Ю.Ю. ГРОМОВ, О.Г. ИВАНОВА, В.Н. ТОЧКА

УПРАВЛЕНИЕ ДАННЫМИ

ИЗДАТЕЛЬСТВО ТГТУ

УДК 681.518(075)
ББК 3973.26-018.3я73
Г874

Р е ц е н з е н т ы :

Доктор физико-математических наук, профессор,
заслуженный деятель науки РФ
В.Ф. Крапивин

Доктор физико-математических наук, профессор
Ф.А. Мкртчян

Громов, Ю.Ю.

Г874 Управление данными : учебное пособие / Ю.Ю. Громов, О.Г. Иванова, В.Н. Точка. – Тамбов : Изд-во Тамб. гос. техн. ун-та, 2009. – 80 с. – 100 экз. – ISBN 978-5-8265-0877-0.

Содержит необходимые сведения о роли информации в современном мире. Представлены основные понятия баз данных, приведена методология их проектирования и построения моделей данных.

Предназначено для студентов, обучающихся по направлению 230200 «Информационные системы» специальности 230201 «Информационные системы и технологии».

УДК 681.518(075)
ББК 3973.26-018.3я73

ISBN 978-5-8265-0877-0

Министерство образования и науки Российской Федерации
ГОУ ВПО «Тамбовский государственный технический университет»

Ю.Ю. ГРОМОВ, О.Г. ИВАНОВА, В.Н. ТОЧКА

УПРАВЛЕНИЕ ДАННЫМИ

Допущено УМО вузов по университетскому
политехническому образованию в качестве учебного пособия
для студентов высших учебных заведений,
обучающихся по направлению 230200 «Информационные системы»
специальности 230201 «Информационные системы и технологии»

Тамбов
Издательство ТГТУ
2009

Учебное издание

ГРОМОВ Юрий Юрьевич,
ИВАНОВА Ольга Геннадьевна,
ТОЧКА Владимир Николаевич

УПРАВЛЕНИЕ ДАННЫМИ

Учебное пособие

Редактор Е.С. Кузнецова
Инженер по компьютерному макетированию М.А. Филатова

Подписано в печать 11.12.2009.
Формат 60 × 84/16. 4,65 усл. печ. л. Тираж 100 экз. Заказ № 597.

Издательско-полиграфический центр
Тамбовского государственного технического университета
392000, Тамбов, Советская, 106, к. 14

ВВЕДЕНИЕ

Функционирование систем баз данных основано на реляционной модели данных.

Реляционная модель описывает, какие данные могут храниться в реляционных базах данных, а также способы манипулирования такими данными. В упрощённом виде основная идея реляционной модели состоит в том, что данные должны храниться в таблицах и только в таблицах. Эта, кажущаяся тривиальной, идея оказывается вовсе не простой при рассмотрении вопроса, а что, собственно, представляет собой таблица? В данный момент существует много различных систем обработки данных, оперирующих понятием «таблица», например, всем известные электронные таблицы, таблицы текстового редактора MS Word и т.п. Ячейки электронной таблицы могут хранить разнотипные данные, например, числа, строки текста, формулы, ссылающиеся на другие ячейки. Собственно, на одном листе электронной таблицы можно разместить несколько совершенно независимых таблиц, если под таблицей понимать прямоугольную область, расчерченную на клеточки и заполненную данными. Таблицы текстовых редакторов вообще могут иметь совершенно произвольную структуру.

Электронные таблицы и текстовые редакторы позволяют хранить и обрабатывать данные очень гибко, но как быть, если требуется хранить информацию обо всех сотрудниках большого предприятия и периодически выдавать ответы на запросы типа «представить список всех сотрудников, принятых на работу не позднее трёх лет назад, имеющих по крайней мере одного ребенка, не имеющих взысканий и с зарплатой не выше 1000 р.» Для получения ответов на подобные запросы и предназначены *Системы Управления Базами Данных* (СУБД).

1. АРХИТЕКТУРА СИСТЕМЫ БАЗ ДАННЫХ

1.1. УРОВНИ АРХИТЕКТУРЫ СИСТЕМ БАЗ ДАННЫХ

Архитектура ANSI/SPARC включает три уровня: внутренний, концептуальный и внешний. В общих чертах они представляют собой следующее.

Внутренний уровень – это уровень, наиболее близкий к физическому хранению, т.е. связанный со способами сохранения информации на физических устройствах хранения.

Внешний уровень наиболее близок к пользователям, т.е. он связан со способами представления данных для отдельных пользователей.

Концептуальный уровень – это «промежуточный» уровень между двумя первыми.

Если внешний уровень связан с индивидуальными представлениями пользователей, то концептуальный уровень связан с обобщённым представлением пользователей. Иначе говоря, может быть несколько внешних представлений, каждое из которых состоит из более или менее абстрактного представления определённой части базы данных, и может быть только одно концептуальное представление, состоящее из абстрактного представления базы данных в целом. Также есть единственное внутреннее представление, отражающее всю базу данных как физически хранимую.

Когда называют некоторое представление абстрактным, имеется в виду, что оно включает логические конструкции, ориентированные на пользователя, такие как логические записи или поля, и не включает машинно-ориентированные конструкции, такие как биты или байты.

1.1.1. Внешний уровень

Внешний уровень – это индивидуальный уровень пользователя. Пользователь может быть прикладным программистом или конечным пользователем с любым уровнем профессиональной подготовки. Особое место среди пользователей занимает администратор базы данных. В отличие от остальных пользователей его интересует также концептуальный и внутренний уровни. У каждого пользователя есть свой язык общения.

Для прикладного программиста это либо один из распространённых языков программирования, такой как C, COBOL или PL/1, либо специальный язык рассматриваемой системы. Такие оригинальные языки называют языками четвёртого поколения на том основании, что машинный код, язык ассемблера и такие языки, как COBOL, можно считать языками трёх первых «поколений», а оригинальные языки модернизированы по сравнению с языками третьего поколения так же, как языки третьего поколения улучшены по сравнению с языком ассемблера.

Для конечного пользователя это или специальный язык запросов, или язык специального назначения, возможно, основанный на формах и меню, созданный специально с учётом требований пользователя и поддерживаемый некоторым оперативным приложением.

Все эти языки включают *подъязык данных*, т.е. подмножество операторов всего языка, связанное только с объектами и операциями баз данных. Иначе говоря, подъязык данных встроен в базовый язык, который также обеспечивает различные не связанные с базами данных возможности, такие как локальные или временные переменные, вычислительные операции, логические операции типа if-then-else и т.д. Система может поддерживать любое количество базовых языков и любое количество подъязыков данных. Однако существует один язык, который поддерживается практически всеми сегодняшними системами, – это язык SQL. Большинство систем позволяет использовать SQL и как отдельный язык запросов, и как встроенный в другие языки, такие как C и COBOL.

Хотя с точки зрения архитектуры удобно различать подъязык данных и включающий его базовый язык, на практике они могут быть неразличимы настолько, насколько это имеет отношение к пользователю. Безусловно, с точки зрения пользователя предпочтительнее, чтобы они были неразличимы. Если они неразличимы или трудноразличимы, их называют сильно связанными. Если они ясно и легко различаются, говорят, что они слабо связаны. Большинство систем на сегодняшний день поддерживает лишь слабую связь. Системы с сильной связью могли бы предоставить пользователю более унифицированный набор возможностей, но требуют больше усилий со стороны системных проектировщиков и разработчиков, однако есть основания предполагать, что на протяжении следующих нескольких лет будет происходить постепенное продвижение к более сильно связанным системам.

В принципе любой подъязык данных является на самом деле комбинацией по крайней мере двух подчинённых языков – *языка определения данных* (*data definition language – DDL*), который поддерживает определения или объявления объектов базы данных, и *языка обработки данных* (*data manipulation language – DML*), который поддерживает операции с такими объектами или их обработку. Например, рассмотрим пользователя языка PL/1. Подъязык данных для этого пользователя содержит такие возможности PL/1, которые используются для поддержки связи с СУБД.

Язык определения данных содержит такие описательные структуры PL/1, которые требуются для объявления объектов базы данных, – сам оператор DECLARE (DCL), определённые типы данных PL/1, возможные специальные дополнения для языка PL/1, чтобы поддерживать новые объекты, которые не обрабатываются существующим PL/1.

Язык обработки данных состоит из таких выполняемых операторов PL/1, которые передают информацию в и из базы данных; опять же, возможно, включая специальные новые операторы.

Отдельного пользователя интересует только некоторая часть всей базы данных; кроме того, пользовательское представление такой части будет вообще чем-то абстрактным по сравнению со способом физического хранения данных. В соответствии с терминологией ANSI/SPARC представление отдельного пользователя называется внешним представлением. Таким образом, *внешнее представление* – это содержимое базы данных, каким видит его определённый пользователь (т.е. для этого пользователя внешнее представление и есть база данных). Например, пользователь из отдела кадров может рассматривать базу данных как набор записей с информацией об отделах плюс набор записей с информацией о служащих и ничего не знать о записях с информацией о деталях и поставщиках, с которыми работают пользователи в отделе обеспечения.

В общем, внешнее представление состоит из множества экземпляров каждого типа внешней записи, которые, в свою очередь, отнюдь не обязательно должны совпадать с хранимыми записями. Находящийся в распоряжении пользователя подъязык данных определён в терминах внешних записей; например, операция выборки языка обработки данных будет проводить выборку из экземпляров внешних, а не хранимых записей.

Замечание. В данный момент предполагается, что вся информация представлена на внешнем уровне в форме записей. Некоторые системы представляют информацию иначе, например, в форме «связей» или указателей. Для систем, использующих такие альтернативные методы, все определения и пояснения этого раздела требуют соответствующих изменений. Это замечание касается также концептуального и внутреннего уровней.

Термин «логическая запись» на самом деле имеет тот же смысл, что и термин «внешняя запись».

Каждое внешнее представление определяется средствами внешней схемы, которая в основном состоит из определений каждого типа записей во внешнем представлении.

Внешняя схема написана с помощью языка определения данных из пользовательского подъязыка данных. Поэтому язык определения данных иногда называют внешним языком определения данных. Например, тип внешней записи можно определить как шестисимвольное поле с номером служащего плюс пятицифровое (десятичное) поле его зарплаты и т.д. Кроме этого, необходимо определить отображение между внешней схемой и основной концептуальной схемой. Это отображение рассматривается.

1.1.2. Концептуальный уровень

Концептуальное представление – это представление всей информации базы данных в несколько более абстрактной форме, как и в случае внешнего представления, по сравнению с физическим способом хранения данных. Однако концептуальное представление существенно отличается от способа представления данных какому-либо отдельному пользователю. Концептуальное представление – это представление данных такими, какие они есть на самом деле, а не такими, какими вынужден их видеть пользователь в рамках, например, определённого языка или используемого аппаратного обеспечения.

Концептуальное представление состоит из множества экземпляров каждого типа концептуальной записи. Например, оно может состоять из набора экземпляров записей, содержащих информацию об отделах, плюс набор экземпляров, содержащих информацию о поставщиках, плюс набор экземпляров, содержащих информацию о деталях, и т.д. Концептуальная запись необязательно должна совпадать с внешней записью, с одной стороны, и с хранимой записью – с другой.

Необходимо отметить, что могут быть и другие способы представления данных на концептуальном уровне, которые вообще не используют записей как таковых и поэтому в некотором отношении предпочтительнее. Например, вместо того чтобы рассматривать понятие концептуальной записи, можно рассматривать объекты и, возможно, отношения между ними в несколько более прямой форме.

Концептуальное представление определяется с помощью концептуальной схемы, которая включает определения каждого типа концептуальных записей. Концептуальная схема использует другой язык определения данных – концептуальный. Чтобы добиться независимости данных, нельзя включать в определения концептуального языка любое рассмотрение структуры хранения или метода доступа. Определения концептуального языка должны относиться только к содержанию информации. Это означает, что в концептуальной схеме не должно быть никакого упоминания о представлении хранимого файла, последовательности хранимых записей, индексировании, хеш-адресации, указателях или других подробностях хранения или доступа. Если концептуальная схема действительно обеспечивает независимость данных в этом смысле, то внешние схемы, определённые на основе концептуальной, заведомо будут обеспечивать независимость данных.

