченности и корректности выполнения приложения базы данных может использоваться несколько различных стратегий тестирования. К ним относятся:

- Нисходящее тестирование.
- Восходящее тестирование.
- Тестирование потоков.
- Интенсивное тестирование.

Нисходящее тестирование начинается на уровне подсистем с модулями, которые представлены заглушками, т.е. простыми компонентами, имеющими такой же интерфейс, как модуль, но без функционального кода. Каждый модуль низкого уровня представляется заглушкой. В конечном итоге все программные компоненты заменяются фактическим кодом и снова тестируются.

Восходящее тестирование выполняется в противоположном направлении по отношению к нисходящему. Оно начинается с тестирования модулей на самых низких уровнях иерархии системы, продолжается на более высоких и заканчивается на самом высоком уровне.

Тестирование потоков – это стратегия тестирования систем, работающих в реальном масштабе времени, которые обычно состоят из большого количества взаимодействующих процессов, управляемых с помощью прерываний. Эти системы довольно трудно тестируются, поскольку взаимодействие процессов системы зависит от времени. Стратегия тестирования потоков направлена на слежение за отдельными процессами.

Некоторые системы создаются с целью работы в конкретных режимах максимальной или минимальной нагрузки. Стратегия **интенсивного тестирования** предназначена для проверки возможности системы справляться с запланированной нагрузкой. Такое тестирование часто включает серию тестов с постепенно возрастающей нагрузкой и продолжается до тех пор, пока система не выйдет из строя. Эта стратегия обладает двумя основными преимуществами: она проверяет поведение системы, а также оказывает давление на систему, вызывая появление сбоев, которые не могли бы быть обнаружены в обычных условиях эксплуатации.

[1] При этом различают функциональное тестирование, т.е. соответствие системы требованиям заказчика, а также тестирование быстродействия, при котором анализируется поведение системы при различных уровнях нагрузки на аппаратную часть.

Эксплуатация и сопровождение

На предыдущих этапах приложение базы данных было полностью реализовано и протестировано. Теперь система входит в последний этап своего жизненного цикла, называемый эксплуатацией и сопровождением.
Он включает выполнение таких действий, как:

- ***контроль производительности системы***. Если производительность падает ниже приемлемого уровня, то может потребоваться дополнительная настройка или реорганизация базы данных;
- ***сопровождение и модернизация приложений баз данных***. Новые требования включаются в приложение базы данных при повторном выполнении предыдущих этапов жизненного цикла.

6. ОСНОВЫ ЯЗЫКА SQL

Одним из языков, появившихся в результате разработки реляционной модели данных, является язык Structured Query Language (SQL)[1], который в настоящее время получил очень широкое распространение и фактически превратился в стандартный язык реляционных баз данных. Стандарт на язык SQL был выпущен Американским национальным институтом стандартов (ANSI) в 1986 году, а в 1987 году Международная организация стандартов (ISO) приняла этот стандарт в качестве международного. В настоящее время язык SQL поддерживается многими десятками СУБД различных типов, разработанных для самых разнообразных вычислительных платформ, начиная от персональных компьютеров и заканчивая мейнфреймами[2]. В данном разделе приводится краткое описание основных возможностей языка SQL. Для более подробного изучения рекомендуется обратиться сначала к монографии [4]. Далее следует обратиться с специфической литературе по конкретной СУБД, в том числе и по языку SQL.

Язык SQL предоставляет пользователю следующие возможности:

- создавать базы данных и таблицы с полным описанием их структуры[3];
- выполнять основные операции манипулирования данными, такие как вставка, модификация и удаление данных из таблиц;
- выполнять простые и сложные запросы, осуществляющие преобразование необработанных данных в необходимую информацию[4].

История языка SQL

История реляционной модели данных (и косвенно языка SQL) началась в 1970 году с публикации основополагающей статьи Е. Ф. Кодда. В 1974 году Д. Чамберлин публикует определение языка, получившего название "Structured English Query Language", или SEQUEL. В 1976 году была выпущена переработанная версия-этого языка, SEQUEL/2. В том же году на базе языка SEQUEL/2 корпорация IBM выпустила прототип СУБД, имевший название "System R". Назначение этой пробной версии состояло в проверке осуществимости реляционной модели.

В последствии название SEQUEL пришлось изменить на SQL из юридических соображений – аббревиатура SEQUEL уже использовалась кем-

[1] Что в переводе означает «структурированный язык запросов».

[2] Несмотря на это нельзя сказать, что это «один и тот же» SQL – практически в каждой СУБД существуют некоторые нюансы, которые распространяются, в том числе и на диалект языка SQL.

[3] Ввиду того, что в различных СУБД присутствуют различные типы данных, следовательно, конструкции языка SQL в данном случае будут различными.

[4] Данная возможность языка SQL представляется наиболее важной.

то ранее. Однако до настоящего времени многие люди[1] по-прежнему произносят аббревиатуру SQL как "си-кьюэл", хотя официально ее рекомендуется читать как "эс-кью-эл".

В конце 70-х годов компанией, которая ныне превратилась в корпорацию Oracle, была выпущена СУБД Oracle. Пожалуй, это самая первая из коммерческих реализаций реляционной СУБД, построенной на использовании языка SQL. Чуть позже появилась СУБД INGRES, использовавшая язык запросов QUEL. Этот язык был более структурированным, чем SQL, но его семантика была менее близка обычному английскому языку. Позднее, когда язык SQL был принят как стандартный язык реляционных баз данных, СУБД INGRES была переведена на использование этого языка. В 1983 году корпорацией IBM была разработана СУБД DB2.

В 1982 году Американский национальный институт стандартов (ANSI) начал работу над языком RDL (Relation Database Language), руководствуясь концептуальными документами, полученными от корпорации IBM. В 1983 году к этой работе подключился Международный комитет по стандартизации (ISO). Совместные усилия обеих организации увенчались выпуском стандарта языка SQL. Исходный вариант стандарта, который был выпущен ISO в 1987 году, вызвал волну критических оценок.

В 1989 году ISO опубликовало дополнение к стандарту, в котором определялись функции поддержки целостности данных. В 1992 году была выпущена первая, существенно пересмотренная версия стандарта ISO, которую иногда называют SQL2 (или SQL-92). Хотя некоторые из функций были определены в этом стандарте впервые, многие из них уже были полностью или частично реализованы в одной или более коммерческих реализаций языка SQL.

Незадолго до завершения работ по определению стандарта SQL2 была начата разработка стандарта SQL3. Реально работу над новым стандартом удалось частично завершить только в 1999 г., и по этой причине (а также в связи с проблемой 2000 года) стандарт получил название SQL:1999. В новых стандартах вводится система типов языка, формулируются правила определения функциональных зависимостей и возможных ключей, определяются синтаксис и семантика основных операторов языка SQL:

- операторов определения и манипулирования схемой базы данных;
- операторов манипулирования данными;
- операторов управления транзакциями (расширенные модели транзакций, контрольные точки, многозвенные транзакций);
- операторов управления подключениями к базе данных и т. д.

В конце 2003 г. был принят и опубликован новый вариант международного стандарта SQL:2003.

[1] К ним относятся, прежде всего, разработчики баз данных.

Использование SQL для извлечения данных

Как уже упоминалось ранее SQL – структурированный язык запросов. Запросы – вероятно, наиболее часто используемый аспект SQL.

Запрос – это команда, которую вы даете вашей программе базы данных, и которая сообщает ей, чтобы она вывела определенную информацию из таблиц в память.

Запросы обычно рассматриваются как часть языка DML. Однако, так как запрос не меняет информацию в таблицах, а просто показывает ее пользователю, мы будем рассматривать запросы как самостоятельную категорию среди команд DML которые производят действие, а не просто показывают содержание базы данных.

Все запросы в SQL состоят из одиночной команды. В общем виде структура этой команды обманчиво проста Эта команда называется - SELECT (ВЫБОР). Синтаксис данной команды имеет вид:

```
SELECT [distinct|All]{*|[column_i [as name_i]][,...]}
FROM table_name [alias] [, ...]
[WHERE condition]
[GROUP BY group_column_list] [HAVING condition]
[ORDER BY order_column_list]
```

Ниже представлено краткое описание основных секций команды SQL (см. таблицу ***1***). Более подробно они описываются ниже при детальном рассмотрении каждой из них.

Таблица 1

Краткое описание основных секций команды SQL

Секция	Описание
FROM	Определяются имена используемой таблицы или нескольких таблиц
WHERE	Выполняется фильтрация строк объекта в соответствии с заданными условиями
GROUP BY	Образуются группы строк, имеющих одно и то же значение в указанном столбце
HAVING	Фильтруются группы строк объекта в соответствии с указанным условием
SELECT	Устанавливается, какие столбцы должны присутствовать в выходных данных
ORDER BY	Определяется упорядоченность результатов выполнения оператора

Выбор всех строк

Пусть, например, необходимо вывести все записи из таблицы «Студенты», представленной ранее на рисунке **13**. Для этого достаточно написать запрос:

```
SELECT * FROM Студенты
```
[1]

Предположим, необходимо вывести имена всех студентов. Можно записать запрос в виде:

```
SELECT Имя FROM Студенты
```

Однако в этом случае одинаковые имена различных студентов будут входить в результирующую выборку. Для того, чтобы результатом запроса были только неповторяющиеся имена, необходимо написать запрос в виде

```
SELECT DISTINCT Имя FROM Студенты
```

В секцию SELECT допускается включать также вычисляемые поля. Например, выведем список всех сумм по заказам из таблицы «Заказы_по_накладным», представленной на рисунке **32**:

```
SELECT Цена*Количество FROM Заказы_по_накладным.
```

Выбор строк (предложение WHERE)

В приведенных выше примерах в результате выполнения операторов SELECT выбирались все строки указанной таблицы. Однако в большинстве случаев требуется тем или иным образом ограничить набор строк, помещаемых в результирующую таблицу запроса. Это достигается с помощью указания в запросе предложения WHERE. Оно состоит из ключевого слова, за которым следует перечень условий поиска, определяющих те строки, которые должны быть выбраны при выполнении запроса. Существует пять основных типов условий поиска:

- ***Сравнение***. Сравниваются результаты вычисления одного выражения с результатами вычисления другого выражения.
- ***Диапазон***. Проверяется, попадает ли результат вычисления выражения в заданный диапазон значений.

[1] Русскоязычные названия таблиц поддерживаются очень немногими СУБД. В данном случае они используются лишь для большей информативности.

- ***Принадлежность к множеству***. Проверяется, принадлежит ли результат вычисления выражения заданному множеству значений.
- ***Соответствие шаблону***. Проверяется, отвечает ли некоторое строковое значение заданному шаблону.
- ***Значение NULL***. Проверяется, содержит ли данный столбец определитель NULL (пустое значение).

Различные условия на выборку данных могут объединяться друг с другом с помощью логических операторов NOT, AND и OR[1]. С помощью скобок можно изменить порядок выполнения условий в запросе. Вычисление выражений в условиях выполняется по следующим правилам:

- Выражение вычисляется слева направо.
- Первыми вычисляются подвыражения в скобках.
- Операторы NOT выполняются до выполнения операторов AND и OR.
- Операторы AND выполняются до выполнения операторов OR.