Концептуальное представление – это представление всего содержимого базы данных, а концептуальная схема – это определение такого представления. Однако концептуальная схема – это набор определений, больше напоминающих простые определения записей в программе на языке COBOL или каком-либо другом. Определения в концептуальной схеме могут включать определения многих дополнительных средств, таких как средства безопасности или правила для обеспечения целостности. Некоторые авторитетные специалисты предлагают в качестве конечной цели концептуальной схемы описание всего предприятия – не только самих его данных, но также и того, как эти данные используются: как они перемещаются внутри предприятия, для чего используются в каждом конкретном месте, какая ревизия или иной контроль применяется к ним в каждом отдельном случае и т.д. Ни одна сегодняшняя система реально не поддерживает такого концептуального уровня, который хотя бы немного приблизился к этой степени развитости; в большинстве существующих систем «концептуальная схема» в действительности представляет собой немного больше, чем простое объединение всех отдельных внешних схем с дополнительными средствами безопасности и правилами обеспечения целостности. Вероятно, со временем системы будут гораздо «интеллектуальнее» в поддержке концептуального уровня.

1.1.3. Внутренний уровень

Третьим уровнем архитектуры является внутренний уровень. *Внутреннее представление* — это представление нижнего уровня всей базы данных; оно состоит из многих экземпляров каждого типа внутренней записи. Термин «внутренняя запись» принадлежит терминологии ANSI/SPARC и означает конструкцию, называемую *хранимой записью*. Внутреннее представление так же, как внешнее и кон-

цептуальное, не связано с физическим уровнем, так как в нём не рассматриваются физические записи, также называемые *блоками* или *страницами*, и не рассматриваются физические области устройства хранения, такие как цилиндры и дорожки. Блоки или страницы *устройства ввода-вывода* – это количество данных, передаваемых из вторичной памяти – памяти накопителя в главную – оперативную за одно обращение. Обычно страницы имеют размер 1, 2 или 4 Кбайт. Другими словами, внутреннее представление предполагает бесконечное линейное адресное пространство; подробности того, как адресное пространство отображено на физическом устройстве хранения, очень зависят от системы и умышленно не включены в общую архитектуру.

Внутреннее представление описывается с помощью *внутренней схемы*, которая определяет не только различные типы хранимых записей, но также существующие индексы, способы представления хранимых полей, физическую последовательность хранимых записей и т.д. Внутренняя схема пишется с использованием ещё одного языка определения данных – внутреннего.

Вместо терминов «внутреннее представление» и «внутренняя схема» можно использовать более понятные термины «хранимая база данных» и «определение структуры хранения» соответственно.

В некоторых исключительных ситуациях прикладные программы, в частности те, которые называют утилитами, могут выполнять операции непосредственно на внутреннем, а не на внешнем уровне. Конечно, такой практикой пользоваться не рекомендуется; она представляет риск с точки зрения безопасности и целостности, к тому же программа будет зависеть от загруженных данных; но иногда это может быть единственным способом достичь выполнения требуемой функции или добиться необходимого быстродействия – так же как пользователю языка высокого уровня иногда по тем же причинам необходимо прибегнуть к языку ассемблера.

1.2. ОТОБРАЖЕНИЯ

Отображение *концептуальный – внутренний* определяет соответствие между концептуальным представлением и хранимой базой данных, т.е. как концептуальные записи и поля представлены на внутреннем уровне. При изменении структуры хранимой базы данных, т.е. при внесении изменений в определение структуры хранения, изменяется и отображение концептуальный – внутренний таким образом, чтобы концептуальная схема осталась неизменной. В обязанности администратора базы данных входит управление такими изменениями. Иначе говоря, чтобы сохранить независимость данных, результаты таких изменений не должны коснуться концептуального уровня.

Отображение *внешний – концептуальный* определяет соответствие между некоторым внешним представлением и концептуальным представлением. В общем, различия, которые могут существовать между этими двумя уровнями, подобны различиям между концептуальным представлением и хранимой базой данных. Например, данные полей могут быть разных типов, названия полей и записей могут быть изменены, несколько концептуальных полей могут быть объединены в одно, виртуальное, внешнее поле и т.д. В одно и то же время может существовать любое количество внешних представлений, одно и то же внешнее представление может принадлежать нескольким пользователям, разные внешние представления могут перекрываться.

Большинство систем позволяет выражать определение одного внешнего представления через другое, т.е. с помощью отображения *внешний-внешний,* не требуя обязательно явно определять отображение на концептуальный уровень. Эта возможность полезна, если несколько представлений схожи между собой. В частности, эта возможность есть во многих реляционных системах.

1.3. СИСТЕМА УПРАВЛЕНИЯ БАЗОЙ ДАННЫХ

Система управления базой данных (СУБД) представляет собой программное обеспечение, которое управляет доступом к базе данных. Это происходит следующим образом.

1. Пользователь выдаёт запрос на доступ, применяя определённый подъязык данных, обычно SQL.
2. СУБД получает этот запрос и анализирует его.
3. СУБД просматривает внешнюю схему для этого пользователя, соответствующее отображение (внешний – концептуальный), концептуальную схему, отображение концептуальный – внутренний и определение структуры хранения.

4. СУБД выполняет необходимые операции над хранимой базой данных.

В качестве примера предположим, что рассматривается выборка определённого экземпляра внешней записи. В общем случае поля будут использоваться из нескольких экземпляров концептуальных записей, которые, в свою очередь, будут запрашивать поля из нескольких экземпляров хранимых записей. СУБД должна сначала выбрать все требуемые экземпляры хранимых записей, затем построить требуемые экземпляры концептуальных записей и после этого сформировать экземпляр внешней записи. На каждом этапе могут требоваться преобразования типов данных или другие преобразования.

В данном случае предполагается, что весь процесс является интерпретируемым, т.е. анализ запроса, проверка различных схем и другие процедуры осуществляются во время выполнения. Интерпретация, в свою очередь, обычно имеет низкую производительность, поскольку на её выполнение затрачивается много времени. На практике, однако, обычно имеется возможность скомпилировать запрос доступа перед его выполнением. Реальный пример системы, которая применяет такой подход, – СУБД DB2 фирмы IBM.

Функции СУБД

Определение данных. СУБД должна допускать определения данных, т.е. внешние схемы, концептуальную схему, внутреннюю схему, а также все связанные отображения, в исходной форме и преобразовывать эти определения в форму соответствующих объектов. Иначе говоря, СУБД должна включать в себя компонент языкового процессора для различных языков определений данных. СУБД должна также понимать синтаксис языка определений данных.

Обработка данных. СУБД должна уметь обрабатывать запросы пользователя на выборку, изменение или удаление существующих данных в базе данных или на добавление новых данных в базу данных, т.е. СУБД должна включать в себя компонент процессора языка обработки данных.

Запросы языка обработки данных бывают «планируемые» и «непланируемые».

1. *Планируемый запрос* – это запрос, необходимость которого предусмотрена заранее. Администратор базы данных, возможно, должен настроить физический проект базы данных таким образом, чтобы гарантировать достаточное быстродействие для таких запросов.

2. *Непланируемый запрос* – это, наоборот, специальный запрос, необходимость которого не была предусмотрена заранее. Физический проект базы данных может подходить, а может и не подходить для рассматриваемого специального запроса. В общем, получение наибольшей возможной производительности для непланируемых запросов представляет собой одну из проблем СУБД.

Планируемые запросы характерны для операционных приложений, а непланируемые – для приложений поддержки решений. Более того, планируемые запросы обычно осуществляются из написанных заранее приложений, а непланируемые запросы по определению производятся интерактивно.

Безопасность и целостность данных. СУБД должна контролировать пользовательские запросы и пресекать попытки нарушения правил безопасности и целостности, определённые АБД.

Восстановление данных и дублирование. СУБД или другой связанный с ней программный компонент, обычно называемый *администратором транзакций*, должны осуществлять необходимый контроль над восстановлением данных и дублированием.

Словарь данных. СУБД должна обеспечить функцию *словаря данных*. Сам словарь данных можно по праву считать базой данных, но не пользовательской, а системной. Словарь содержит данные о данных (иногда называемые метаданными), т.е. определения других объектов системы, а не просто «сырые данные». В частности, исходная и объектная формы различных схем (внешних, концептуальной и т.д.) и отображений будут сохранены в словаре. Расширенный словарь будет включать также перекрёстные ссылки, показывающие, например, какие из программ какую часть базы данных используют, какие отчёты требуются тем или иным пользователям, какие терминалы подключены к системе и т.д. Словарь может и должен быть интегрирован в определяемую им базу данных, а значит, должен содержать описание самого себя. Конечно, должна быть возможность обращения к словарю, как и к другой базе данных, например, для того чтобы узнать, какие программы и/или пользователи будут затронуты при предполагаемом внесении изменения в систему.

Производительность. СУБД должна выполнять все указанные функции с максимально возможной эффективностью.

В целом назначением СУБД является предоставление *пользовательского интерфейса* с базой данных. Пользовательский интерфейс может быть определён как граница в системе, ниже которой всё невидимо для пользователя. Следовательно, по определению пользовательский интерфейс находится на внешнем уровне. Тем не менее иногда встречаются случаи, когда внешнее представление незначительно отличается от относящейся к нему части основного концептуального представления, по крайней мере, в современных коммерческих продуктах.

1.4. СИСТЕМА УПРАВЛЕНИЯ ПЕРЕДАЧЕЙ ДАННЫХ

Запросы к базе данных от конечных пользователей в действительности передаются в форме коммуникационных сообщений – от рабочей станции пользователя, которая может быть физически удалена от самой системы, к некоторому оперативному приложению, встроенному или нет, а от него к СУБД. Более того, ответы пользователю от СУБД и оперативного приложения к станции пользователя также передаются в форме таких сообщений. Передача подобных сообщений происходит под управлением другого программного компонента – диспетчера передачи данных.

Диспетчер передачи данных не является частью СУБД, а представляет собой автономную систему со своими собственными правами. Однако, поскольку от диспетчера передачи данных и СУБД требуется согласованная совместная работа, они иногда рассматриваются как равноправные партнёры высокого уровня, называемого системой базы данных и передачи данных; в этой системе после обработки базой данных и диспетчером передачи данных СУБД просматривает все сообщения используемого ею приложения.

1.5. АРХИТЕКТУРА КЛИЕНТ/СЕРВЕР

Общая цель систем баз данных – поддержка разработки и выполнения приложений баз данных. Поэтому на высоком уровне систему баз данных можно рассматривать как систему с очень простой структурой, состоящей из двух частей – сервера, или машины базы данных, и набора клиентов, или внешнего интерфейса.

Сервер – это СУБД. Он поддерживает все основные функции СУБД, а именно: определение данных, обработку данных, защиту и целостность данных и т.д. В частности, он предоставляет полную поддержку на внешнем, концептуальном и внутреннем уровнях. Поэтому термин сервер в этом контексте – это просто другое имя СУБД.

Клиенты – это различные приложения, которые выполняются над СУБД: приложения, написанные пользователями, и встроенные приложения, предоставляемые поставщиками СУБД или некоторыми сторонними поставщиками программного обеспечения. С точки зрения пользователей, нет разницы между встроенными приложениями и приложениями, написанными пользователем, – все они используют один и тот же интерфейс сервера, а именно интерфейс внешнего уровня.

Исключениями являются специальные «служебные» приложения. Как упоминалось выше, такие приложения иногда могут работать только непосредственно на внутреннем уровне системы. Такие утилиты скорее относятся к непосредственным компонентам СУБД, чем к приложениям в обычном смысле.