Для устранения любой возможной неоднозначности рекомендуется использовать скобки. Рассмотрим примеры использования всех указанных типов условий поиска.

Условие типа «Сравнение»

В языке SQL используются следующие операторы сравнения:

=	равенство
<	меньше
>	больше
<=	меньше или равно
>=	больше или равно
<>	не равно (стандарт ISO)
!=	не равно (используется в некоторых диалектах)

Например, если необходимо вывести все товары из прайс-листа с закупочной ценой меньшей 10000 единиц, которой имеется на складе в количестве, большем 100 штук (см. рис. **32**), то потребуется выполнить следующий запрос:

```
SELECT * FROM Прайс-лист
WHERE Закупочная_цена<10000 AND Количество>100
```

[1] Логические операторы приведены в порядке убывания их приоритета. Кроме этого приоритет самих логических операторов ниже, чем приоритет какого-либо условия поиска, поэтому нет необходимости различные логические условия заключать в скобки.

Условие типа «Диапазон»

Данный тип условия реализуется с помощью условий (BETWEEN/NOT BETWEEN). Например, если мы хотим вывести из прайс-листа (см. рис. **32**) все товары с кредитной ценой от 10000 до 20000, то можно воспользоваться следующим запросом:

```
SELECT * FROM Прайс-лист
WHERE Цена_в_кредит BETWEEN 10000 AND 20000
```

Наличие ключевого слова BETWEEN и соответствующей проверки лишь незначительно повышает выразительную мощность языка SQL, поскольку те же самые результаты могут быть достигнуты с помощью выполнения двух обычных проверок. Так, приведенный выше запрос можно переписать следующим образом:

```
SELECT * FROM Прайс-лист
WHERE Цена_в_кредит>=10000 AND
      Цена_в_кредит<=20000
```

Однако проверка вхождения в диапазон с помощью ключевого слова BETWEEN является более простым способом записи условий выборки, чем обычные проверки.

Условие типа «Принадлежность к множеству»

Данный тип условия реализуется с помощью условий (IN/NOT IN). Например, если мы хотим вывести всех студентов (см. рис. **13**) с именами Иван и Петр, то можно воспользоваться следующим запросом:

```
SELECT * FROM Студенты
WHERE Имя IN ('Иван', 'Петр')
```

Как и в случае ключевого слова BETWEEN, условие IN незначительно повышает выразительную мощность языка SQL, так как тот же самый запрос может быть переписан следующим образом:

```
SELECT * FROM Студенты
WHERE Имя='Иван' OR Имя='Петр'
```

Однако использование ключевого слова IN представляет собой более эффективный способ записи условий поиска, особенно если набор допустимых значений достаточно велик.

Условие типа «Соответствие шаблону»

Данный тип условия реализуется с помощью условий (LIKE/NOT LIKE). Он используется в случае, если невозможно точно указать значение поиска полностью, но можно задать какие-либо его части. В языке SQL существует два специальных символа шаблона, используемых при проверке значений:

% Символ процента представляет собой произвольную последовательность из нуля и более символов

_ Символ подчеркивания представляет любой одиночный символ

Так, например, если мы хотим вывести всех студентов, фамилия которых начинается на ‘Ива’, вторая буква имени – буква ‘А’, а отчество оканчивается на ‘ИЧ’, то можно воспользоваться следующим запросом:

```
SELECT * FROM Студенты
WHERE Фамилия  LIKE 'Ива%' AND
      Имя      LIKE '_А%'  AND
      Отчество LIKE '%ИЧ'
```

Условие типа «Значение NULL»

С помощью условия данного типа реализуется проверка, содержит ли какой-либо атрибут пустые значения. Например, если мы хотим вывести всех студентов с незаполненным именем, необходим следующий запрос:

SELECT * FROM Студенты **WHERE** Имя **IS NULL**

Казалось бы, тот же самый результат может быть получен с помощью запроса

SELECT * FROM Студенты Имя=’’

Однако это не так. Отсутствующие имена (значение NULL) рассматриваются как неизвестные значения, поэтому их нельзя проверить на равенство или неравенство с любой другой строкой. Если попробовать выполнить последний запрос, то результирующая таблица окажется пустой[1].

[1] Здесь имеется ввиду тот факт, что NULL<>’’, значение NULL не равно никакому другому значению, в том числе другому значению NULL.

Сортировка результатов (предложение ORDER BY)

В общем случае строки в результирующей таблице SQL-запроса не упорядочены каким-либо определенным образом[1]. Однако их можно требуемым образом отсортировать, для чего в оператор SELECT помещается фраза ORDER BY. Она включает список идентификаторов столбцов, разделенных запятыми, по которым требуется упорядочить результирующую таблицу запроса. Идентификатор столбца может представлять собой либо его имя, либо номер, который идентифицирует элемент списка SELECT его позицией в этом списке. Самый левый элемент списка имеет номер 1, следующий - номер 2 и т.д.[2]

Так, если требуется упорядочить список студентов по фамилии, то можно использовать следующий SQL:

```
SELECT Фамилия,Имя,Отчество FROM Студенты
ORDER BY Фамилия
```

или

```
SELECT Фамилия,Имя,Отчество FROM Студенты
ORDER BY 1
```

Фраза ORDER BY позволяет упорядочить выбранные записи в порядке возрастания (ASC) или убывания (DESC) значений любого столбца или комбинации столбцов, независимо от того, присутствуют эти столбцы в таблице результатов или нет. Фраза ORDER BY всегда должна быть последним элементом в операторе SELECT.

Обобщающие функции

В стандарте языка SQL имеется пять обобщающих функций:

COUNT	Возвращает количество значений в указанном столбце
SUM	Возвращает сумму значений в указанном столбце
AVG	Возвращает усредненное значение в указанном столбце
MIN	Возвращает минимальное значение в указанном столбце
MAX	Возвращает максимальное значение в указанном столбце

Все эти функции оперируют со значениями в единственном столбце таблицы и возвращают единственное значение. Функции COUNT, MIN и

[1] Это значит, что порядок следования записей в результате запроса является неопределенным.

[2] Номера столбцов являются функцией, отвергнутой стандартом ISO, кроме этого жесткая завязка на номер столбца является источником потенциальной ошибки при частой правке SQL-запросов. По этим причинам использование номеров столбцов при сортировке не рекомендуется.

MAX применимы как к числовым, так и к нечисловым полям, тогда как функции SUM и AVG могут использоваться только в случае числовых полей. За исключением COUNT(*), при вычислении результатов любых функции сначала исключаются все пустые значения, после чего требуемая операция применяется только к оставшимся конкретным значениям столбца.

Обобщающие функции могут использоваться только списке предложения SELECT и в составе предложения HAVING. Во всех других случаях использование этих функций недопустимо.

Например, если необходимо определить количество студентов с именем Иван, то можно воспользоваться следующим запросом:

```
SELECT count(*) FROM Студенты where имя='Иван'
```

Группировка результатов (предложение GROUP BY)

Приведенный выше пример сводных данных по студентам подобен итоговым строкам, обычно размещаемым в конце отчетов. В них все детальные данные отчета сжимаются в одну обобщающую строку. Однако очень часто в отчетах требуется формировать и промежуточные итоги. Для этой цели в операторе SELECT может указываться фраза GROUP BY. Запрос, в котором присутствует фраза GROUP BY, называется **группирующим запросом**, поскольку в нем группируются данные, полученные в результате выполнения операции SELECT, после чего для каждой отдельной группы создается единственная суммарная строка. Столбцы, перечисленные во фразе GROUP BY, называются **группируемыми столбцами**. Стандарт ISO требует, чтобы предложение SELECT и фраза GROUP BY были тесно связаны между собой. При использовании в операторе SELECT фразы GROUP BY каждый элемент списка в предложении SELECT должен иметь **единственное значение для всей группы**. Более того, предложение SELECT может включать только следующие типы элементов:

- имена столбцов;
- обобщающие функции;
- константы;
- выражения из комбинации перечисленных выше элементов.

Все имена столбцов, приведенные в списке предложения SELECT, должны присутствовать и во фразе GROUP BY - за исключением случаев, когда имя столбца используется в обобщающей функции. Обратное правило не является справедливым – во фразе GROUP BY могут присутствовать имена столбцов, отсутствующие в списке предложения SELECT. Если совместно с фразой GROUP BY используется предложение WHERE, то оно обрабатывается первым, а группировке подвергаются только те строки, которые удовлетворяют условию поиска.

Стандартом ISO определено, что при проведении группировки все отсутствующие значения рассматриваются как равные. Если две строки таблицы в одном и том же группируемом столбце содержат значения NULL, а значения во всех остальных непустых группируемых столбцах идентичны, то они помещаются в одну и ту же группу.

Если необходимо вывести список всех номеров накладных с указанием суммы заказа из таблицы «Заказы_по_накладным», изображенной на рисунке **32**, то для этого можно использовать следующий запрос:

```
SELECT Код_накладной, sum(Цена*Количество)
FROM Заказы_по_накладным
GROUP BY Код_накладной
ORDER BY Код_накладной
```

Ограничения на выполнение группирования (предложение HAVING)

Если в предложении GROUP BY необходимо задать какие-либо дополнительные ограничения на отбор тех групп, которые будут помещены в результирующую таблицу запроса, то для этого необходимо использовать внутри ORDER BY предложение HAVING. Хотя данная фраза и предложение WHERE имеют сходный синтаксис, их назначение различно. Предложение WHERE предназначено для фильтрации отдельных строк, тогда как фраза HAVING используется для фильтрации групп, помещаемых в результирующую таблицу запроса. Стандарт ISO требует, чтобы имена столбцов, используемые во фразе HAVING, обязательно присутствовали в списке фразы GROUP BY или применялись в обобщающих функциях. На практике условия поиска во фразе HAVING всегда включают, по меньшей мере, одну обобщающую функцию, в противном случае эти условия поиска должны быть помещены в предложение WHERE и применяться для отбора отдельных строк[1].

Фраза HAVING не является необходимой частью языка SQL - любой запрос, написанный с использованием фразы HAVING, может быть представлен в ином виде, без ее применения.

Если в предыдущем случае необходимо вывести список только тех накладных, в заказ которых входило не менее 10 наименований продукции, необходимо предыдущий запрос немного изменить:

```
SELECT Код_накладной, sum(Цена*Количество)
FROM Заказы_по_накладным
GROUP BY Код_накладной
HAVING count(*)>10
ORDER BY Код_накладной
```

[1] Не забывайте, что обобщающие функции не могут использоваться в предложении WHERE.

Подзапросы

Подзапросами называются законченные операторы SELECT, внедренные в тело другого оператора SELECT. Внешний оператор SELECT использует результат выполнения внутреннего оператора для определения содержания окончательного результата всей операции. Внутренние запросы могут быть помещены в предложения WHERE и HAVING внешнего оператора SELECT – в этом они получают название подзапросов, или вложенных запросов. Кроме того, внутренние операторы SELECT могут использоваться в операторах INSERT, UPDATE и DELETE[1]. Существует три типа подзапросов.

- ***Скалярный подзапрос*** возвращает единственное значение, выбираемое из пересечения одного столбца с одной строкой.
- ***Строковый подзапрос*** возвращает значения нескольких столбцов таблицы, но в виде единственной строки. Строковый подзапрос может использоваться везде, где применяется конструктор строковых значений – обычно это предикаты.
- ***Табличный подзапрос*** возвращает значения одного или больше столбцов таблицы, размещенные в более чем одной строке. Табличный подзапрос может использоваться везде, где допускается указывать таблицу – например, как операнд предиката IN.

Примеры подзапросов

Пусть имеются сущности «Клиенты» и «Города», представленные на рисунке **34**.

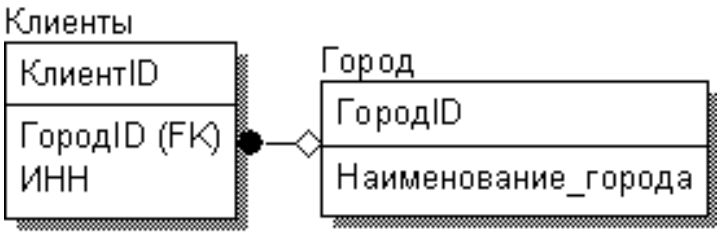

Рис. 34. Сущности «Клиенты» и «Адреса»

Если необходимо вывести всех клиентов, проживающих в Самаре, то можно использовать следующий запрос[2]:

[1] Данные SQL-операторы предназначены для модификации информации в базе данных.
[2] Данный запрос нельзя назвать удачным примером решения проблемы. Впоследствии будут приведены еще два варианта для получения той же самой информации.

```
SELECT * FROM Клиенты
WHERE ГородID=
   (SELECT ГородID FROM Город
    WHERE Наименование_города='Самара')
```

В предыдущем примере внутренний подзапрос не может выдавать более одной записи. В случае, например, если необходимо вывести всех клиентов, проживающих в городах, начинающихся на букву “С’, можно использовать следующий запрос:

```
SELECT * FROM Клиенты
WHERE ГородID IN
   (SELECT ГородID FROM Город
    WHERE Наименование_города LIKE 'С%')
```

В подзапросах можно использовать обобщающие функции. Предположим, необходимо вывести список всех товаров из прайс-листа (см. рис. **32**), закупочная цена которых меньше средней цены всех товаров. Для этого можно использовать следующий запрос:

```
SELECT * FROM Прайс_лист
WHERE Закупочная_Цена<
   (SELECT AVG(Закупочная_Цена) FROM Прайс_лист)
```

Многотабличные запросы

Во всех рассмотренных выше примерах помещаемые в результирующую таблицу столбцы всегда выбираются из единственной таблицы. Для того чтобы объединить в результирующей таблице столбцы из нескольких исходных таблиц, необходимо выполнить операцию **соединения**.

В языке SQL операция соединения используется для объединения информации из двух таблиц посредством образования пар связанных строк, выбранных из каждой таблицы.

Пусть, например, необходимо вывести список ИНН клиентов с указанием города его проживания (см. рис. **34**). В этом случае запрос будет выглядеть следующим образом:

```
SELECT Клиенты.ИНН, Город.Наименование_города
FROM Клиенты, Город
WHERE Клиенты.ГородID=Город.ГородID
```

Объединение таблицы с самой собой

Предположим, у нас имеется какая-либо иерархическая структура, например, таблица подразделений, каждое из которых может подчиняться какому-либо головному подразделению, а также иметь ряд дочерних. Самый простой способ задания иерархических структур – ввести в таблицу поле, ссылающееся на головную запись (см. рис. **35**).

Рис. 35. Задание иерархических структур

Предположим, данная таблица имеет наполнение, представленное на рисунке **36**.

ПодразделенияID	Наименование	ParentID
1	Самарский государственный университет	NULL
2	СамГТУ	NULL
3	Мех-мат	1
4	Социологический факультет	1
5	Юридический факультет	1
6	Кафедра БИС	3
7	Кафедра МММ	3
8	Инженерно-экономический факультет	2

Рис. 36. Наполнение таблицы «Подразделения»

В случае, если необходимо вывести список всех подразделений с указанием головного, потребуется следующий запрос:

```
SELECT D.Наименование, P.Наименование
FROM Подразделения D, Подразделения P
WHERE D.ParentID=P.ПодразделенияID
```

Следует обратить внимание на тот факт, что в результат данного запроса не попадают записи, у которых значение атрибута ParentID равно пустому значению[1]. В случае, если данные записи необходимо выводить в запросе, необходимо познакомиться с различными типами открытых соединений таблиц.

[1] Т.к. значение NULL не равен никакому другому значению, в том числе значению NULL.

Типы открытых соединений таблиц

Соединение является подмножеством более общей комбинации данных двух лиц, называемой их декартовым произведением. Пример декартова произведения приводился ранее на рисунке **27**.

Стандарт ISO предусматривает специальный формат оператора SELECT, позволяющий вычислить декартово произведение двух таблиц:

```
SELECT [DISTINCT | ALL] {*|column_list}
FROM table_name1 CROSS JOIN table_name2
```

Для получения декартова произведения таблицы клиентов и таблицы городов[1] (см. рис. **34**), можно использовать следующий запрос:

```
SELECT * FROM Клиенты CROSS JOIN Город
```

Однако то же самое можно получить более привычным запросом:

```
SELECT * FROM Клиенты, Город
```[2]

При выполнении операции соединения данные из двух таблиц комбинируются с образованием пар связанных строк, в которых значения сопоставляемых столбцов одинаковые. Если строка одной из таблиц не находит себе соответствия в другой, то она не попадает в результирующий набор данных. Именно это правило применялось во всех рассмотренных выше примерах соединения таблиц. Стандартом ISO предусмотрен и другой набор операторов соединений, называемых **открытыми соединениями**. При открытом соединении в результирующую таблицу помещаются также строки, не удовлетворяющие условию соединения. Чтобы понять особенности выполнения операций открытых соединений, воспользуемся таблицами «Отделения» и «Аренда» (см. рис. **37**).

№ отдела	Город
Отдел 1	Самара
Отдел 2	Тольятти
Отдел 3	Москва

№ объекта	Город
A15	Казань
A16	Москва
A17	Самара

Рис. 37. Отношения «Отделения» и «Аренда»

Обычное (закрытое) соединение этих таблиц выполняется с помощью следующего SQL-оператора:

---

[1] Смысла, правда, в таком запросе нет абсолютно.

[2] Как правило, появление декартова произведения является следствием ошибки в запросе, когда пропущено какое-либо условие связи между таблицами.

```
SELECT Отделения.*, Аренда.№_объекта
FROM Отделения, Аренда
WHERE Отделения.Город=Аренда.Город
```

Результат выполнения запроса приведен ниже (см. рис. **38**)

№ отдела	Город	№ объекта
Отдел 1	Самара	A17
Отдел 3	Москва	A16

Рис. 38. Результат выполнения закрытого соединения

Как можно видеть, в результирующей таблице запроса имеются только две строки, содержащие одинаковые названия городов, выбранные из обеих таблиц. Если в результирующую таблицу требуется включить строки, не имеющие соответствия по городам, следует использовать открытое соединение. Существует три типа открытых соединений: **левое**, **правое** и **полное открытое**. Рассмотрим особенности каждого из них на приведенных ниже примерах.

Так, если необходимо вывести все отделения компании и сдаваемые в аренду объекты, которые расположены в одном и том же городе, а также прочие отделения компании, то необходимо использовать **левое открытое соединение**, которое на стандартном языке SQL записывается следующим образом[1]:

```
SELECT Отделения.*, Аренда.№_объекта
FROM Отделения
LEFT JOIN Аренда ON Отделения.Город=Аренда.Город
```

Результат запроса представлен на рисунке **39**.

№ отдела	Город	№ объекта
Отдел 1	Самара	A17
Отдел 2	Тольятти	NULL
Отдел 3	Москва	A16

Рис. 39. Результат выполнения левого открытого соединения

Если же необходимо вывести все отделения компании и сдаваемые в аренду объекты, которые расположены в одном и том же городе, а также

[1] В СУБД Oracle данный запрос можно написать более простым образом:
SELECT Отделения.*, Аренда.№_объекта FROM Отделения, Аренда
WHERE Отделения.Город=Аренда.Город(+)

прочие объекты недвижимости, потребуется использовать **правое открытое соединение**. Для этого в предыдущем запросе необходимо заменить LEFT на RIGHT, а все остальное оставить без изменения. Результат запроса представлен на рисунке **40**.

№ отдела	Город	№ объекта
NULL	Казань	A15
Отдел 3	Москва	A16
Отдел 1	Самара	A17

Рис. 40. Результат выполнения правого открытого соединения

Если же необходимо вывести все отделения компании и сдаваемые в аренду объекты, которые расположены в одном и том же городе, а также прочие объекты недвижимости и отделы компании, то потребуется использовать **полностью открытое соединение**. Для этого в последнем приведенном запросе необходимо заменить LEFT на FULL, а все остальное оставить без изменения. Результат запроса представлены на рисунке **41**.

№ отдела	Город	№ объекта
Отдел 1	Самара	A17
Отдел 2	Тольятти	NULL
Отдел 3	Москва	A16
NULL	Казань	A15

Рис. 41. Результат выполнения полностью открытого соединения

Возвратимся к иерархической структуре сущности «Подразделения», представленной на рисунке 36. Если необходимо, чтобы в запрос попали записи, у которых значение атрибута ParentID равно пустому значению (NULL), то потребуется использовать в запросе левое открытое соединение:

```
SELECT D.Наименование, P.Наименование
FROM Подразделения D
LEFT JOIN Подразделения P ON
          D.ParentID=P.ПодразделенияID
```

Ранее для сущностей «Клиенты» и «Адреса», представленных на рисунке **34** составлялся запрос, который выводил список всех клиентов, проживающих в Самаре. Для этого использовался скалярный подзапрос. Тот

же самый результат можно получить с помощью объединения данных из нескольких таблиц[1]:

```
SELECT Клиенты.* FROM Клиенты, Город
WHERE Клиенты.ГородID=Город.ГородID AND
      Город.Наименование_города='Самара'
```

Ключевые слова EXISTS и NOT EXISTS

Ключевые слова EXISTS и NOT EXISTS предназначены для использования только совместно с подзапросами. Результат их обработки представляет собой логическое значение TRUE или FALSE. Для ключевого слова EXISTS результат равен TRUE в том и только в том случае, если в возвращаемой подзапросом результирующей таблице присутствует хотя бы одна строка. В противном случае результатом обработки ключевого слова EXISTS будет значение FALSE. Для ключевого слова NOT EXISTS используются правила обработки, обратные по отношению к ключевому слову EXISTS.

Возвратимся к примеру, когда необходимо получить список всех клиентов, проживающих в Самаре. Те же самые данные можно получить также и с помощью запроса, содержащего внутри себя подзапрос с ключевым словом EXISTS:

```
SELECT Клиенты.*
FROM Клиенты
WHERE EXISTS(
  SELECT NULL FROM Город
  WHERE Город.ГородID=Клиенты.ГородID AND
        Город.Наименование_города='Самара')
```

Комбинирование результирующих таблиц

В языке SQL можно использовать обычные операции над множествами – объединение, пересечение и разность, позволяющие комбинировать результаты выполнения двух и более запросов в единую результирующую таблицу.