Приложения, в свою очередь, делятся на несколько чётко определённых категорий.

1. *Приложения, написанные пользователями.* Это в основном профессиональные прикладные программы, написанные обычно либо на общепринятом языке программирования, таком как C или COBOL, либо на некоторых оригинальных языках, таких как FOCUS, хотя в обоих случаях эти языки должны как-то связываться с соответствующим подъязыком данных, как указывалось выше в этой главе.

2. *Приложения, предоставляемые поставщиками,* часто называемые инструментальными средствами. В целом назначение таких средств – содействовать в процессе создания и выполнения других приложений, т.е. приложений, которые делаются специально для некоторой специфической задачи. Эта категория инструментальных средств позволяет пользователям, особенно конечным, создавать приложения без написания традиционных программ. Например, одно из предоставляемых поставщиками инст-

рументальных средств может быть процессором языка запросов, с помощью которого конечный пользователь может выдавать незапланированные запросы к системе. Каждый такой запрос является, по существу, не чем иным, как маленьким специальным приложением, предназначенным для выполнения некоторых специфических функций.

Поставляемые инструментальные средства, в свою очередь, делятся на несколько самостоятельных классов:

- процессоры языков запросов;
- генераторы отчётов;
- графические бизнес-подсистемы;
- электронные таблицы;
- процессоры обычных языков;
- средства управления копированием;
- генераторы приложений;
- другие средства разработки приложений, включая CASE-продукты (CASE, или Computer-Aided Software Engineering – автоматизация разработки программного обеспечения), и т.д.

Главная задача системы баз данных – это поддержка создания и выполнения приложений, поэтому качество имеющихся клиентных инструментальных средств должно быть главным фактором при выборе базы данных. СУБД сама по себе не единственный и необязательно важнейший фактор, который нужно учитывать.

Так как система в целом может быть чётко разделена на две части – сервер и клиенты, появляется возможность работы этих двух частей на разных машинах. Иначе говоря, существует возможность распределённой обработки. Распределённая обработка предполагает, что отдельные машины можно соединить какой-нибудь коммуникационной сетью таким способом, что определённая задача, обрабатывающая данные, может быть распределена на нескольких машинах в сети.

1.6. РАСПРЕДЕЛЁННАЯ ОБРАБОТКА

Термин «распределённая обработка» означает, что разные машины можно соединить в коммуникационную сеть так, что одна задача обработки данных распределяется на несколько машин в сети. Связь между различными машинами осуществляется с помощью специального программного обеспечения для управления сетью.

Распределённая обработка может быть самой разнообразной и осуществляться на разных уровнях. Как отмечалось выше, в одном из простых случаев запускается сервер СУБД на одной машине и клиентское приложение на другой.

Термин «клиент/сервер» фактически стал синонимом структуры, в соответствии с которой клиент и сервер запускаются на разных машинах. В действительности существует множество аргументов в пользу такой схемы.

Первый аргумент связан с параллельной обработкой, а именно: в этом случае для всей задачи применяется несколько процессоров и обработка сервера – базы данных и клиента – приложения осуществляется параллельно. Поэтому время ответа и производительное время должны уменьшиться.

Машина сервера может быть изготовлена по специальному заказу, приспособлена для работы с СУБД и может обеспечить лучшую производительность СУБД.

Машина клиента может быть персональной станцией, приспособленной к потребностям конечного пользователя, и поэтому обеспечивать лучший интерфейс, полное соответствие требованиям, быструю реакцию и в целом дополнительные удобства при использовании.

Несколько разных машин клиентов могут иметь доступ к одной и той же машине сервера. Поэтому одна база данных может совместно использоваться несколькими отдельными клиентными системами.

Имеется ещё одно преимущество выполнения сервера и клиента на отдельных машинах – соответствие практической работе многих предприятий. Это распространённый способ для отдельного предприятия; например, банк работает со многими компьютерами, сохраняющими данные для одной части предприятия на одном компьютере, а данные для другой части – на другом. Это также очень распространено среди пользователей, которым необходим, по крайней мере иногда, доступ с одного компьютера к данным, хранимым на другом компьютере. Следуя примеру банка, можно сказать, что весьма вероятно, пользователям одного отделения банка будет иногда необходим доступ к данным, сохраняемым в другом отделении. Следовательно, машины клиентов могут иметь свои собственные сохраняемые

данные, а машина сервера может иметь свои собственные приложения. Поэтому каждая машина будет выступать в роли сервера для одних пользователей и в роли клиента для других, иными словами, каждая машина будет поддерживать полную систему баз данных.

Последнее преимущество состоит в том, что отдельная машина клиента может иметь доступ к нескольким разным машинам серверов. Это полезная возможность, поскольку, как уже упоминалось, предприятие обычно выполняет обработку данных таким образом, что полный набор всех данных не сохраняется на одной машине, а распределяется на отдельных машинах, а для приложений иногда необходим доступ к данным нескольких машин. Такой доступ в основном предоставляется двумя способами.

1. Клиент может получать доступ к любому количеству серверов, но лишь к одному в одно и то же время, т.е. каждый запрос к базе данных должен быть направлен только к одному серверу. В такой системе невозможно за один запрос получить комбинированные данные двух или более серверов. Кроме того, пользователь в такой системе должен знать, на какой именно машине какая часть данных содержится.

2. Клиент может получать доступ к любому количеству серверов одновременно, т.е. за один запрос можно получить комбинированные данные двух или более серверов. В этом случае серверы рассматриваются клиентом как один, с логической точки зрения, и пользователь может не знать, на какой именно машине какая часть данных содержится.

Второй случай – это пример системы, которую обычно называют распределённой *системой баз данных*. Полная поддержка для распределённых баз данных означает, что отдельное приложение может обрабатывать данные, распределённые на множестве различных баз данных, управление которыми осуществляют разные СУБД, работающие на многочисленных машинах с различными операционными системами, соединённых вместе коммуникационными сетями. Это означает, что приложение выполняет обработку данных с логической точки зрения, как будто управление данными полностью осуществляется одной СУБД, работающей на отдельной машине. Такая возможность может показаться невероятно трудной задачей, но весьма желаемой с практической точки зрения.

ВОПРОСЫ ДЛЯ САМОПРОВЕРКИ

1. Что такое архитектура ANSI/SPARC?
2. Охарактеризуйте внешний уровень архитектуры ANSI/SPARC.
3. Охарактеризуйте внутренний уровень архитектуры ANSI/ SPARC.
4. Охарактеризуйте концептуальный уровень архитектуры ANSI/ SPARC.
5. Что такое подъязык данных?
6. Функции СУБД.

2. БАЗОВЫЕ ПОНЯТИЯ РЕЛЯЦИОННОЙ МОДЕЛИ ДАННЫХ

2.1. ОБЩАЯ ХАРАКТЕРИСТИКА РЕЛЯЦИОННОЙ МОДЕЛИ ДАННЫХ

Основы реляционной модели данных были впервые изложены в статье Е. Кодда в 1970 г. Эта работа послужила стимулом для большого количества статей и книг, в которых реляционная модель получила дальнейшее развитие. Наиболее распространённая трактовка реляционной модели данных принадлежит К. Дейту. Согласно Дейту, реляционная модель состоит из трёх частей.

1. Структурная часть.
2. Целостная часть.
3. Манипуляционная часть.

Структурная часть описывает, какие объекты рассматриваются реляционной моделью. Постулируется, что единственной структурой данных, используемой в реляционной модели, являются нормализованные n-арные отношения.

Целостная часть описывает ограничения специального вида, которые должны выполняться для любых отношений в любых реляционных базах данных. Это целостность сущностей и целостность внешних ключей.

Манипуляционная часть описывает два эквивалентных способа манипулирования реляционными данными – реляционную алгебру и реляционное исчисление.

В данной лекции рассматривается структурная часть реляционной модели.

2.1.1. Типы данных

Любые данные, используемые в программировании, имеют свои типы данных.

Реляционная модель требует, чтобы типы используемых данных были простыми.

Для уточнения этого утверждения рассмотрим, какие вообще типы данных обычно рассматриваются в программировании. Как правило, типы данных делятся на три группы.

1. Простые типы данных.
2. Структурированные типы данных.
3. Ссылочные типы данных.

2.1.2. Простые типы данных

Простые, или атомарные, типы данных не обладают внутренней структурой. Данные такого типа называют *скалярами*. К простым типам данных относятся следующие:

- логический;
- строковый;
- численный.

Различные языки программирования могут расширять и уточнять этот список, добавляя такие типы, как:

- целый;
- вещественный;
- дата;
- время;
- денежный;
- перечислимый;
- интервальный;
- и т.д.

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

2.1.3. Структурированные типы данных

Структурированные типы данных предназначены для задания сложных структур данных. Структурированные типы данных конструируются из составляющих элементов, называемых компонентами, которые, в свою очередь, могут обладать структурой. В качестве структурированных типов данных можно привести следующие:

- массивы;
- записи (структуры).

С математической точки зрения массив представляет собой функцию с конечной областью определения. Например, рассмотрим конечное множество натуральных чисел $A = \{1, 2, \ldots, n\}$, называемое множеством индексов. Отображение $F: A \to R$ из множества A во множество вещественных чисел R задаёт одномерный вещественный массив. Значение этой функции для некоторого значения индекса I называется элементом массива, соответствующим i. Аналогично можно задавать многомерные массивы.

Запись (или структура) представляет собой кортеж из некоторого декартового произведения множеств. Действительно, запись представляет собой именованный упорядоченный набор элементов r_i, каждый из которых принадлежит типу T_i. Таким образом, запись $r = (r_1, r_2, ..., r_n)$ есть элемент множества $T = T_1 \times T_2 \times ... \times T_n$. Объявляя новые типы записей на основе уже имеющихся, пользователь может конструировать сколь угодно сложные типы данных.

Общим для структурированных типов данных является то, что они *имеют внутреннюю структуру*, используемую *на том же уровне абстракции*, что и сами типы данных.

Поясним это следующим образом. При работе с массивами или записями можно манипулировать массивом или записью как с единым целым (создавать, удалять, копировать целые массивы или записи), так и поэлементно. Для структурированных типов данных есть специальные функции – конструкторы типов, позволяющие создавать массивы или записи из элементов более простых типов.

Работая же с простыми типами данных, например с числовыми, мы манипулируем ими как неделимыми целыми объектами. Чтобы «увидеть», что числовой тип данных на самом деле сложен (является набором битов), нужно перейти на более низкий уровень абстракции. На уровне программного кода это будет выглядеть как ассемблерные вставки в код на языке высокого уровня или использование специальных побитных операций.

2.2. ССЫЛОЧНЫЕ ТИПЫ ДАННЫХ

Ссылочный тип данных (указатели) предназначен для обеспечения возможности указания на другие данные. Указатели характерны для языков процедурного типа, в которых есть понятие области памяти для хранения данных. Ссылочный тип данных предназначен для обработки сложных изменяющихся структур, например деревьев, графов, рекурсивных структур.

2.2.1. Типы данных, используемые в реляционной модели

Для реляционной модели данных тип используемых данных не важен. Требование, чтобы тип данных был *простым*, нужно понимать так, что *в реляционных операциях не должна учитываться внутренняя структура данных*. Конечно, должны быть описаны действия, которые можно производить с данными как с единым целым, например, данные числового типа можно складывать, для строк возможна операция конкатенации и т.д.

С этой точки зрения, если рассматривать массив, например, как единое целое и не использовать поэлементных операций, то массив можно считать простым типом данных. Более того, можно создать свой, сколь угодно сложный тип данных, описать возможные действия с ним, и если в операциях не требуется знание внутренней структуры данных, то такой тип данных также будет простым с точки зрения реляционной теории. Например, можно создать новый тип – комплексные числа как запись вида $z = (x, y)$, где $x \in R$, $y \in R$. Можно описать функции сложения, умножения, вычитания и деления и все действия с компонентами x и y выполнять только внутри этих операций. Тогда, если в действиях с этим типом использовать только описанные операции, то внутренняя структура не играет роли, и тип данных извне выглядит как атомарный.

Именно так в некоторых постреляционных СУБД реализована работа со сколь угодно сложными типами данных, создаваемых пользователями.

2.2.2. Домены

В реляционной модели данных с понятием тип данных тесно связано понятие домена, которое можно считать уточнением типа данных.

Домен – это семантическое понятие. Домен можно рассматривать как подмножество значений некоторого типа данных, имеющих определённый смысл. Домен характеризуется следующими свойствами:

- имеет уникальное имя (в пределах базы данных);
- определён на некотором простом типе данных или на другом домене;
- может иметь некоторое логическое условие, позволяющее описать подмножество данных, допустимых для данного домена;

– несёт определённую смысловую нагрузку.

Например, домен *D*, имеющий смысл «возраст сотрудника», можно описать как следующее подмножество множества натуральных чисел:

$$D = \{n \in N : n \geq 18 \text{ and } n \leq 60\}.$$

Если тип данных можно считать множеством всех возможных значений данного типа, то домен напоминает подмножество в этом множестве.

Отличие домена от понятия подмножества состоит именно в том, что *домен отражает семантику*, определённую предметной областью. Может быть несколько доменов, совпадающих как подмножества, но несущих различный смысл. Например, домены «Вес детали» и «Имеющееся количество» можно одинаково описать как множество неотрицательных целых чисел, но смысл этих доменов будет различным, и это будут различные домены.

Основное значение доменов состоит в том, что *домены ограничивают сравнения*. Некорректно, с логической точки зрения, сравнивать значения из различных доменов, даже если они имеют одинаковый тип. В этом проявляется смысловое ограничение доменов. Синтаксически правильный запрос «выдать список всех деталей, у которых вес детали больше имеющегося количества» не соответствует смыслу понятий «количество» и «вес».

Замечание. Понятие домена помогает правильно моделировать предметную область. При работе с реальной системой в принципе возможна ситуация, когда требуется ответить на запрос, приведённый выше. Система даст ответ, но, вероятно, он будет бессмысленным.

Замечание. Не все домены обладают логическим условием, ограничивающим возможные значения домена. В таком случае множество возможных значений домена совпадает с множеством возможных значений типа данных.

Замечание. Не всегда очевидно, как задать логическое условие, ограничивающее возможные значения домена. Автор будет благодарен тому, кто приведёт ему условие на строковый тип данных, задающий домен «Фамилия сотрудника». Ясно, что строки, являющиеся фамилиями, не должны начинаться с цифр, служебных символов, с мягкого знака и т.д. Но вот является ли допустимой фамилия «Ггггггыыыыы»? Почему бы нет? Очевидно, нет! А может, кто-то назло так себя назовёт. Трудности такого рода возникают потому, что смысл реальных явлений далеко не всегда можно формально описать. Просто мы интуитивно понимаем, что такое фамилия, но никто не может дать такое формальное определение, которое отличало бы фамилии от строк, фамилиями не являющихся. Выход из этой ситуации простой – положиться на разум сотрудника, вводящего фамилии в компьютер.

2.2.3. Отношения, атрибуты, кортежи отношения. Определения и примеры

Фундаментальным понятием реляционной модели данных является понятие *отношения*.

Определение 1. *Атрибут отношения* есть пара вида <Имя_атрибута : Имя_домена>.

Имена атрибутов должны быть уникальны в пределах отношения. Часто имена атрибутов отношения совпадают с именами соответствующих доменов.

Определение 2. *Отношение R*, определённое на множестве доменов D_1, D_2, …, D_n (необязательно различных), содержит две части: заголовок и тело.

Заголовок отношения содержит фиксированное количество атрибутов отношения: $(< A_1 : D_1 >, < A_2 : D_2 >, ..., < A_n : D_n >)$.

Тело отношения содержит множество кортежей отношения. Каждый *кортеж отношения* представляет собой множество пар вида <Имя_атрибута : Значение_атрибута>:

$(< A_1 : Val_1 >, < A_2 : Val_2 >, ..., < A_n : Val_n >)$ таких, что значение Val_i атрибута A_i принадлежит домену D_i.

Отношение обычно записывается в виде:

$$R(< A_1 : D_1 >, < A_2 : D_2 >, ..., < A_n : D_n >),$$

или короче: $R(A_1, A_2, ..., A_n)$, или просто *R*.

Число атрибутов в отношении называют *степенью* (или *-арностью*) отношения.

Мощность множества кортежей отношения называют *мощностью* отношения.

Возвращаясь к математическому понятию отношения, введённому в предыдущей главе, можно сделать следующие выводы.

Вывод 1. Заголовок отношения описывает декартово произведение доменов, на котором задано отношение. Заголовок статичен, он не меняется во время работы с базой данных. Если в отношении изменены, добавлены или удалены атрибуты, то в результате получим уже другое отношение (пусть даже с прежним именем).

Вывод 2. Тело отношения представляет собой набор кортежей, т.е. подмножество декартового произведения доменов. Таким образом, тело отношения собственно и является отношением в математическом смысле слова. Тело отношения может изменяться во время работы с базой данных – кортежи могут изменяться, добавляться и удаляться.

Пример 1. Рассмотрим отношение «Сотрудники», заданное на доменах «Номер_сотрудника», «Фамилия», «Зарплата», «Номер_отдела». Так как все домены различны, то имена атрибутов отношения удобно назвать так же, как и соответствующие домены. Заголовок отношения имеет вид: Сотрудники (Номер_сотрудника, Фамилия, Зарплата, Номер_отдела).

Пусть в данный момент отношение содержит три кортежа:

(1, Иванов, 1000, 1)

(2, Петров, 2000, 2)

(3, Сидоров, 3000, 1)

такое отношение естественным образом представляется в виде таблицы:

Отношение «Сотрудники»

Номер_сотрудника	Фамилия	Зарплата	Номер_отдела
1	Иванов	1000	1
2	Петров	2000	2
3	Сидоров	3000	1

Определение 3. *Реляционной базой данных* называется набор отношений.

Определение 4. *Схемой реляционной базы* данных называется набор заголовков отношений, входящих в базу данных.

Хотя любое отношение можно изобразить в виде таблицы, нужно чётко понимать, что *отношения не являются таблицами*. Это близкие, но не совпадающие понятия. Различия между отношениями и таблицами будут рассмотрены ниже.

Термины, которыми оперирует реляционная модель данных, имеют соответствующие «табличные» синонимы:

Реляционный термин	Соответствующий «табличный» термин
База данных	Набор таблиц
Схема базы данных	Набор заголовков таблиц
Отношение	Таблица
Заголовок отношения	Заголовок таблицы
Тело отношения	Тело таблицы
Атрибут отношения	Наименование столбца таблицы
Кортеж отношения	Строка таблицы
Степень (-арность) отношения	Количество столбцов таблицы
Мощность отношения	Количество строк таблицы
Домены и типы данных	Типы данных в ячейках таблицы

Свойства отношений. Свойства отношений непосредственно следуют из приведённого выше определения отношения. В этих свойствах в основном и состоят различия между отношениями и таблицами.

1. *В отношении нет одинаковых кортежей.* Действительно, тело отношения есть *множество* кортежей и, как всякое множество, не может содержать неразличимые элементы (см. понятие множества в гл. 1.). Таблицы в отличие от отношений могут содержать одинаковые строки.

2. *Кортежи не упорядочены* (*сверху вниз*). Действительно, несмотря на то что мы изобразили отношение «Сотрудники» в виде таблицы, нельзя сказать, что сотрудник Иванов «предшествует» сотруднику Петрову. Причина та же – тело отношения есть множество, а множество не упорядочено. Это вторая причина, по которой нельзя отождествить отношения и таблицы – строки в таблицах упорядочены. Одно и то же отношение может быть изображено разными таблицами, в которых строки идут в различном порядке.

3. *Атрибуты не упорядочены* (*слева направо*). Так как каждый атрибут имеет уникальное имя в пределах отношения, то порядок атрибутов не имеет значения. Это свойство несколько отличает отношение от математического определения отношения (см. гл. 1 – компоненты кортежей там упорядочены). Это также третья причина, по которой нельзя отождествить отношения и таблицы – столбцы в таблице упорядочены. Одно и то же отношение может быть изображено разными таблицами, в которых столбцы идут в различном порядке.

4. *Все значения атрибутов атомарны.* Это следует из того, что лежащие в их основе атрибуты имеют атомарные значения. Это четвёртое отличие отношений от таблиц – в ячейки таблиц можно поместить что угодно – массивы, структуры и даже другие таблицы.

Замечание. Из свойств отношения следует, что не каждая таблица может задавать отношение. Для того чтобы некоторая таблица задавала отношение, необходимо, чтобы таблица имела простую структуру (содержала бы только строки и столбцы, причём в каждой строке было бы одинаковое количество полей), в таблице не должно быть одинаковых строк, любой столбец таблицы должен содержать данные только одного типа, все используемые типы данных должны быть простыми.

Замечание. Каждое отношение можно считать *классом эквивалентности таблиц*, для которых выполняются следующие условия:

- таблицы имеют одинаковое количество столбцов;
- таблицы содержат столбцы с одинаковыми наименованиями;
- столбцы с одинаковыми наименованиями содержат данные из одних и тех же доменов;
- таблицы имеют одинаковые строки с учётом того, что порядок столбцов может различаться.

Все такие таблицы есть различные изображения одного и того же отношения.

2.2.4. Первая нормальная форма

Труднее всего дать определение вещей, которые всем понятны. Если давать нестрогое, описательное определение, то всегда остаётся возможность неправильной его трактовки. Если дать строгое, формальное определение, то оно, как правило, или тривиально, или слишком громоздко. Именно такая ситуация с определением отношения в *Первой Нормальной Форме* (*1НФ*). Совсем не говорить об этом нельзя, так как на основе 1НФ строятся более высокие нормальные формы, которые рассматриваются далее в гл. 3. Дать определение 1НФ сложно ввиду его тривиальности. Поэтому дадим просто несколько объяснений.

Объяснение 1. Говорят, что отношение *R* находится в 1НФ, если оно удовлетворяет определению 2.

Это, собственно, тавтология, ведь из определения 2 следует, что других отношений не бывает. Действительно, определение 2 описывает, что является отношением, а что – нет, следовательно, отношений в непервой нормальной форме просто нет.

Объяснение 2. Говорят, что отношение *R* находится в 1НФ, если его атрибуты содержат только скалярные (атомарные) значения.

Опять же, определение 2 опирается на понятие домена, а домены определены на простых типах данных.

Непервую нормальную форму можно получить, если допустить, что атрибуты отношения могут быть определены на сложных типах данных – массивах, структурах или даже на других отношениях. Легко себе представить таблицу, у которой в некоторых ячейках содержатся массивы, в других ячейках – определённые пользователями сложные структуры, а в третьих ячейках – целые реляционные таблицы, которые, в свою очередь, могут содержать такие же сложные объекты. Именно такие возможности предоставляются некоторыми современными пост-реляционными и объектными СУБД.

Требование, что отношения должны содержать только данные простых типов, объясняет, почему отношения иногда называют *плоскими таблицами* (*plain table*). Действительно, таблицы, задающие отношения, двумерны. Одно измерение задаётся списком столбцов, второе – списком строк. Пара координат (Номер строки, Номер столбца) однозначно идентифицирует ячейку таблицы и содержащееся в ней значение. Если же допустить, что в ячейке таблицы могут содержаться данные сложных типов (массивы, структуры, другие таблицы), то такая таблица будет уже не плоской. Например, если в ячейке таблицы содержится массив, то для обращения к элементу массива нужно знать три параметра (Номер строки, Номер столбца, Номер элемента в массиве).

Таким образом, появляется третье объяснение Первой Нормальной Формы:

Объяснение 3. Отношение R находится в 1НФ, если оно является плоской таблицей.