На таблицы, которые могут комбинироваться с помощью операций над множествами, накладываются определенные ограничения. Самое важное из них состоит в том, что таблицы должны иметь одну и ту же структуру. Это означает, что таблицы должны иметь одинаковое количество

[1] Таким образом, нетрудно видеть, что одни и те же данные можно получить с помощью различных запросов. Далее будет приведен еще один пример запроса той же самой информации.

столбцов, причем в соответствующих столбцах должны размещаться данные одного и того же типа и длины. Обязанность убедиться в том, что значения данных соответствующих столбцов принадлежат одному и тому же домену, возлагается на пользователя.

Три операции над множествами, предусмотренные стандартом ISO, носят название UNION, INTERSECT и EXCEPT. В каждом случае формат предложения с операцией над множествами должен быть следующим:

```
Operator [ALL] [CORRESPONDING [BY {column1 [,...]}]]
```

При указании фразы CORRESPONDING BY операция над множествами выполняется для указанных столбцов. Если задано только ключевое слово CORRESPONDING, а фраза BY отсутствует, операция над множествами выполняется для столбцов, которые являются общими для обеих таблиц. Если указано ключевое слово ALL, результирующая таблица может содержать дублирующиеся строки.

Одни диалекты языка SQL не поддерживают операций INTERSECT и EXCEPT, а в других вместо ключевого слова EXCEPT используется ключевое слово MINUS.

Рассмотрим сущности «Физические лица», «Юридические лица» и «Учредители» (см. рис. **42**).

Рис. 42. Сущности «Физические лица», «Юридические лица» и «Учредители»

У какого-либо юридического лица могут быть учредители (атрибут OrganizationID в таблице «Учредители»), которые, в свою очередь, могут быть либо физическими лицами (атрибут FPeopleID), либо юридическими лицами (атрибут FOrganizationID). Для любой записи из таблицы «Учредители» может быть заполнено только значение одного из атрибутов – либо FOrganizationID, либо FPeopleID, значение оставшегося атрибута должно быть выставлено в NULL[1].

Предположим, что необходимо для какого-либо юридического лица определить список его учредителей. В этом случае потребуется осуществлять объединение двух запросов:

[1] Т.к. учредитель не может быть одновременно и физическим, и юридическим лицом.

```
SELECT ФИО AS NAME, ИНН
FROM Учредители, Физические_лица FL
WHERE Учредители.OrganizationID=123456
AND Учредители.FPeopleID=FL.PeopleID
  UNION
SELECT Наименование AS NAME, ИНН
FROM Учредители, Юридические_лица UL
WHERE Учредители.OrganizationID=123456
AND Учредители.FOrganizationID = UL.OrganizationID
ORDER BY ИНН
```

Изменение содержимого базы данных

Язык SQL является полнофункциональным языком манипулирования данными, который может использоваться не только для выборки из базы, но и для модификации ее содержимого. Операторы модификации информации в базе данных не столь сложны, как оператор SELECT. Для модификации содержимого базы данных в языке SQL предусмотрены три оператора:

- **INSERT** - предназначен для добавления данных в таблицу.
- **UPDATE** - предназначен для модификации уже помещенных в таблицу данных.
- **DELETE** - позволяет удалять из таблицы строки данных.

Добавление новых данных в таблицу (оператор INSERT)

Существует две формы оператора INSERT. Первая предназначена для вставки единственной строки в указанную таблицу. Эта форма оператора имеет следующий формат:

```
INSERT INTO table name [(column_list)]
VALUES (data_value_list)
```

Здесь параметр column_list представляет собой список, состоящий из имен одного или более столбцов, разделенных запятыми. Данный параметр является необязательным. Если он опущен, то предполагается использование списка из имен всех столбцов таблицы. Параметр data_value_list (список значений данных) должен соответствовать параметру colum_list следующим образом:

- количество элементов в обоих списках должно быть одинаковым;
- должно существовать прямое соответствие между позицией одного и того же элемента в обоих списках,
- типы данных элементов списка data_value_list должны быть совместимы с типом данных соответствующих столбцов таблицы.

Так, например, для того, чтобы вставить в таблицу физических лиц (см. рис. **42**) какую-либо запись, то можно использовать следующую конструкцию языка SQL:

```
INSERT INTO Физические_лица(PeopleID,ФИО,ИНН)
VALUES ('1', 'Иванов И.И.',NULL)
```

Вторая форма оператора INSERT позволяет скопировать множество строк одной таблицы в другую таблицу. Этот оператор имеет следующий формат:

```
INSERT INTO table_name [(column_list)]
SELECT ...
```

Здесь параметры table_name и column_list имеют тот же формат и смысл, что и при вставке в таблицу одной строки. Предложение SELECT может представлять собой любой допустимый запрос. Строки, вставляемые в указанную таблицу, в точности соответствуют строкам результирующей таблицы, созданной при выполнении вложенного запроса. Все ограничения, указанные выше для первой формы оператора INSERT, применимы и в этом случае.

Модификация данных в базе (оператор UPDATE)

Оператор UPDATE позволяет изменять содержимое уже существующих строк указанной таблицы. Этот оператор имеет следующий формат:

```
UPDATE table_name
SET column_name1 = data_value1
[, column_name2=data_value2 ...]
[WHERE search_condition]
```

В предложении SET необходимо указывать имена одного или более столбцов, данные в которых необходимо изменить. Предложение WHERE является необязательным. Если оно опущено, значения указанных столбцов будут изменены во всех строках таблицы. Если предложение WHERE присутствует, обновлены будут только те строки, которые удовлетворяют условию поиска, заданному в параметре search_condition. Параметры data_value представляют новые значения соответствующих столбцов и должны быть совместимы с ними по типу данных.

Вспоминая деноминацию 1998 года, можно написать оператор, который уменьшает всем единицам продукции из прайс-листа (см. рис. **32**) закупочную, кредитную и наличную цену в 1000 раз[1]:

[1] К сожалению, в период деноминации не все осуществлялось по такой нехитрой формуле.

```
UPDATE Прайс-лист
SET Закупочная_цена = Закупочная_цена/1000,
    Цена_в_кредит = Цена_в_кредит/1000,
    Цена_наличными = Цена_наличными/1000
```

Удаление данных из базы (оператор DELETE)

Оператор DELETE позволяет удалять строки данных из указанной таблицы, данный оператор имеет следующий формат:

```
DELETE FROM table_name [WHERE search_condition]
```

Параметр search_condition является необязательным - если он опущен, из таблицы будут удалены все существующие в ней строки. Однако сама по себе таблица удалена не будет. Если предложение WHERE присутствует, то из таблицы будут удалены только те строки, которые удовлетворяют условию отбора, заданному параметром search_condition.

Например, если из прайс-листа необходимо удалить все записи о товарах, которых не осталось на складе, то потребуется следующий оператор:

```
DELETE FROM Прайс-лист WHERE Количество=0
```

Определение данных

Язык DDL (Data Definition Language), будучи частью языка SQL, позволяет создавать и уничтожать различные объекты базы данных, например, таблицы, представления или индексы. В данном разделе кратко рассматривается, что такое представление и индексы, а также каким образом они создаются и удаляются.

Основными операторами языка SQL, предназначенными для определения данных, являются следующие:

CREATE TABLE	ALTER TABLE	DROP TABLE
CREATE VIEW		DROP VIEW
CREATE INDEX		DROP INDEX

Создание/модификация/удаление таблиц

Таблицы создаются командой **CREATE TABLE**. Эта команда создает пустую таблицу – таблицу без строк. В команде CREATE TABLE задается имя таблицы, а также набор ее столбцов, указанных в определенном порядке. Она также определяет типы данных и размеры столбцов. Каждая

таблица должна иметь, по крайней мере, один столбец. Синтаксис команды CREATE TABLE имеет следующий вид:

```
CREATE TABLE <table-name >
( <column name > <data type>[(<size>)][,
<column name > <data type> [(<size>)] ... ]);
```

Типы данных у различных СУБД отличаются друг от друга[1]. Так как пробелы используются для разделения частей команды SQL, они не могут быть частью имени таблицы (или любого другого объекта, такого как индекс). Символ подчеркивания «_» обычно используется для разделения слов в именах таблиц.

Модификация структуры таблицы осуществляется с помощью команды **ALTER TABLE**. Синтаксис данной команды для различных СУБД сильно отличается. Например, если в СУБД Oracle необходимо расширить размер строкового поля, можно воспользоваться следующим оператором:

```
ALTER TABLE PBOSS MODIFY TREATMENTROD VARCHAR2(200);
```

Для удаления таблицы из схемы базы данных, необходимо воспользоваться командой **DROP TABLE**.

Представления

Представление – тип таблицы, чье содержимое выбирается из других таблиц с помощью выполнения SQL-запроса.

Данные представлений можно использовать в запросах абсолютно также как и данные таблиц, для пользователя не будет никаких внешних различий. Представления подобны окнам, которые из всей информации, находящейся в базе данных выводят только то, что какой-либо пользователь имеет желание или полномочие видеть. Таким образом, с помощью представлений можно построить хотя и простую, но при определенных условиях, достаточно мощную защиту от несанкционированного доступа к базе данных. Способам защиты информации в базах данных будет посвящен целый раздел ниже.

Для создания представлений необходимо использовать специальную команду. Она состоит из слов CREATE VIEW (СОЗДАТЬ

[1] Например, для СУБД Oracle таблицу можно создать с помощью следующего оператора:
CREATE TABLE PACTUAL (ACTUALID NUMBER(10) NOT NULL,
NAME VARCHAR2(40) NOT NULL
PRIMARY KEY (ACTUALID));

ПРЕДСТАВЛЕНИЕ), имени представления, которое нужно создать, слова AS (КАК), и далее запроса, как в следующем примере:

```
CREATE VIEW GROUPS AS
SELECT GROUPSID,NAME,COMMENTS
FROM PGROUPS WHERE DP=1;
```

Содержание представления не фиксировано, и переназначается каждый раз, когда выполняется запрос, использующий данные представления. Если вставить в таблицу PGROUPS новую запись со значением атрибута DP равным единице, то она автоматически появится в представлении GROUPS.

С помощью некоторых представлений можно модифицировать данные. Также представления называются **модифицируемыми**. Т.е. имена представлений можно подставлять в DML-операторы, такие как INSERT, UPDATE или DELETE, вместо имен таблиц. Однако для этого представления должны удовлетворять достаточно большому количеству ограничений.

Критерии модифицируемости представления:

- Оно должно выводиться в одну и только в одну базовую таблицу.
- Оно должно содержать первичный ключ этой таблицы[1].
- Оно не должно иметь никаких полей, которые бы являлись агрегатными функциями.
- Оно не должно содержать DISTINCT, GROUP BY или HAVING в своем определении.
- Оно не должно использовать подзапросы (это - ANSI ограничение, которое не предписано для некоторых реализаций языка SQL).
- Оно не должно использовать константы, строки, или выражения значений среди выбранных полей вывода.
- Для команды INSERT, необходимо, чтобы представление содержало все поля основной таблицы, которые имеют ограничение NOT NULL.

Для удаления представления из схемы базы данных, необходимо воспользоваться командой **DROP VIEW**.