Мы сознательно ограничиваемся рассмотрением только классической реляционной теории, в которой все отношения имеют только атомарные атрибуты и заведомо находятся в 1НФ.

ВОПРОСЫ ДЛЯ САМОПРОВЕРКИ

1. Реляционные базы данных.
2. Схема реляционной базы данных.
3. Первая Нормальная Форма (1НФ)

3. НОРМАЛЬНЫЕ ФОРМЫ ОТНОШЕНИЙ

3.1. ЭТАПЫ РАЗРАБОТКИ БАЗЫ ДАННЫХ

Целью разработки любой базы данных является хранение и использование информации о какой-либо предметной области. Для реализации этой цели имеются следующие инструменты.

1. Реляционная модель данных – удобный способ представления данных предметной области.
2. Язык SQL – универсальный способ манипулирования такими данными.

Однако очевидно, что для одной и той же предметной области реляционные отношения можно спроектировать множеством различных способов. Например, можно спроектировать несколько отношений с большим количеством атрибутов или, наоборот, разнести все атрибуты по большому числу мелких отношений. Как определить, по каким признакам нужно помещать атрибуты в те или иные отношения?

В данной лекции рассматриваются способы «хорошего», или «правильного», проектирования реляционных отношений. Сначала мы обсудим, что значит «хорошие», или «правильные» модели данных. Потом будут введены понятия первой, второй и третьей нормальных форм отношений (1НФ, 2НФ, 3НФ) и показано, что «хорошими» являются отношения в третьей нормальной форме.

При разработке базы данных обычно выделяется несколько уровней моделирования, при помощи которых происходит переход от предметной области к конкретной реализации базы данных средствами конкретной СУБД. Можно выделить следующие уровни:

- сама предметная область;
- модель предметной области;
- логическая модель данных;
- физическая модель данных;
- собственно база данных и приложения.

Предметная область – это часть реального мира, данные о которой мы хотим отразить в базе данных. Например, в качестве предметной области можно выбрать бухгалтерию какого-либо предприятия, отдел кадров, банк, магазин и т.д. Предметная область бесконечна и содержит как существенно важные понятия и данные, так и малозначащие или вообще незначащие данные. Так, если в качестве предмет-

ной области выбрать учёт товаров на складе, то понятия «накладная» и «счёт-фактура» являются существенно важными понятиями, а то, что сотрудница, принимающая накладные, имеет двоих детей, – это для учёта товаров неважно. Однако с точки зрения отдела кадров данные о наличии детей являются существенно важными. Таким образом, важность данных зависит от выбора предметной области.

Модель предметной области. Модель предметной области – это наши знания о предметной области. Знания могут быть как в виде неформальных знаний в мозгу эксперта, так и выражены формально при помощи каких-либо средств. В качестве таких средств могут выступать текстовые описания предметной области, наборы должностных инструкций, правила ведения дел в компании и т.п. Опыт показывает, что текстовый способ представления модели предметной области крайне неэффективен. Гораздо более информативными и полезными при разработке баз данных являются описания предметной области, выполненные при помощи специализированных графических нотаций. Имеется большое количество методик описания предметной области. Из наиболее известных можно назвать методику структурного анализа SADT и основанную на нём IDEF0, диаграммы потоков данных Гейна-Сарсона, методику объектно-ориентированного анализа UML и др. Модель предметной области описывает скорее процессы, происходящие в предметной области, и данные, используемые этими процессами. От того, насколько правильно смоделирована предметная область, зависит успех дальнейшей разработки приложений.

Логическая модель данных. На следующем, более низком уровне находится логическая модель данных предметной области. Логическая модель описывает понятия предметной области, их взаимосвязь, а также ограничения на данные, налагаемые предметной областью. Примеры понятий – «сотрудник», «отдел», «проект», «зарплата». Примеры взаимосвязей между понятиями – «сотрудник числится ровно в одном отделе», «сотрудник может выполнять несколько проектов», «над одним проектом может работать несколько сотрудников». Примеры ограничений – «возраст сотрудника не менее 16 и не более 60 лет».

Логическая модель данных является начальным прототипом будущей базы данных. Логическая модель строится в терминах информационных единиц, но без привязки к конкретной СУБД. Более того, логическая модель данных необязательно должна быть выражена средствами именно реляционной модели данных. Основным средством разработки логической модели данных в настоящий момент являются различные варианты *ER-диаграмм* (*Entity-Relationship*, *диаграммы сущность-связь*). Одну и ту же ER-модель можно преобразовать как в реляционную модель данных, так и в модель данных для иерархических и сетевых СУБД, или в постреляционную модель данных. Однако, так как мы рассматриваем именно реляционные СУБД, можно считать, что логическая модель данных для нас формулируется в терминах реляционной модели данных.

Решения, принятые на предыдущем уровне, при разработке модели предметной области определяют некоторые границы, в пределах которых можно развивать логическую модель данных, в пределах же этих границ можно принимать различные решения. Например, модель предметной области складского учёта содержит понятия «склад», «накладная», «товар». При разработке соответствующей реляционной модели эти термины обязательно должны быть использованы, но различных способов реализации тут много – можно создать одно отношение, в котором будут присутствовать в качестве атрибутов «склад», «накладная», «товар», а можно создать три отдельных отношения, по одному на каждое понятие.

При разработке логической модели данных возникают вопросы: хорошо ли спроектированы отношения? Правильно ли они отражают модель предметной области, а следовательно и саму предметную область?

Физическая модель данных. На ещё более низком уровне находится физическая модель данных. Физическая модель данных описывает данные средствами конкретной СУБД. Мы будем считать, что физическая модель данных реализована средствами именно реляционной СУБД, хотя, как уже сказано выше, это необязательно. Отношения, разработанные на стадии формирования логической модели данных, преобразуются в таблицы, атрибуты становятся столбцами таблиц, для ключевых атрибутов создаются уникальные индексы, домены преображаются в типы данных, принятые в конкретной СУБД.

Ограничения, имеющиеся в логической модели данных, реализуются различными средствами СУБД, например, при помощи индексов, декларативных ограничений целостности, триггеров, хранимых процедур. При этом решения, принятые на уровне логического моделирования, определяют некоторые границы, в пределах которых можно развивать физическую модель данных. Точно так же в пределах этих границ можно принимать различные решения. Например, отношения, содержащиеся в логической модели данных, должны быть преобразованы в таблицы, но для каждой таблицы можно допол-

нительно объявить различные индексы, повышающие скорость обращения к данным. Многое тут зависит от конкретной СУБД.

При разработке физической модели данных возникают вопросы: хорошо ли спроектированы таблицы? Правильно ли выбраны индексы? Насколько много программного кода в виде триггеров и хранимых процедур необходимо разработать для поддержания целостности данных?

Собственно база данных и приложения. И наконец, как результат предыдущих этапов появляется собственно сама база данных. База данных реализована на конкретной программно-аппаратной основе, и выбор этой основы позволяет существенно повысить скорость работы с базой данных. Например, можно выбирать различные типы компьютеров, менять количество процессоров, объём оперативной памяти, дисковые подсистемы и т.п. Очень большое значение имеет также настройка СУБД в пределах выбранной программно-аппаратной платформы.

Но опять решения, принятые на предыдущем уровне – уровне физического проектирования, определяют границы, в пределах которых можно принимать решения по выбору программно-аппаратной платформы и настройки СУБД.

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

3.2. АДЕКВАТНОСТЬ БАЗЫ ДАННЫХ ПРЕДМЕТНОЙ ОБЛАСТИ

База данных должна адекватно отражать предметную область. Это означает, что должны выполняться следующие условия.

1. Состояние базы данных в каждый момент времени должно соответствовать состоянию предметной области.
2. Изменение состояния предметной области должно приводить к соответствующему изменению состояния базы данных.
3. Ограничения предметной области, отражённые в модели предметной области, должны некоторым образом отражаться и учитываться в базе данных.

Лёгкость разработки и сопровождения базы данных. Практически любая база данных, за исключением совершенно элементарных, содержит некоторое количество программного кода в виде триггеров и хранимых процедур.

Хранимые процедуры – это процедуры и функции, которые хранятся непосредственно в базе данных в откомпилированном виде и которые могут запускаться пользователями или приложениями, работающими с базой данных. Хранимые процедуры обычно пишутся либо на специальном процедурном расширении языка SQL (например, PL/SQL для ORACLE или Transact-SQL для MS SQL Server), или на некотором универсальном языке программирования, например, C++, с включением в код операторов SQL в соответствии со специальными правилами такого включения. Основное назначение хранимых процедур – реализация бизнес-процессов предметной области.

Триггеры – это хранимые процедуры, связанные с некоторыми событиями, происходящими во время работы базы данных. В качестве таких событий выступают операции вставки, обновления и удаления строк таблиц. Если в базе данных определён некоторый триггер, то он запускается автоматически всегда при возникновении события, с которым этот триггер связан. Очень важным является то, что пользователь не может обойти триггер. Триггер срабатывает независимо от того, кто из пользователей и каким способом инициировал событие, вызвавшее запуск триггера. Таким образом, основное назначение триггеров – автоматическая поддержка целостности базы данных. Триггеры могут быть как достаточно простыми, например, поддерживающими ссылочную целостность, так и довольно сложными, реализующими какие-либо сложные ограничения предметной области или сложные действия, которые должны произойти при наступлении некоторых событий. Например, с операцией вставки нового товара в накладную может быть связан триггер, который выполняет следующие действия – проверяет, есть ли необходимое количество товара, при наличии товара добавляет его в накладную и уменьшает данные о наличии товара на складе, при отсутствии товара формирует заказ на поставку недостающего товара и тут же посылает заказ по электронной почте поставщику.

Очевидно, что, чем больше программного кода в виде триггеров и хранимых процедур содержит база данных, тем сложнее её разработка и дальнейшее сопровождение.

***Скорость операций обновления данных* (*вставка, обновление, удаление*).** На уровне логического моделирования мы определяем реляционные отношения и атрибуты этих отношений. На этом уровне мы не можем определять какие-либо физические структуры хранения (индексы, хеширование и т.п.). Единственное, чем мы можем управлять – это распределением атрибутов по различным отношениям. Можно описать мало отношений с большим количеством атрибутов или много отношений, каждое из которых содержит мало атрибутов. Таким образом, необходимо попытаться ответить на вопрос – влияет ли количество отношений и количество атрибутов в отношениях на скорость выполнения операций обновления данных. Такой вопрос, конечно, не является достаточно корректным, так как скорость выполнения операций с базой данных сильно зависит от физической реализации базы данных. Тем не менее попытаемся качественно оценить это влияние при одинаковых подходах к физическому моделированию.

Основными операциями, изменяющими состояние базы данных, являются операции вставки, обновления и удаления записей. В базах данных, требующих постоянных изменений (складской учёт, системы продаж билетов и т.п.), производительность определяется скоростью выполнения большого количества небольших операций вставки, обновления и удаления.

Рассмотрим операцию вставки записи в таблицу. Вставка записи производится в одну из свободных страниц памяти, выделенных для данной таблицы. СУБД постоянно хранит информацию о наличии и расположении свободных страниц. Если для таблицы не созданы индексы, то операция вставки выполняется фактически с одинаковой скоростью независимо от размера таблицы и от количества атрибутов в ней. Если в таблице имеются индексы, то при выполнении операции вставки записи индексы должны быть перестроены. Таким образом, скорость выполнения операции вставки уменьшается при увеличении количества индексов у таблицы и мало зависит от числа строк в таблице.

Рассмотрим операции обновления и удаления записей из таблицы. Прежде чем обновить или удалить запись, её необходимо найти. Если таблица не индексирована, то единственным способом поиска является последовательное сканирование таблицы в поиске нужной записи. В этом случае скорость операций обновления и удаления существенно повышается с увеличением количества записей в таблице и не зависит от количества атрибутов. Но на самом деле неиндексированные таблицы практически никогда не используются. Для каждой таблицы обычно объявляется один или несколько индексов, соответствующих потенциальным ключам. При помощи этих индексов поиск записи производится очень быстро и практически не зависит от количества строк и атрибутов в таблице (хотя, конечно, некоторая зависимость имеется). Если для таблицы объявлено несколько индексов, то при выполнении операций обновления и удаления эти индексы должны быть перестроены, на что тратится дополнительное время. Таким образом, скорость выполнения операций обновления и удаления также уменьшается при увеличении количества индексов у таблицы и мало зависит от числа строк в таблице.

Можно предположить, что, чем больше атрибутов имеет таблица, тем больше для неё будет объявлено индексов. Эта зависимость, конечно, не прямая, но при одинаковых подходах к физическому моделированию обычно так и происходит. Таким образом, можно принять допущение, что, *чем больше атрибутов имеют отношения*, разработанные в ходе логического моделирования, *тем медленнее будут выполняться операции обновления данных*, за счёт затраты времени на перестройку большего количества индексов.

Скорость операций выборки данных. Одно из назначений базы данных – предоставление информации пользователям. Информация извлекается из реляционной базы данных при помощи оператора SQL – SELECT. Одной из наиболее дорогостоящих операций при выполнении оператора SELECT является операция соединения таблиц. Таким образом, чем больше взаимосвязанных отношений было создано в ходе логического моделирования, тем больше вероятность того, что при выполнении запросов эти отношения будут соединяться, и, следовательно, тем медленнее будут выполняться запросы. Таким образом, увеличение количества отношений приводит к замедлению выполнения операций выборки данных, особенно если запросы заранее неизвестны.

Основной пример. Рассмотрим в качестве предметной области некоторую организацию, выполняющую некоторые проекты. Модель предметной области опишем следующим неформальным текстом.

1. Сотрудники организации выполняют проекты.
2. Проекты состоят из нескольких заданий.
3. Каждый сотрудник может участвовать в одном или нескольких проектах или временно не участвовать ни в каких проектах.
4. Над каждым проектом может работать несколько сотрудников, или временно проект может быть приостановлен, тогда над ним не работает ни один сотрудник.
5. Над каждым заданием в проекте работает один сотрудник.
6. Каждый сотрудник числится в одном отделе.
7. Каждый сотрудник имеет телефон, находящийся в его отделе.

В ходе дополнительного уточнения того, какие данные необходимо учитывать, выяснилось следующее.

1. О каждом сотруднике необходимо хранить табельный номер и фамилию. Табельный номер является уникальным для каждого сотрудника.
2. Каждый отдел имеет уникальный номер.
3. Каждый проект имеет номер и наименование. Номер проекта является уникальным.
4. Каждая работа из проекта имеет номер, уникальный в пределах проекта. Работы в разных проектах могут иметь одинаковые номера.

3.3. 1НФ (ПЕРВАЯ НОРМАЛЬНАЯ ФОРМА)

Первая нормальная форма (1НФ) – это обычное отношение. Согласно определению отношений любое отношение автоматически уже находится в 1НФ. Напомним кратко свойства отношений (это и будут свойства 1НФ):

- в отношении нет одинаковых кортежей;
- кортежи не упорядочены;
- атрибуты не упорядочены и различаются по наименованию;
- все значения атрибутов атомарны.

В ходе логического моделирования на первом шаге предложено хранить данные в одном отношении, имеющем следующие атрибуты:

СОТРУДНИКИ_ОТДЕЛЫ_ПРОЕКТЫ (*Н_СОТР*, *ФАМ*, *Н_ОТД*, *ТЕЛ*, *Н_ПРО*, *ПРОЕКТ*, *Н_ЗАДАН*), где

Н_СОТР – табельный номер сотрудника

ФАМ – фамилия сотрудника

Н_ОТД – номер отдела, в котором числится сотрудник

ТЕЛ – телефон сотрудника

Н_ПРО – номер проекта, над которым работает сотрудник

ПРОЕКТ – наименование проекта, над которым работает сотрудник

Н_ЗАДАН – номер задания, над которым работает сотрудник. Так как сотрудник в каждом проекте выполняет одно задание, то в качестве потенциального ключа отношения необходимо взять пару атрибутов {*Н_СОТР, Н_ПРО*}.

В текущий момент состояние предметной области отражается следующими фактами:

- сотрудник Иванов, работающий в 1 отделе, выполняет в первом проекте «Космос» задание 1 и во втором проекте «Климат» задание 1;
- сотрудник Петров, работающий в 1 отделе, выполняет в первом проекте «Космос» задание 2;
- сотрудник Сидоров, работающий во 2 отделе, выполняет в первом проекте «Космос» задание 3 и во втором проекте «Климат» задание 2.

Это состояние отражается в таблице (курсивом выделены ключевые атрибуты):

Н_СОТР	*ФАМ*	*Н_ОТД*	*ТЕЛ*	*Н_ПРО*	*ПРОЕКТ*	*Н_ЗАДАН*
1	Иванов	1	11-22-33	*1*	Космос	1

1	Иванов	1	11-22-33	*2*	Климат	1
2	Петров	1	11-22-33	*1*	Космос	2
3	Сидоров	2	33-22-11	*1*	Космос	3
3	Сидоров	2	33-22-11	*2*	Климат	2

Аномалии обновления. Даже одного взгляда на таблицу отношения *СОТРУДНИКИ_ОТДЕЛЫ_ПРОЕКТЫ* достаточно, чтобы увидеть, что данные хранятся в ней с большой избыточностью. Во многих строках повторяются фамилии сотрудников, номера телефонов, наименования проектов. Кроме того, в данном отношении хранятся вместе независимые друг от друга данные – и данные о сотрудниках, и об отделах, и о проектах, и о работах по проектам. Пока никаких действий с отношением не производится, это нестрашно. Но как только состояние предметной области изменяется, так при попытках соответствующим образом изменить состояние базы данных возникает большое количество проблем.

Исторически эти проблемы получили название *аномалии обновления*. Попытки дать строгое понятие аномалии в базе данных не являются вполне удовлетворительными. В данных работах аномалии определены как противоречие между моделью предметной области и физической моделью данных, поддерживаемых средствами конкретной СУБД. Аномалии возникают в том случае, когда наши знания о предметной области оказываются по каким-то причинам невыразимыми в схеме БД или входящими в противоречие с ней. Мы придерживаемся другой точки зрения, заключающейся в том, что аномалий в смысле определений упомянутых авторов нет, а есть либо неадекватность модели данных предметной области, либо некоторые дополнительные трудности в реализации ограничений предметной области средствами СУБД. Более глубокое обсуждение проблемы строгого определения понятия аномалий выходит за пределы данной работы.

Таким образом, мы будем придерживаться интуитивного понятия аномалии как неадекватности модели данных предметной области (что говорит на самом деле о том, что логическая модель данных попросту неверна!) или как необходимости дополнительных усилий для реализации всех ограничений, определённых в предметной области (дополнительный программный код в виде триггеров или хранимых процедур).

Так как аномалии проявляют себя при выполнении операций, изменяющих состояние базы данных, то различают следующие виды аномалий:

- аномалии вставки (INSERT);
- аномалии обновления (UPDATE);
- аномалии удаления (DELETE).

В отношении *СОТРУДНИКИ_ОТДЕЛЫ_ПРОЕКТЫ* можно привести примеры следующих аномалий.

***Аномалии вставки* (*INSERT*).** В отношение *СОТРУДНИКИ_ОТДЕЛЫ_ПРОЕКТЫ* нельзя вставить данные о сотруднике, который пока не участвует ни в одном проекте. Действительно, если, например, во втором отделе появляется новый сотрудник, скажем, Пушников и он пока не участвует ни в одном проекте, то мы должны вставить в отношение кортеж (4, Пушников, 2, 33-22-11, null, null, null). Это сделать невозможно, так как атрибут *Н_ПРО* (номер проекта) входит в состав потенциального ключа и, следовательно, не может содержать null-значений.

Точно так же нельзя вставить данные о проекте, над которым пока не работает ни один сотрудник.

Причина аномалии – хранение в одном отношении разнородной информации (и о сотрудниках, и о проектах, и о работах по проекту).

Вывод. Логическая модель данных неадекватна модели предметной области. База данных, основанная на такой модели, будет работать неправильно.

***Аномалии обновления* (*UPDATE*).** Фамилии сотрудников, наименования проектов, номера телефонов повторяются во многих кортежах отношения. Поэтому если сотрудник меняет фамилию, или проект меняет наименование, или меняется номер телефона, то такие изменения необходимо одновременно выполнить во всех местах, где эта фамилия, наименование или номер телефона встречаются,

иначе отношение станет некорректным (например, один и тот же проект в разных кортежах будет называться по-разному). Таким образом, обновление базы данных одним действием реализовать невозможно. Для поддержания отношения в целостном состоянии необходимо написать триггер, который при обновлении одной записи корректно исправлял бы данные и в других местах.

Причина аномалии – избыточность данных, также порождённая тем, что в одном отношении хранится разнородная информация.

Вывод. Увеличивается сложность разработки базы данных. База данных, основанная на такой модели, будет работать правильно только при наличии дополнительного программного кода в виде триггеров.

***Аномалии удаления* (*DELETE*).** При удалении некоторых данных может произойти потеря другой информации. Например, если закрыть проект «Космос» и удалить все строки, в которых он встречается, то будут потеряны все данные о сотруднике Петрове. Если удалить сотрудника Сидорова, то будет потеряна информация о том, что в отделе номер 2 находится телефон 33-22-11. Если по проекту временно прекращены работы, то при удалении данных о работах по этому проекту будут удалены и данные о самом проекте (наименование проекта). При этом если был сотрудник, который работал только над этим проектом, то будут потеряны и данные об этом сотруднике.

Причина аномалии – хранение в одном отношении разнородной информации (и о сотрудниках, и о проектах, и о работах по проекту).

Вывод. Логическая модель данных неадекватна модели предметной области. База данных, основанная на такой модели, будет работать неправильно.

Функциональные зависимости. Отношение *СОТРУДНИКИ_ОТДЕЛЫ_ПРОЕКТЫ* находится в 1НФ, при этом, как было показано выше, логическая модель данных неадекватна модели предметной области. Таким образом, первой нормальной формы недостаточно для правильного моделирования данных.

Определение функциональной зависимости. Для устранения указанных аномалий (а на самом деле *для правильного проектирования модели данных*!) применяется метод нормализации отношений. Нормализация основана на понятии функциональной зависимости атрибутов отношения.

Определение 1. Пусть R – отношение. Множество атрибутов Y *функционально зависимо* от множества атрибутов X (X *функционально определяет* Y) тогда и только тогда, когда *для любого состояния* отношения R для любых кортежей $r_1, r_2 \in R$ из того, что $r_1X = r_2X$ следует, что $r_1Y = r_2Y$ (т.е. во всех кортежах, имеющих одинаковые значения атрибутов X, значения атрибутов Y также совпадают *в любом состоянии* отношения R). Символически функциональная зависимость записывается $X \rightarrow Y$.

Множество атрибутов X называется *детерминантом функциональной зависимости*, а множество атрибутов Y называется *зависимой частью*.

Замечание. Если атрибуты X составляют потенциальный ключ отношения R, то *любой* атрибут отношения R функционально зависит от X.