Индексы

Индекс представляет собой структуру, позволяющую выполнять ускоренный доступ к строкам таблицы на основе значений одного или более ее столбцов. Наличие индекса может существенно повысить скорость выполнения некоторых запросов. Индексы обычно создаются с целью удов-

[1] В СУБД Oracle данное требование дополняется необходимостью, чтобы кроме первичного ключа в представление был включен уникальный идентификатор строки таблицы (ROWID), который генерируется автоматически самой СУБД.

летворения определенных критериев поиска, после того как таблица уже находилась некоторое время в работе и увеличилась в размерах. Создание индексов не предусмотрено стандартом на язык SQL[1]. Однако большинство диалектов поддерживает как минимум, следующий оператор:

```
CREATE [UNIQUE] INDEX index_name
ON table_name (column [ASC | DESC][, ...])
```

Указанные в операторе столбцы составляют ключ индекса и должны быть перечислены в возрастающем или убывающем порядке. Индексы могут создаваться только для таблиц базы данных, но не для представлений. Если в операторе указано ключевое слово UNIQUE, уникальность значений ключа индекса будет автоматически поддерживаться системой. Требование уникальности значений обязательно, для первичных ключей, а также, возможно, и для других столбцов таблицы (например, для ключей-кандидатов). Хотя создание индексов возможно в любой момент, при построении индекса для уже заполненной данными таблицы могут возникнуть проблемы, связанные с дублированием данных в различных строках. Следовательно, имеет смысл создавать уникальные индексы (по крайней мере, для первичного ключа) непосредственно при создании таблицы. В результате система сразу же возьмет на себя контроль над уникальностью значений данных в соответствующих столбцах.

За счет чего же ускоряется доступ к данным при использовании индексов? На самом деле никаких чудес в индексах нет. В данном случае используется тот факт, что в отсортированной таблице информацию можно найти на порядок быстрее, чем в неотсортированной.

Индекс – это упорядоченный (буквенный или числовой) список столбцов или групп столбцов в таблице. Таблицы могут иметь большое количество строк, а, так как строки не находятся в определенном порядке, на их поиск по указанному значению может потребовать время.

В то время как индекс значительно улучшает эффективность запросов, использование индекса несколько замедляет операции модификации данных (такие как INSERT, UPDATE и DELETE), а сам индекс занимает дополнительный объем дисковой памяти. Следовательно, каждый раз, когда вы создаете таблицу, необходимо принять решение, индексировать ее атрибуты или нет. Индексы могут состоять из многочисленных полей. Если больше чем одно поле указывается для одного индекса, то второе упорядочивается внутри первого, третье внутри второго, и так далее.

Для удаления индексов из схемы базы данных, необходимо воспользоваться командой **DROP INDEX**.

[1] Таким образом, индексы не являются обязательной частью базы данных в теории, но на практике при больших объемах баз данных без индексов невозможно обеспечить приемлемого быстродействия.

7. ЗАЩИТА БАЗ ДАННЫХ

Если СУБД используется в крупной организации для какой-либо информационной системы, то ее данные являются ценным ресурсом, доступ к которому необходимо строго контролировать и регламентировать как и к другим корпоративным ресурсам. СУБД должна гарантировать, что созданная в ее среде база данных будет надежно защищена. Термин «защита» относится к защищенности базы данных от несанкционированного доступа, как намеренного, так и случайного. Однако это не только набор стандартных служб[1].

Защита базы данных – обеспечение защищенности базы данных против любых предумышленных и непредумышленных угроз с помощью различных компьютерных и некомпьютерных средств.

Таким образом, для защиты базы данных должна быть осуществлена система мероприятий, включающая в себя анализ/адаптацию используемого оборудования, программного обеспечения, персонала и самих данных с целью создания максимального уровня защищенности.

Наиболее распространенными рисками, связанными с защитой базы данных являются:

- риск похищения и/или фальсификации данных;
- утрата конфиденциальности (нарушение тайны);
- утрата целостности и/или потеря доступности данных.

Для того, чтобы составить обобщенную схему потенциальных опасностей, создадим классификацию рисков по ***источникам возникновения***:

Оборудование	Пожар, затопление, диверсии, отключение питания, кража, физическое повреждение и т.п.
СУБД и ПО	Искажение программ, кража ПО, отказ механизмов защиты из-за расширения возможностей доступа
Сети	Обрыв, отключение, электрические наводки, радиация, подключение к кабельной системе
Базы данных	Несанкционированная корректировка или копирование данных, разрушение данных
Пользователи	Чужие средства доступа, несанкционированный просмотр, внедрение вирусов, занесение некорректной информации

[1] Хорошую аналогию в данном случае можно сделать с установкой противоугонных систем на новый автомобиль. Только комбинация каких-либо средств, желательно нестандартных, будет с достаточно большой долей вероятности защищать от угона. Однако, естественно, стопроцентной гарантии не будет ни при какой комбинации средств.

Программисты	Создание лазеек в систему, недостаточный уровень разработки ограничений и процедур защиты
Администраторы БД	Недостаточный уровень настройки ограничений и процедур защиты

Некоторые из опасностей носят случайный характер, однако именно случайные опасности являются причиной основной части потерь в большинстве организаций. Они должны фиксироваться и анализироваться на предмет являются ли они действительно случайными.

Типы угроз лежат в широком диапазоне – от пожаров до пролива кофе на клавиатуру. Необходима всесторонняя оценка рисков. В крупных организациях для этого создаются специальные группы, в которые входят сотрудники различных подразделений, например отдела информационных технологий, отдела кадров, юридической службы, службы, отвечающей за эксплуатацию строений и т.п. Ниже в соответствующих разделах рассматриваются некомпьютерные и компьютерные средства защиты баз данных.

Некомпьютерные средства защиты

К некомпьютерным средствам защиты относится следующий комплекс мероприятий:

- меры обеспечения безопасности и планирование защиты от непредвиденных обстоятельств (документы, регламентирующие деятельность организации в плане защиты информации);
- контроль за персоналом;
- защита помещений и хранилищ;
- договора о сопровождении;
- контроль за физическим доступом (внутренний, внешний контроль).

Меры обеспечения безопасности

В документе по мерам обеспечения безопасности должно быть определено следующее:

- область деловых процессов организации;
- ответственность и обязанности отдельных работников;
- дисциплинарные меры;
- процедуры, которые должны обязательно выполняться.

Процедуры, определяемые как меры обеспечения безопасности, могут потребовать разработки других процедур. В подобном случае документ с определением мер обеспечения безопасности постоянно сохраняет свою значимость, хотя может потребоваться его периодический пересмотр, тогда как выполняемые процедуры реализации этих требований должны мо-

дифицироваться при каждом внесении изменений в систему или модернизации используемой технологии.

Планирование защиты от непредвиденных обстоятельств

План защиты от непредвиденных обстоятельств разрабатывается с целью подробного определения последовательности действий, необходимых для выхода из различных необычных ситуаций, например в случае пожара или диверсии. Типичный план такой защиты должен включать в себя такие элементы, как:

- сведения о том, кто является главным ответственным лицом;
- информация о том, кто и на каком основании принимает решение о возникновении необычной ситуации;
- технические требования к передаче управления резервным службам;
- организационные требования в отношении персонала;
- сведения о наличии страховки на случай данной ситуации.

Любой разработанный план защиты должен периодически пересматриваться и тестироваться на предмет его осуществимости.

Контроль за персоналом

С точки зрения защиты системы, исключительно важную роль играют отношение к делу и действия людей, непосредственно вовлеченные в управление информационной системой[1]. Основной риск в любой организации связан с действиями, производимыми сотрудниками самой организации, а не с возможными внешними угрозами. Отсюда следует, что обеспечение необходимого уровня контроля за персоналом позволяет минимизировать возможный риск.

Защита помещений и хранилищ

Для любой организации жизненно необходимо иметь надежное защищенное место, предназначенное для хранения носителей информации, копий программ, прочих архивных материалов и документации. Предпочтительно, чтобы это помещение находилось на площадке, отличной от места расположения основного оборудования системы. Все, что хранится в подобном помещении, должно быть зарегистрировано в специальном каталоге, с указанием даты помещения носителя или документа на хранение.

[1] В данном случае внесение информации в какую-либо ИС также будет являться управлением ею, так как некорректные данные могут привести к нарушению целостности или потере непротиворечивости.

Договора о сопровождении

Для всего используемого организацией оборудования и программного обеспечения сторонней разработки, обязательно должны быть заключены соответствующие договора о сопровождении. Установленный интервал ожидания ответа в случае какого-либо отказа или ошибки зависит от важности отказавшего элемента для нормального функционирования всей системы.

Контроль за физическим доступом

Контроль за физическим доступом включает внутренний и внешний контроль. Внутренний контроль используется внутри отдельных зданий. Наиболее важные помещения могут быть оборудованы системами входного контроля. Внешний метод контроля применяется вне строений и предназначен для ограничения доступа на площадку или отдельные здания.

Компьютерные средства защиты

К компьютерным средствам защиты относятся следующие действия:

- авторизация пользователей;
- использование представлений как простейшего средства защиты;
- резервное копирование и восстановление;
- поддержка целостности;
- шифрование;
- вспомогательные процедуры.

Авторизация пользователей

Авторизация – предоставление прав или привилегий, позволяющих их владельцу иметь законный доступ ко всей системе или к ее отдельным частям

Обычно авторизация включает права на объекты и права на операции. К объектам, в частности, относятся таблицы и представления[1]. К операциям – вставка записей, их редактирование и удаление. Процесс авторизации включает в себя процесс аутентификации субъектов, требующих получения доступа к объектам. Практически во всех СУБД данный процесс подразумевает под собой необходимость ввода в соответствующей форме имени пользователя и пароля.

Выделяют два типа систем – **открытые** и **закрытые**. В открытых системах после успешно проведенной процедуры аутентификации пользователь получает доступ ко всем объектам базы данных и ко всем операциям

[1] Таким образом, пользователь обладает полномочием запрашивать какую-либо информацию из конкретной таблицы или представления.

над ее объектами[1]. В закрытых же системах после аутентификации пользователь наделяется только теми правами и полномочиями, которые для него определил администратор базы данных[2].

Использование представлений

Ранее в настоящем пособии уже упоминалось о возможности использования представлений как самого простейшего средства защиты от несанкционированного доступа. Действительно, представления являются мощным и гибким инструментом, позволяющим скрывать от определенных пользователей некоторые части базы данных.

Предположим, создается информационная система, предназначенная как для сотрудников отдела кадров, так и для бухгалтерии. Естественно, что часть данных, относящихся к персоналу должны быть общими для обоих отделов. Однако сотрудники отдела кадров не должны видеть данные по заработной плате, а сотрудники бухгалтерии – данные по образованию сотрудников и повышению их квалификации. Поэтому необходимо будет создать набор представлений, адаптированных под эти две службы и в приложениях пользователей реализовать доступ через данные представления, а не таблицы. Данный способ защиты является достаточным для защиты пользователей от самих себя, но не от злоумышленников. Для того, чтобы повысить степень защиты, можно построить приложение в трехзвенной архитектуре с использованием доступа через представления. На рисунке **43** показана двухзвенная архитектура приложения с базами данных, которая используется в большинстве приложений. В них программа состоит из клиентской и серверной части. Клиентские машины соединяются непосредственно с СУБД и посылают ей запросы. Сервер же эти запросы обрабатывает и посылает клиентам результаты выборок.