Пример 1. В отношении *СОТРУДНИКИ_ОТДЕЛЫ_ ПРОЕКТЫ* можно привести следующие примеры функциональных зависимостей:

Зависимость атрибутов от ключа отношения:

{Н_СОТР, Н_ПРО} $\rightarrow$ *ФАМ*

{Н_СОТР, Н_ПРО} $\rightarrow$ *Н_ОТД*

{Н_СОТР, Н_ПРО} $\rightarrow$ *ТЕЛ*

{Н_СОТР, Н_ПРО} $\rightarrow$ *ПРОЕКТ*

{Н_СОТР, Н_ПРО} $\rightarrow$ *Н_ЗАДАН*

Зависимость атрибутов, характеризующих сотрудника, зависимость от табельного номера сотрудника:

Н_СОТР $\rightarrow$ *ФАМ*

Н_СОТР $\rightarrow$ *Н_ОТД*

Н_СОТР $\rightarrow$ *ТЕЛ*

Зависимость наименования проекта от номера проекта:

Н_ПРО $\rightarrow$ *ПРОЕКТ*

Зависимость номера телефона от номера отдела:

Н_ОТД → ТЕЛ

Замечание. Приведённые функциональные зависимости отражают взаимосвязи, обнаруженные между объектами предметной области, и являются дополнительными ограничениями, определяемыми предметной областью. Таким образом, функциональная зависимость – *семантическое понятие*. Она возникает, когда по значениям одних данных в предметной области можно определить значения других данных. Например, зная табельный номер сотрудника, можно определить его фамилию, по номеру отдела можно определить телефон. Функциональная зависимость задаёт дополнительные ограничения на данные, которые могут храниться в отношениях. Для корректности базы данных (адекватности предметной области) необходимо при выполнении операций модификации базы данных проверять все ограничения, определённые функциональными зависимостями.

Функциональные зависимости отношений и математическое понятие функциональной зависимости. Функциональная зависимость атрибутов отношения напоминает понятие функциональной зависимости в математике. *Но это не одно и то же*. Для сравнения напомним математическое понятие функциональной зависимости:

Определение 2. Функциональная зависимость (*функция*) – это тройка объектов

$$\{X, Y, f\},$$

где X – множество (*область определения*); Y – множество (*множество значений*); f – правило, согласно которому каждому элементу $x \in X$ ставится в соответствие один и только один элемент $y \in Y$ (*правило функциональной зависимости*).

Функциональная зависимость обычно обозначается как $f: X \to Yf$ или $y = f(x)$.

Замечание. Правило f может быть задано любым способом – в виде формулы (чаще всего), при помощи таблицы значений, при помощи графика, текстовым описанием и т.д.

Функциональная зависимость атрибутов отношения тоже напоминает это определение. Действительно:

– в качестве области определения выступает домен, на котором определён атрибут X (или декартово произведение доменов, если X является множеством атрибутов);

– в качестве множества значений выступает домен, на котором определён атрибут Y (или декартово произведение доменов).

Правило f реализуется следующим алгоритмом: 1) по данному значению атрибута X найти любой кортеж отношения, содержащий это значение; 2) значение атрибута Y в этом кортеже и будет значением функциональной зависимости, соответствующим данному X. Определение функциональной зависимости в отношении гарантирует, что найденное значение Y не зависит от выбора кортежа, поэтому правило f определено корректно.

Отличие от математического понятия отношения состоит в том, что если рассматривать математическое понятие функции, то *для фиксированного значения* $x \in X$ соответствующее значение функции $y = f(x)$ *всегда одно и то же*. Например, если задана функция $y = x^2$, то для значения $x = 2$ соответствующее значение y всегда будет равно 4. В противоположность этому в отношениях значение зависимого атрибута может принимать различные значения в различных состояниях базы данных. Например, атрибут *ФАМ* функционально зависит от атрибута *Н_СОТР*. Предположим, что сейчас сотрудник с табельным номером 1 имеет фамилию Иванов, т.е. при значении детерминанта равного 1, значение зависимого аргумента равно «Иванов». Но сотрудник может сменить фамилию, например на «Сидоров». Теперь при том же значении детерминанта, равного 1, значение зависимого аргумента равно «Сидоров».

Таким образом, понятие функциональной зависимости атрибутов нельзя считать полностью эквивалентным математическому понятию функциональной зависимости, так как значения этой зависимости различны при разных состояниях отношения, и, самое главное, эти значения могут меняться непредсказуемо.

Функциональная зависимость атрибутов утверждает лишь то, что для каждого конкретного состояния базы данных по значению одного атрибута (детерминанта) можно однозначно определить значение другого атрибута (зависимой части). Но конкретные значения зависимой части могут быть различны в различных состояниях базы данных.

3.4. 2НФ (ВТОРАЯ НОРМАЛЬНАЯ ФОРМА)

Определение 3. Отношение *R* находится во *второй нормальной форме* (*2НФ*) тогда и только тогда, когда отношение находится в 1НФ и *нет неключевых атрибутов, зависящих от части сложного ключа.* (*Неключевой атрибут* – это атрибут, не входящий в состав никакого потенциального ключа).

Замечание. Если потенциальный ключ отношения является простым, то отношение автоматически находится в 2НФ.

Отношение *СОТРУДНИКИ_ОТДЕЛЫ_ПРОЕКТЫ* не находится в 2НФ, так как есть атрибуты, зависящие от части сложного ключа:

Зависимость атрибутов, характеризующих сотрудника, от табельного номера сотрудника является зависимостью от части сложного ключа:

Н_СОТР → *ФАМ*

Н_СОТР → *Н_ОТД*

Н_СОТР → *ТЕЛ*

Зависимость наименования проекта от номера проекта является зависимостью от части сложного ключа:

Н_ПРО → *ПРОЕКТ*

Для того чтобы устранить зависимость атрибутов от части сложного ключа, нужно произвести *декомпозицию* отношения на несколько отношений. При этом те атрибуты, которые зависят от части сложного ключа, выносятся в отдельное отношение.

Отношение *СОТРУДНИКИ_ОТДЕЛЫ_ПРОЕКТЫ* декомпозируем на три отношения – *СОТРУДНИКИ_ОТДЕЛЫ, ПРОЕКТЫ, ЗАДАНИЯ.*

Отношение *СОТРУДНИКИ_ОТДЕЛЫ* (*Н_СОТР, ФАМ, Н_ОТД, ТЕЛ*):

Функциональные зависимости:

Зависимость атрибутов, характеризующих сотрудника, от табельного номера сотрудника:

Н_СОТР → *ФАМ*

Н_СОТР → *Н_ОТД*

Н_СОТР → *ТЕЛ*

Зависимость номера телефона от номера отдела:

Н_ОТД → *ТЕЛ*

Н_СОТР	*ФАМ*	*Н_ОТД*	*ТЕЛ*
1	Иванов	1	11-22-33
2	Петров	1	11-22-33
3	Сидоров	2	33-22-11

Отношение *ПРОЕКТЫ* (*Н_ПРО, ПРОЕКТ*):

Функциональные зависимости:

Н_ПРО → *ПРОЕКТ*

Н_ПРО	*ПРОЕКТ*
1	Космос
2	Климат

Отношение *ЗАДАНИЯ* (*Н_СОТР, Н_ПРО, Н_ЗАДАН*):

Функциональные зависимости:

{*Н_СОТР, Н_ПРО*} → *Н_ЗАДАН*

Н_СОТР	*Н_ПРО*	*Н_ЗАДАН*
1	*1*	1

1	*2*	1
2	*1*	2
3	*1*	3
3	*2*	2

Анализ декомпозированных отношений. Отношения, полученные в результате декомпозиции, находятся в 2НФ. Действительно, отношения *СОТРУДНИКИ_ОТДЕЛЫ* и *ПРОЕКТЫ* имеют простые ключи, следовательно, автоматически находятся в 2НФ, отношение *ЗАДАНИЯ* имеет сложный ключ, но единственный неключевой атрибут *Н_ЗАДАН* функционально зависит от всего ключа *{Н_СОТР, Н_ПРО}*.

Часть аномалий обновления устранена. Так, данные о сотрудниках и проектах теперь хранятся в различных отношениях, поэтому при появлении сотрудников, не участвующих ни в одном проекте, просто добавляются кортежи в отношение *СОТРУДНИКИ_ОТДЕЛЫ*. Точно так же при появлении проекта, над которым не работает ни один сотрудник, просто вставляется кортеж в отношение *ПРОЕКТЫ*.

Фамилии сотрудников и наименования проектов теперь хранятся без избыточности. Если сотрудник сменит фамилию или проект сменит наименование, то такое обновление будет произведено в одном месте.

Если по проекту временно прекращены работы, но требуется, чтобы сам проект сохранился, то для этого проекта удаляются соответствующие кортежи в отношении *ЗАДАНИЯ*, а данные о самом проекте и данные о сотрудниках, участвовавших в проекте, остаются в отношениях *ПРОЕКТЫ* и *СОТРУДНИКИ_ОТДЕЛЫ*.

Тем не менее часть аномалий разрешить не удалось.

***Оставшиеся аномалии вставки* (*INSERT*).** В отношение СОТРУДНИКИ_ОТДЕЛЫ нельзя вставить кортеж (4, Пушников, 1, 33-22-11), так как при этом получится, что два сотрудника из 1-го отдела (Иванов и Пушников) имеют разные номера телефонов, а это противоречит модели предметной области. В этой ситуации можно предложить два решения в зависимости от того, что реально произошло в предметной области. Другой номер телефона может быть введён по двум причинам – по ошибке человека, вводящего данные о новом сотруднике, или потому что номер в отделе действительно изменился. Тогда можно написать триггер, который при вставке записи о сотруднике проверяет, совпадает ли телефон с уже имеющимся телефоном у другого сотрудника этого же отдела. Если номера отличаются, то система должна задать вопрос, оставить ли старый номер в отделе или заменить его новым. Если нужно оставить старый номер (новый номер введён ошибочно), то кортеж с данными о новом сотруднике будет вставлен, но номер телефона будет у него тот, который уже есть в отделе (в данном случае 11-22-33). Если же номер в отделе действительно изменился, то кортеж будет вставлен с новым номером и одновременно будут изменены номера телефонов у всех сотрудников этого же отдела. И в том и в другом случае не обойтись без разработки громоздкого триггера.

Причина аномалии – избыточность данных, порождённая тем, что в одном отношении хранится разнородная информация (о сотрудниках и об отделах).

Вывод. Увеличивается сложность разработки базы данных. База данных, основанная на такой модели, будет работать правильно только при наличии дополнительного программного кода в виде триггеров.

***Оставшиеся аномалии обновления* (*UPDATE*).** Одни и те же номера телефонов повторяются во многих кортежах отношения. Поэтому если в отделе меняется номер телефона, то такие изменения необходимо одновременно выполнить во всех местах, где этот номер телефона встречается, иначе отношение станет некорректным. Таким образом, обновление базы данных одним действием реализовать невозможно. Необходимо написать триггер, который при обновлении одной записи корректно исправляет номера телефонов в других местах.

Причина аномалии – избыточность данных, также порождённая тем, что в одном отношении хранится разнородная информация.

Вывод. Увеличивается сложность разработки базы данных. База данных, основанная на такой модели, будет работать правильно только при наличии дополнительного программного кода в виде триггеров.

Оставшиеся аномалии удаления (DELETE). При удалении некоторых данных по-прежнему может произойти потеря другой информации. Например, если удалить сотрудника Сидорова, то будет потеряна информация о том, что в отделе номер 2 находится телефон 33-22-11.

Причина аномалии – хранение в одном отношении разнородной информации (и о сотрудниках, и об отделах).

Вывод. Логическая модель данных неадекватна модели предметной области. База данных, основанная на такой модели, будет работать неправильно.

Заметим, что при переходе ко второй нормальной форме отношения стали почти адекватными предметной области. Остались также трудности в разработке базы данных, связанные с необходимостью написания триггеров, поддерживающих целостность базы данных. Эти трудности теперь связаны только с одним отношением *СОТРУДНИКИ_ОТДЕЛЫ*.

3.5. 3НФ (ТРЕТЬЯ НОРМАЛЬНАЯ ФОРМА)

Определение 4. Атрибуты называются *взаимно независимыми*, если ни один из них не является функционально зависимым от другого.

Определение 5. Отношение *R* находится в *третьей нормальной форме* (*3НФ*) тогда и только тогда, когда отношение находится в 2НФ и *все неключевые атрибуты взаимно независимы*.