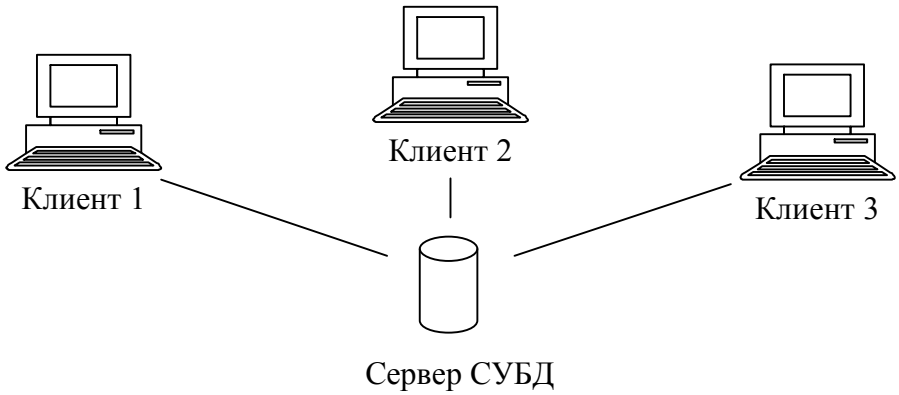

Рис. 43. Двухзвенная архитектура

[1] К открытым системам, в частности, относится СУБД MySQL.

[2] К закрытым системам относятся практически все современные мощные СУБД, например, Oracle, MSSQL, Interbase и т.п.

В случае, если злоумышленник каким-то образом перехватит соединение клиентской машины с сервером СУБД, то он имеет потенциальную возможность получить доступ к информации, превышающей полномочия текущего пользователя.

На рисунке **44** показана трехзвенная архитектура приложения с базами данных. В ней между клиентским приложением и базой данных располагается сервер приложений, который физически часто располагается на отдельном сервере[1]. Клиентские машины теперь напрямую с СУБД не соединяются, а взаимодействуют только с сервером приложений, который преобразует запросы клиентских машин в запросы, понятные СУБД. Результаты запросов поступают с сервера СУБД сначала на сервер приложений, который, в свою очередь, преобразует результат в вид, понятный для клиентской машины.

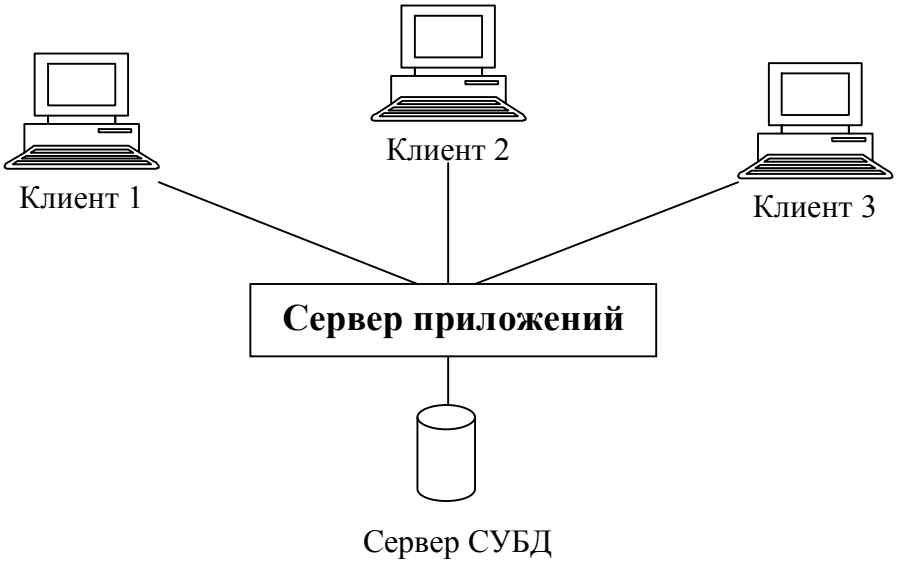

Рис. 44. Трехзвенная архитектура

Трехзвенная архитектура имеет как достоинства, так и недостатки. К достоинствам можно отнести более легкий способ перехода на новый сервер СУБД в случае необходимости. Также в данном случае повышается уровень безопасности данных. В случае, если злоумышленник каким-то образом перехватит соединение клиентской машины с сервером приложения[2], то он уже не сможет получить доступ к информации, превышающей полномочия текущего пользователя. Для этого достаточно заменить обращение к таблицам обращением к представлениям, составленным с учетом полномочий текущего пользователя. Однако трехзвенная архитектура име-

[1] Если приложение разрабатывается по операционную систему Windows, то сервер приложений, как правило, представляет собой DCOM-приложение.

[2] Так как клиентское приложение уже не связано непосредственно с сервером СУБД, то на уровне клиента потенциально можно перехватить только данное соединение.

ет также и недостатки. Во-первых, ввиду того, что приложение теперь состоит не из двух частей, а из трех, то повышается вероятность возникновения ошибок. Кроме этого, приложение, построенное в рамках трехзвенной архитектуры всегда будет выполняться медленнее, чем для двухзвенной. Это будет иметь место опять же из-за появления среднего звена и отсутствия прямого соединения между клиентом и сервером.

Резервное копирование и восстановление

Средства резервного копирования – обязательная составляющая современных СУБД[1]. Как правило, это некие исполнимые файлы, которые запускаются в консольном режиме. Рекомендуется организовать процесс автоматического создания резервных копий базы данных с некоторой установленной периодичностью. Иногда требуется более широкий набор средств, нежели полное копирование, либо полное восстановление. Например, восстановление какой-либо записи на определенный момент времени. В этом случае, кроме всего прочего, понадобится создание дополнительных таблиц, в которые будут заноситься все изменения над записями таблиц с указанием времени изменения и пользователя, инициировавшего изменения.

Поддержка целостности

Средства поддержки целостности данных также вносят определенный вклад в общую защищенность базы данных, поскольку их назначением является предотвращение перехода данных в несогласованное состояние, а значит, и предотвращение угрозы получения ошибочных или некорректных результатов запросов.

Некоторые СУБД не имеют средств поддержки целостности, например, СУБД Foxpro 2.6. В этом случае потребуется программно поддерживать целостность данных.

Шифрование

Если в СУБД содержится весьма важная и конфиденциальная информация, то имеет смысл закодировать ее с целью предупреждения возможной угрозы несанкционированного доступа с внешней стороны. Некоторые СУБД включают средства шифрования, предназначенные для использования в подобных целях. Подпрограммы таких СУБД обеспечивают санкционированный доступ к данным (после их декодирования), хотя это будет связано с некоторым снижением производительности, вызванных необхо-

[1] В случае, если средства резервного копирования в выбранной СУБД все же отсутствуют, их придется реализовывать программно.

димостью перекодировки[1]. Шифрование также может использоваться для защиты данных при их передаче по линиям связи.

Вспомогательные процедуры

Описанные ранее различные механизмы могут использоваться для защиты данных в среде СУБД. Однако сами по себе они не гарантируют необходимого уровня защищенности и могут оказаться неэффективными в случае неправильного применения или управления. По этой причине в данном разделе рассматриваются различные вспомогательные процедуры, которые должны использоваться совместно с описанными выше механизмами защиты.

Авторизация и аутентификация

Для того, чтобы минимизировать риск взлома паролей пользователей, их необходимо периодически менять, для этого в организации должна быть разработана и внедрена соответствующая процедура. Кроме этого необходимо наличие способов обнаружения попыток несанкционированного доступа к информации или искажения ее, случайного или намеренного. В случае, если какой-либо сотрудник увольняется с работы, необходима процедура оперативного удаления всех его учетных записей из оперативной базы данных.

Копирование и восстановление

Кроме того, что у базы данных должны быть средства резервного копирования и восстановления, необходимо, чтобы данные средства использовались, причем с наибольшей эффективностью и с наименьшими затратами[2]. Именно поэтому должны быть разработаны процедуры резервного копирования. На случай возникновения какого-либо критического сбоя должны существовать четкие инструкции с описанием последовательности операций, с помощью которых за наименьшее время база данных вернется в рабочее состояние[3].

---

[1] Автор пособия достаточно скептически относится к шифрованию как средству защиты баз данных. Естественно зашифровать информацию можно, однако, одна из важнейших функций СУБД – это быстрый доступ к информации посредством SQL-запросов. Здесь-то и возникнут проблемы, т.к. в случае, если значения какое-либо поля будут зашифрованы, то по ним не получится использовать полноценный поиск, например, по шаблону. Поэтому шифрование возможно применять только к наиболее важной информации, по которой не потребуется осуществлять непосредственный поиск с помощью SQL-запросов.

[2] Имеются ввиду затраты, связанные с нагрузками на сервер СУБД и человеческие затраты

[3] Как правило, это специальные инструкции, адресованные администраторам баз данных.

Аудит

Одно из назначений процедуры аудита состоит в проверке того, все ли предусмотренные средства управления задействованы и соответствует ли уровень защищенности установленным требованиям. Регулярное проведение аудиторских проверок, дополненное постоянным контролем содержимого файлов журналов с целью выявления анормальной активности в системе, очень часто позволяет своевременно обнаружить и пресечь любые попытки нарушения защиты.

Установка нового прикладного программного обеспечения

Новые приложения, разработанные собственными силами или сторонними организациями, обязательно следует тщательно протестировать, прежде чем передавать их в промышленную эксплуатацию. В противном случае существенно возрастает риск разрушения базы данных. Следует считать хорошей практикой выполнения резервного копирования базы данных непосредственно перед сдачей нового программного обеспечения в эксплуатацию. Кроме этого, в первый период эксплуатации нового приложения обязательно следует организовать тщательное наблюдение за функционированием системы.

Системы защиты от несанкционированного доступа

В крупной организации, использующей несколько информационных систем, очень остро строит вопрос построения надежной защиты от несанкционированного доступа. Каждый пользователь должен видеть только тот набор данных и иметь возможность осуществлять только от набор функций, который за ним закреплен должностными инструкциями и ничего более. В противном случае многократно повышается риск неумышленного искажения информации, а также появление сбоев в работе информационных систем. Защиту от несанкционированного доступа можно построить различными способами. В настоящем разделе приводятся различные способы построения данной защиты.

Обеспечение безопасности средствами СУБД

Практически во всех современных СУБД существует возможность ограничить выполнение определенных действий определенным временем и кругом пользователей. Общая модель безопасности СУБД представлена на рис. **45**.

Пользователю может быть назначена одна или несколько ролей, а роль может принадлежать одному или многим пользователям. Как пользователи, так и роли могут иметь много различных полномочий. С каждым объектом (в широком смысле этого слова) связаны определенные полно-

мочия. Каждое полномочие относится к одному пользователю или группе и одному объекту.

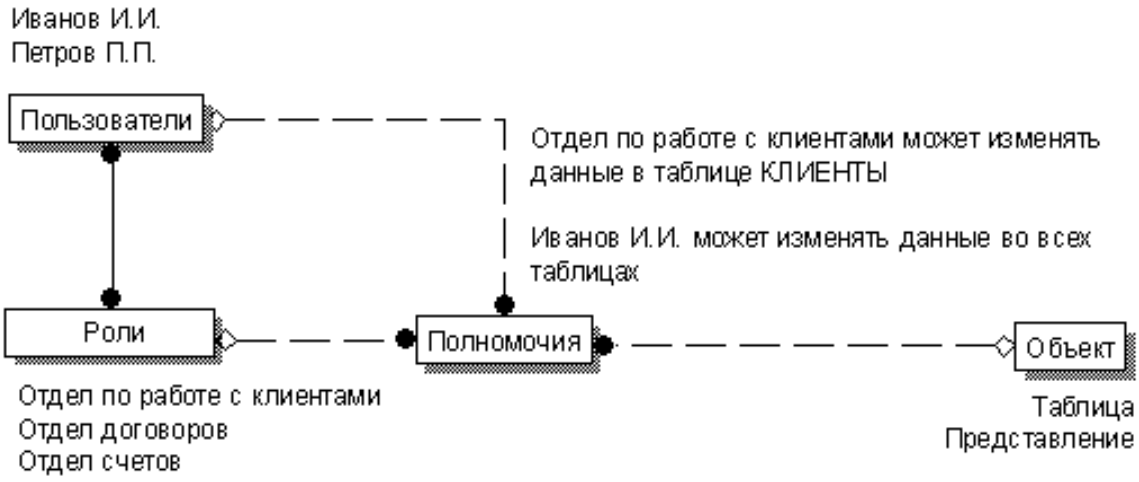

Рис. 45. Модель безопасности СУБД

Когда пользователь входит в систему базы данных, СУБД ограничивает его действия полномочиями, определенными индивидуально для данного пользователя, а также для роли, назначенной данному пользователю. Определить, является ли пользователь тем, за кого он себя выдает, – задача, вообще говоря, сложная. Во всех коммерческих СУБД используется тот или иной вариант парольной защиты, даже при том, что такой метод обеспечения безопасности можно легко обойти, если пользователи небрежно относятся к своим личным идентификаторам.

Как правило, пользователи самостоятельно вводят свое имя и пароль, а в некоторых приложениях имя пользователя и пароль вводятся системой от лица пользователя. Например, Windows 2000 может непосредственно передать имя пользователя и пароль для входа в систему СУБД SQL Server. В других случаях имя пользователя и пароль предоставляются прикладной программой.

Обеспечение безопасности средствами приложения

Хотя современные СУБД, такие как Oracle или SQL Server, предоставляют значительные возможности по обеспечению безопасности, эти возможности по своей натуре являются весьма общими. Если приложению требуются специфические меры безопасности, например «пользователь не может просматривать договора, превышающие 1000000 рублей», то средствами СУБД в плане создания полномочий на таблицы, эту задачу не решить. В таких случаях система безопасности может быть дополнена функциями, реализованными в самом приложении. Обеспечение безопасности средствами приложения следует рассматривать как крайнюю меру, когда

никакими другими способами задача не может быть решена на серверной части[1].

Обеспечение безопасности средствами приложения, как правило, сводится к написанию SQL-запросов с учетом идентификаторов пользователей, которые в данный момент работают с приложением. В этом случае вся защита, что называется, «на виду», т.е. заполучив исходные тексты программы, либо перехватив запросы, идущие с клиентского приложения на сервер, в принципе, можно разобраться с логикой работы системы защиты и, при желании, обойти ее.

Есть много других способов расширения возможностей СУБД по обеспечению безопасности за счет приложения. Однако в первую очередь следует задействовать систему безопасности СУБД. Только если ее функций оказывается недостаточно, стоит дополнять их за счет прикладных программ. Чем ниже уровень, на котором идет обеспечение безопасности, тем меньше возможностей для несанкционированного вторжения. Кроме того, система безопасности СУБД работает быстрее, ее использование приводит к меньшим затратам и, возможно, дает лучшие результаты, чем разработка собственной системы.

Привилегии в СУБД

Каждый пользователь в SQL-базе данных имеет набор привилегий. Это то, что пользователю разрешается делать. Эти привилегии могут изменяться со временем – новые добавляться, старые удаляться. Некоторые из этих привилегий определены в ANSI SQL, но имеются и дополнительные привилегии, которые являются также необходимыми.

SQL привилегии, определенные в ANSI – это привилегии объекта. Это означает, что пользователь имеет привилегию, чтобы выполнить данную команду только на определенном объекте в базе данных. Привилегии объекта связаны одновременно и с пользователями, и с таблицами. То есть, привилегия дается определенному пользователю в указанной таблице, или базовой таблице или представлении. Пользователь, создавший таблицу, является владельцем этой таблицы. Это означает, что пользователь имеет все привилегии в этой таблице и может передавать привилегии другим пользователям в этой таблице.

[1] Например, запрещение печати отчетов для выбранных пользователей через определенное приложение. В этом случае на серверной части можно реализовать систему хранения данных по конкретным элементам интерфейса приложения пользователя с одновременным сопоставлением конкретному элементу группы пользователей, имеющих право с ним работать. Однако для того, чтобы программно закрыть тот или иной элемент интерфейса, необходимо будет внести изменение в код приложения.

Привилегии, которые можно назначить пользователю:

SELECT	Возможность выполнять запросы в таблице
INSERT	Возможность вставлять записи в таблицу
UPDATE	Возможность изменять записи в таблице
DELETE	Возможность удалять записи в таблице
REFERENCES	Возможность создавать внешний ключ на таблицу[1]

Для назначения пользователям привилегий, в языке SQL предназначена команда **GRANT**. Например, если необходимо разрешить пользователю Иванову обращаться к таблице заказов, то потребуется следующий оператор:

```
GRANT SELECT ON Заказы TO Иванов;
```

Когда SQL получает команду GRANT, он проверяет привилегии пользователя, подавшего эту команду, чтобы определить, допустима ли эта команда. Сам Иванов, естественно, самостоятельно не может выдать эту команду. Для того, чтобы разрешить Иванову также редактировать записи в таблице заказов, потребуется выполнить следующий SQL-оператор:

GRANT UPDATE ON Заказы **TO** Иванов;

Однако, если необходимо ограничить Иванова в изменении только количества заказанного товара и наименования товара, потребуется небольшая корректировка оператора:

GRANT UPDATE (Кол-во, Наим_товара) **ON** Заказы **TO** Иванов;

Иногда, создателю таблицы хочется, чтобы другие пользователи могли получить привилегии в его таблице. Обычно это делается в системах, где один или более разработчиков создают несколько базовых таблиц, а затем передают ответственность за них тем, кто будет фактически с ними работать. SQL позволяет делать это с помощью предложения WITH GRANT OPTION. Если необходимо дать Иванову право предоставлять привилегию SELECT в таблице заказов другим пользователям, необходимо выполнить оператор:

```
GRANT SELECT ON Заказы TO Иванов WITH GRANT OPTION;
```

[1] Кроме этого, существует ряд привилегий, которые не являются стандартными, например, право создавать индекс, изменять структуру таблицы и т.п. Для того, чтобы узнать весь набор привилегий конкретной СУБД необходимо тщательно изучить специфическую техническую литературу, желательно непосредственно от самого разработчика СУБД.

Для отмены привилегий предназначена команда **REVOKE**. Она похожа по синтаксису на команду GRANT, но имеет обратный смысл. Чтобы удалить привилегию INSERT для Иванова в таблице Заказов, можно ввести

```
REVOKE INSERT ON Заказы FROM Иванов;
```

Варианты реализации защиты от несанкционированного доступа

В данном разделе приводятся способы реализации защиты от несанкционированного доступа с обсуждением достоинств и недостатков каждого из способов. Причем рассматриваются защиты на уровне отдельных записей.

Способ 1. Реализация защиты на клиентском приложении

Данный способ уже обсуждался выше. Обеспечение безопасности сводится к написанию SQL-запросов с учетом идентификаторов пользователей, которые в данный момент работают с приложением.
Плюсы: относительная простота реализации, неплохая производительность.
Минусы: низкий уровень защищенности (только через текущее приложение), легкое обнаружение.

Способ 2. Создание представлений для каждого пользователя

В рамках данного способа создается представление для каждого пользователя или группы пользователей. В приложении пользователя же необходимо динамически переключаться на соответствующие представления в зависимости от идентификатора текущего пользователя.
Плюсы: относительная простота первоначальной реализации, неплохая производительность.
Минусы: сложность администрирования, отладки и сопровождения. Высокий риск появления ошибок при изменении структуры таблиц.

Способ 3. Использование функций в запросах

В рамках данного способа предлагается доработать предыдущий способ в плане автоматизации создания представлений. Предлагается писать представления по образцу:

```
SELECT * FROM TABLE_NAME WHERE FUNC(REC_ID,USER_ID)=1
```

Таким образом предлагается написать функцию с двумя параметрами. Один из них – это значение первичного ключа таблицы TABLE_NAME, а второй – значение идентификатора пользователя. Если функция возвращает еди-

ницу, это значит, что текущую запись пользователь имеет полномочия видеть, в противном случае нет. Естественно, функцию придется писать вручную, однако процесс создания представлений можно автоматизировать.
Плюсы: относительная простота реализации.
Минусы: необходимость написания достаточно сложных функций. Низкая производительность, связанная с невозможностью использования индексов из-за использования функций в представлении.

Способ 4. Использование триггеров[1]

В данном случае предлагается вместо создания отдельных представлений создавать триггеры – для операторов SELECT, INSERT, UPDATE и DELETE, которые при построчном просмотре выполняемых пользователем изменений будут одни действия разрешать, а другие отвергать.
Плюсы: высокий уровень защищенности.
Минусы: низкая производительность, сложность создания и администрирования.

Способ 5. Разработка системы защиты с использованием уникальных возможностей какой-либо СУБД

Все предлагаемые выше способы построения системы защиты были достаточно универсальными. Действительно, первый способ можно реализовать на любой СУБД, которая поддерживает SQL-запросы. Второй способ возможно реализовать в СУБД, в которых существует возможность создания представлений. Третий способ возможен, если СУБД поддерживает возможность создания функций пользователей, четвертый – триггеров.

В данном случае предлагается разработка системы защиты с использованием уникальных возможностей какой-либо СУБД. Для этого необходимо детально ознакомиться с уникальными возможностями целевой СУБД в плане безопасности.
Плюсы: высокий уровень защищенности, высокий уровень производительности.
Минусы: сильная привязанность к какой-либо СУБД, невозможность быстрого перехода на другую СУБД в случае необходимости. Сложность разработки.

Ниже описывается возможность разработки системы защиты данным способом с использования СУБД Oracle. Используется средство Oracle, называемое fine grained access control, которое в переводе означает ***тщательный контроль доступа***.

Тщательный контроль доступа в Oracle

[1] **Триггеры** – это процедуры на языке SQL, которые выполняются **автоматически** при выполнении определенных действий в базе данных, например, вставке в таблицу.

Средство тщательного контроля доступа появились в СУБД Oracle, начиная с версии 8.1.5. Оно позволяет во время выполнения динамически добавлять условие (конструкцию WHERE) ко всем запросам, обращенным к таблице или представлению базы данных[1]. Теперь можно процедурно изменять запрос во время выполнения, другими словами, динамически создавать представление. Можно проверить, кто выполнял запрос, с какого терминала и когда (например, по времени суток) он выполнялся, а затем создать условие на основе этой информации. С помощью контекстов приложений[2] можно безопасно добавлять в среду информацию (например, роль пользователя в отношении приложения) и обращаться к этой информации в процедуре или условии.

Средства тщательного контроля доступа позволяют с помощью одной таблицы и одной хранимой процедуры[3] справиться с задачей, для решения которой могло бы понадобиться несколько представлений или триггеров, или большой объем специализированной обработки в приложениях.

Подход с использованием нескольких представлений достаточно типичен[4]. Разработчики приложений создают несколько учетных записей в базе данных, например **EMPLOYEE, MANAGER** и устанавливают для каждого из пользователей полный набор представлений, выбирающих только необходимые данные. Для управления тем, что конечные пользователи могут просматривать, создавать, изменять и удалять, придется создавать до четырех различных представлений. Это быстро приводит к запредельному увеличению количества объектов базы данных - каждый раз при добавлении новой группы пользователей придется создавать и поддерживать новый набор представлений.

Если правила защиты изменятся (например, если необходимо разрешить руководителям просматривать записи не только непосредственных подчиненных, но и подчиненных следующего уровня), придется пересоздать представление в базе данных, делая недействительными все объекты, которые на него ссылаются. Такой подход приводит не только к увеличению количества представлений в базе данных, но и требует, чтобы пользователи регистрировались от имени нескольких совместно используемых учетных записей, что осложняет контроль за их работой.

---

[1] Причем внешне это никак не будет видно.

[2] Для каждого пользователя, работающего с приложением, выделяется отдельная область памяти, которая доступна только ему и в которую можно заносить значения каких-либо переменных или идентификаторов.

[3] Хранимые процедуры – это подпрограммы на языке SQL, хранящиеся в базах данных и представляющие собой один из видов их общих ресурсов. Тело любой хранимой процедуры представляет последовательность SQL-операторов, например таких, как выборка данных (SELECT), их модификация (UPDATE), удаление данных (DELETE), операторы цикла (LOOP), условные операторы (IF, CASE) и ряд других.

[4] Имеется в виду третий способ построения системы защиты, описанный ранее.

Средства тщательного контроля доступа позволяют справиться с этими трудностями и избежать потери функциональных возможностей с помощью всего двух объектов – исходной таблицы (или представления) и пакета[1] (или функции) базы данных. Пакет можно изменить в любой момент, разработав новые правила защиты. Вместо поддержки десятков представлений, реализующих правила защиты для объекта, всю соответствующую информацию можно задавать в одном месте.

При тщательном контроле доступа алгоритмы защиты, определяющие, какие данные может "видеть" пользователь, помещаются в базу данных. При этом гарантируется защита данных, независимо от используемого средства доступа к ним.

Потребность в таких средствах вполне объяснима. В начале и середине 1990-х годов преимущественно использовалась модель клиент-сервер (а еще раньше нормой считалось централизованное выполнение приложений). Большинство клиент-серверных приложений (и практически все централизованные) включали алгоритмы, проверяющие уровень доступа к приложению. Сегодня очень модно использовать серверы приложений[2] и размещать на них все прикладные алгоритмы. По мере переноса клиент-серверных приложений на новую архитектуру разработчики начали переносить алгоритмы защиты с клиентской части и встраивать их в серверы приложений. Это привело к двойной реализации алгоритмов защиты (некоторые клиент-серверные приложения продолжают использоваться), так что теперь поддерживать и отлаживать эти алгоритмы надо в двух местах. Ситуация станет еще хуже при появлении следующей парадигмы программирования. Только при наличии централизованной системы защиты, реализованной на сервере возможно достаточно безболезненно переходить с одной архитектуры приложения на другую.

Средства тщательного контроля доступа позволяют отделить алгоритмы защиты от других алгоритмов работы приложения. Разработчик приложения может заняться прикладными алгоритмами, а не алгоритмами безопасного доступа к данным. Поскольку тщательный контроль доступа выполняется полностью на сервере баз данных, эти алгоритмы немедленно наследуются всеми приложениями. Раньше разработчикам приходилось включать алгоритмы защиты в приложения, что усложняло разработку и дальнейшее сопровождение приложений. Если приложения должны обеспечивать защиту доступа к данным, а доступ к одним и тем же данным выполняется во многих компонентах приложения, изменение правил защиты повлияет на десятки модулей приложения. При использовании средств тщательного контроля доступа все соответствующие модули автоматически наследуют новые правила доступа без каких-либо изменений в коде.

---

[1] Пакет – совокупность логически связанных типов, программных объектов и подпрограмм.

[2] Имеется в виду трехзвенная архитектура, рассмотренная ранее.

Средства тщательного контроля доступа в Oracle реализуются с помощью двух конструкций:

- **Контекст приложения**. Это пространство имен с соответствующим набором пар атрибут/значение. Контекст приложения всегда привязан к пакету. Этот пакет – единственный метод установки значений в контексте, что предотвращает установку значений в контексте приложений злонамеренными пользователями с целью получения несанкционированного доступа к информации. Читать значения в контексте приложения может кто угодно, но устанавливать их может только один пакет.
- **Правила защиты**. Это созданная разработчиком функция, возвращающая условие для динамического фильтрования данных при выполнении запроса. Эту функцию можно привязывать к определенной таблице или представлению, и вызываться она может для всех или только для некоторых операторов, обращающихся к таблице. Это означает, что можно задать одни правила для оператора SELECT, другие – для INSERT и третьи – для операторов UPDATE и DELETE. Обычно в этой функции используются значения атрибутов в контексте приложений для создания соответствующего условия (например, проверяется, кто зарегистрировался и что он пытается сделать и создается соответствующее подмножество строк для работы). Следует помнить, что для пользователя **SYS** (или **INTERNAL**) правила защиты никогда не вызываются, эти пользователи всегда могут читать и изменять данные[1].

К сожалению, для того, чтобы по-настоящему пользоваться возможностями средств тщательного доступа к данным, скорее всего придется писать соответствующее программное обеспечение для администратора, т.к. никаких приложений, ориентированных на пользователей в этом плане, в Oracle не разработано.

[1] Это и хорошо и плохо. Хорошо, т.к. не возникнет случая, что из-за неправильной настройки системы безопасности никто не сможет увидеть записи таблицы. Плохо, т.к. в случае, если злоумышленник каким-то образом узнает пароли либо пользователя SYS, либо пользователя INTERNAL, то на этом о защите данных можно просто забыть.

8. ЗАДАЧИ ДЛЯ ЛАБОРАТОРНЫХ РАБОТ

В данном разделе приводится список предлагаемых лабораторных работ. Обычно одна задача последовательно разрабатывается в рамках всего курса СУБД. Для выбранной задачи необходимо разработать логическую схему базы данных. Далее необходимо рассмотреть вопрос нормализации полученной схемы. В случае необходимости провести ряд процедур нормализации с тем, чтобы вся схема удовлетворяла как минимум четвертой нормальной форме. Далее необходимо выбрать целевую СУБД и для нее разработать физическую схему базы данных, после чего разработать приложение для работы с ней. Ограничений на среду разработки и на тип СУБД не накладывается.

Список задач:

1. Автоматизация работы склада готовой продукции.
2. Автоматизация работы пунктов проката кассет.
3. Система учета товаров в магазине.
4. Библиотечная система.
5. Система поддержки составления расписания занятий.
6. Система учета успеваемости студентов на факультете.
7. Система учета успеваемости студентов в ВУЗе.
8. Ведение домашней бухгалтерии.
9. Оценка поставщиков согласно требованиям стандарта ГОСТ Р ИСО 9001-2000.
10. Составление кулинарных рецептов.
11. Анализ динамики курсов валют за период времени в различных банках.
12. Анализ динамики курсов акций за период времени на различных биржах.
13. Автоматизация хранения компакт дисков.
14. Автоматизация процедуры ОСАГО с точки зрения страховых агентов.
15. Ведение истории развития юридических лиц.
16. Ведение штатного расписания для крупной организации.
17. Разработка ежедневника/телефонной книги.
18. АРМ[1] расчетов за телефонные звонки.
19. АРМ технического обслуживания автомобилей.
20. Опись имущества в крупной организации.
21. Автоматизация документооборота в организации.
22. Учет рабочего времени сотрудников.
23. Создание иерархической записной книжки.
24. Автоматизация процесса планирования проектов.
25. Задача учета компьютеров, включая изменения в их конфигурации.
26. Автоматизация процесса продажи железнодорожных билетов.

[1] АРМ – автоматизированное рабочее место.

ЗАКЛЮЧЕНИЕ

Развитие технологий, связанных с системами управления базами данных не останавливается ни на минуту. Высокий темп разработок диктует сам рынок – для того, чтобы занять достойное место в конкурентной борьбе, фирмам разработчикам баз данных приходится придумывать все новые и новые механизмы в плане надежности хранения данных, обеспечения безопасности, увеличения скорости доступа и т.п. В настоящем пособии не нашло отражение достаточно большое количество технологий, например, распределенные базы данных, иерархические и объектные СУБД и т.п. Нельзя объять необъятное. Только после того, как заинтересованный читатель попробует создать свою собственную базу данных, только после того, как он почувствует искру творчества в этом нелегком деле, только после этого, на взгляд автора, имеет смысл идти дальше в плане повышения уровня своей компетенции в базах данных. Если у кого-то из читателей возникло желание идти вперед в этом направлении, автор считает свою задачу выполненной.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Коннолли, Томас, Бегг, Каролин, Страчан, Анна. Базы данных: проектирование, реализация и сопровождение. Теория и практика, 2-е изд.; пер. с англ. М.: Издательский дом “Вильямс”, 2001. 1120 с.
2. Д. Кренке. Теория и практика построения баз данных. 8-е изд. СПб.: Питер, 2003. 800 с.
3. Крутов А.Н. Методы программирования. ООП. UML. RUP: учебное пособие. Самара: Изд-во «Самарский университет», 2004. 116 с.
4. Грабер М. Введение в SQL. М.: ЛОРИ, 1996. 380 с.
5. Том Кайт. Oracle для профессионалов. Книга 2. Расширение возможностей и защита. 2-е издание. Киев: ООО “ТИД “ДС”, 2004. 848 с.
6. Хоббс Л., Хилсон С., Лоуенд Ш. Oracle9iR2: разработка и эксплуатация хранилищ баз данных; пер. с англ. М.: КУДИЦ-ОБРАЗ, 2004. 592 с.

Электронное учебное издание

Крутов Алексей Николаевич

ПРОЕКТИРОВАНИЕ И РАЗРАБОТКА БАЗ ДАННЫХ

Электронное учебное пособие

Публикуется в авторской редакции
Титульное редактирование *С. В. Жидовой*
Компьютерная верстка, макет *Т. В. Кондратьевой*

Управление по информационно-издательской деятельности Самарского государственного университета: www.infopress.samsu.ru
Издательство «Самарский университет», 443011, г. Самара, ул. Акад. Павлова, 1.
Тел. 8 (846) 334-54-23