Отношение *СОТРУДНИКИ_ОТДЕЛЫ* не находится в 3НФ, так как имеется функциональная зависимость неключевых атрибутов (зависимость номера телефона от номера отдела):

$Н\_ОТД \rightarrow ТЕЛ$

Для того чтобы устранить зависимость неключевых атрибутов, нужно произвести декомпозицию отношения на несколько отношений. При этом те неключевые атрибуты, которые являются зависимыми, выносятся в отдельное отношение.

Отношение *СОТРУДНИКИ_ОТДЕЛЫ* декомпозируем на два отношения – *СОТРУДНИКИ*, *ОТДЕЛЫ*.

Отношение *СОТРУДНИКИ* (*Н_СОТР*, *ФАМ*, *Н_ОТД*).

Функциональные зависимости:

Зависимость атрибутов, характеризующих сотрудника, от табельного номера сотрудника:

$Н\_СОТР \rightarrow ФАМ$

$Н\_СОТР \rightarrow Н\_ОТД$

$Н\_СОТР \rightarrow ТЕЛ$

Н_СОТР	*ФАМ*	*Н_ОТД*
1	Иванов	1
2	Петров	1
3	Сидоров	2

Отношение *ОТДЕЛЫ* (*Н_ОТД*, *ТЕЛ*):

Функциональные зависимости:

Зависимость номера телефона от номера отдела:

$Н\_ОТД \rightarrow ТЕЛ$

Н_ОТД	*ТЕЛ*
1	11-22-33
2	33-22-11

Обратим внимание на то, что атрибут *Н_ОТД, не являвшийся ключевым* в отношении *СОТРУДНИКИ_ОТДЕЛЫ, становится потенциальным ключом* в отношении *ОТДЕЛЫ*. Именно за счёт этого устраняется избыточность, связанная с многократным хранением одних и тех же номеров телефонов.

Вывод. Все обнаруженные аномалии обновления устранены. Реляционная модель, состоящая из четырёх отношений – *СОТРУДНИКИ, ОТДЕЛЫ, ПРОЕКТЫ, ЗАДАНИЯ*, находящихся в третьей нормальной форме, является адекватной описанной модели предметной области и требует наличия только тех триггеров, которые поддерживают ссылочную целостность. Такие триггеры являются стандартными и не требуют больших усилий в разработке.

***Алгоритм нормализации* (*приведение к 3НФ*).** Итак, алгоритм нормализации (т.е. алгоритм приведения отношений к 3НФ) описывается следующим образом.

Шаг 1 (*Приведение к 1НФ*). На первом шаге задаётся одно или несколько отношений, отображающих понятия предметной области. По модели предметной области (не по внешнему виду полученных отношений!) выписываются обнаруженные функциональные зависимости. Все отношения автоматически находятся в 1НФ.

Шаг 2 (*Приведение к 2НФ*). Если в некоторых отношениях обнаружена зависимость атрибутов от части сложного ключа, то проводим декомпозицию этих отношений на несколько отношений следующим образом: те атрибуты, которые зависят от части сложного ключа, выносятся в отдельное отношение вместе с этой частью ключа. В исходном отношении остаются все ключевые атрибуты:

Исходное отношение: $R(K_1,\ K_2,\ A_1,\ ...,\ A_n,\ B_1, ...,\ B_m)$.

Ключ: $\{K_1,\ K_2\}$ – сложный.

Функциональные зависимости:

$\{K_1,\ K_2\} \to \{A_1,\ ...,\ A_n,\ B_1, ...,\ B_m\}$ – зависимость всех атрибутов от ключа отношения.

$\{K_1\} \to \{A_1,\ ...,\ A_n\}$ – зависимость некоторых атрибутов от части сложного ключа.

Декомпозированные отношения:

$R_1(K_1,\ K_2,\ B_1, ...,\ B_m)$ – остаток от исходного отношения. Ключ $\{K_1,\ K_2\}$.

$R_2(K_1,\ A_1, ...,\ A_n)$ – атрибуты, вынесенные из исходного отношения вместе с частью сложного ключа. Ключ K_1.

Шаг 3 (*Приведение к 3НФ*). Если в некоторых отношениях обнаружена зависимость некоторых неключевых атрибутов других неключевых атрибутов, то проводим декомпозицию этих отношений следующим образом: те неключевые атрибуты, которые зависят от других неключевых атрибутов, выносятся в отдельное отношение. В новом отношении ключом становится детерминант функциональной зависимости:

Исходное отношение: $R_1(K,\ B_1, ...,\ B_m)$.

Ключ: K.

Функциональные зависимости:

$K \to \{A_1,\ ...,\ A_n,\ B_1, ...,\ B_m\}$ – зависимость всех атрибутов от ключа отношения.

$\{A_1,\ ...,\ A_n\} \to \{B_1, ...,\ B_m\}$ – зависимость некоторых неключевых атрибутов от других неключевых атрибутов.

Декомпозированные отношения:

$R_1(K,\ A_1, ...,\ A_n)$ – остаток от исходного отношения.

Ключ: K.

$R_2(A_1, ..., A_n,\ B_1, ...,\ B_m)$ – атрибуты, вынесенные из исходного отношения вместе с детерминантом функциональной зависимости. Ключ $\{A_1, ...,\ A_n\}$.

Замечание. На практике при создании логической модели данных, как правило, не следуют прямо приведённому алгоритму нормализации. Опытные разработчики обычно сразу строят отношения в 3НФ. Кроме того, основным средством разработки логических моделей данных являются различные варианты ER-диаграмм. Особенность этих диаграмм в том, что они сразу позволяют создавать отношения в 3НФ. Тем не менее приведённый алгоритм важен по двум причинам. Во-первых, этот алгоритм показывает, какие проблемы возникают при разработке слабо нормализованных отношений. Во-вторых, как правило, модель предметной области никогда не бывает правильно разработана с первого шага. Эксперты предметной области могут забыть о чём-либо упомянуть, разработчик может неправильно понять эксперта, во время разработки могут измениться правила, принятые в предметной области и т.д. Всё это может привести к появлению новых зависимостей, которые отсутствовали в первоначальной модели предметной области. Тут как раз и необходимо использовать алгоритм нормализации хотя бы для того, чтобы убедиться, что отношения остались в 3НФ и логическая модель не ухудшилась.

3.6. OLTP И OLAP-СИСТЕМЫ

Можно выделить некоторые классы систем, для которых больше подходят сильно или слабо нормализованные модели данных.

Сильно нормализованные модели данных хорошо подходят для так называемых *OLTP-приложений* (*On-Line Transaction Processing* (*OLTP*) – *оперативная обработка транзакций*). Типичными примерами OLTP-приложений являются системы складского учёта, системы заказов билетов, банковские системы, выполняющие операции по переводу денег и т.п. Основная функция подобных систем заключается в выполнении большого количества коротких транзакций. Сами транзакции выглядят относительно просто, например, «снять сумму денег со счёта А, добавить эту сумму на счёт В». Проблема заключается в том, что, во-первых, транзакций очень много; во-вторых, выполняются они одновременно (к системе может быть подключено несколько тысяч одновременно работающих пользователей); в-третьих, при возникновении ошибки транзакция должна целиком откатиться и вернуть систему к состоянию, которое было до начала транзакции (не должно быть ситуации, когда деньги сняты со счёта А, но не поступили на счёт В). Практически все запросы к базе данных в OLTP-приложениях состоят из команд вставки, обновления, удаления. Запросы на выборку в основном предназначены для предоставления пользователям возможности выбора из различных справочников. Большая часть запросов известна заранее ещё на этапе проектирования системы. Таким образом, критическим для OLTP-приложений является скорость и надёжность выполнения коротких операций обновления данных. Чем выше уровень нормализации данных в OLTP-приложении, тем оно, как правило, быстрее и надёжнее. Отступления от этого правила могут происходить тогда, когда уже на этапе разработки известны некоторые часто возникающие запросы, которые требуют соединения отношений и от скорости выполнения которых существенно зависит работа приложений. В этом случае можно пожертвовать нормализацией для ускорения выполнения подобных запросов.

Другим типом приложений являются так называемые *OLAP-приложения* (*On-Line Analitical Processing* (*OLAP*) – *оперативная аналитическая обработка данных*). Это обобщённый термин, характеризующий принципы построения *систем поддержки принятия решений* (*Decision Support System – DSS*), *хранилищ данных* (*Data Warehouse*), *систем интеллектуального анализа данных* (*Data Mining*). Такие системы предназначены для нахождения зависимостей между данными (например, можно попытаться определить, как связан объём продаж товаров с характеристиками потенциальных покупателей), для проведения анализа «что если…». OLAP-приложения оперируют с большими массивами данных, уже накопленными в OLTP-приложениях, взятыми их электронных таблиц или из других источников данных. Такие системы характеризуются следующими признаками:

– добавление в систему новых данных происходит относительно редко крупными блоками (например, раз в квартал загружаются данные по итогам квартальных продаж из OLTP-приложения);

– данные, добавленные в систему, обычно никогда не удаляются;

– перед загрузкой данные проходят различные процедуры «очистки», связанные с тем, что в одну систему могут поступать данные из многих источников, имеющих различные форматы представления для одних и тех же понятий, данные могут быть некорректны, ошибочны;

– запросы к системе являются нерегламентированными и, как правило, достаточно сложными. Очень часто новый запрос формулируется аналитиком для уточнения результата, полученного в результате предыдущего запроса;

– скорость выполнения запросов важна, но не критична.

Данные OLAP-приложений обычно представлены в виде одного или нескольких гиперкубов, измерения которого представляют собой справочные данные, а в ячейках самого гиперкуба хранятся собственно данные. Например, можно построить гиперкуб, измерениями которого являются: время (в кварталах, годах), тип товара и отделения компании, а в ячейках хранятся объёмы продаж. Такой гиперкуб будет содержать данные о продажах различных типов товаров по кварталам и подразделениям. Основываясь на этих данных, можно отвечать на вопросы вроде «у какого подразделения самые лучшие объёмы продаж в текущем году?» или «каковы тенденции продаж отделений Юго-Западного региона в текущем году по сравнению с предыдущим годом?»

Физически гиперкуб может быть построен на основе специальной *многомерной модели данных* (*MOLAP – Multidimensional OLAP*) или средствами реляционной модели данных (*ROLAP – Relational OLAP*).

Возвращаясь к проблеме нормализации данных, можно сказать, что в системах OLAP, использующих реляционную модель данных (ROLAP), данные целесообразно хранить в виде слабо нормализованных отношений, содержащих заранее вычисленные основные итоговые данные. Большая избыточность и связанные с ней проблемы тут нестрашны, так как обновление происходит только в момент загрузки новой порции данных. При этом происходит как добавление новых данных, так и пересчёт итогов.

Корректность процедуры нормализации – декомпозиция без потерь. Как было показано выше, алгоритм нормализации состоит в выявлении функциональных зависимостей предметной области и соответствующей декомпозиции отношений. Предположим, что мы уже имеем работающую систему, в которой накоплены данные. Пусть данные корректны в текущий момент, т.е. факты предметной области правильно отражаются текущим состоянием базы данных. Если в предметной области обнаружена новая функциональная зависимость (либо она была пропущена на этапе моделирования предметной области, либо просто изменилась предметная область), то возникает необходимость заново нормализовать данные. При этом некоторые отношения придётся декомпозировать в соответствии с алгоритмом нормализации. Возникают естественные вопросы – что произойдёт с уже накопленными данными? Не будут ли данные потеряны в ходе декомпозиции? Можно ли вернуться обратно к исходным отношениям, если будет принято решение отказаться от декомпозиции, восстановятся ли при этом данные?

Для ответов на эти вопросы нужно ответить на вопрос – что же представляет собой декомпозиция отношений с точки зрения операций реляционной алгебры? При декомпозиции мы из одного отношения получаем два или более отношений, каждое из которых содержит часть атрибутов исходного отношения. В полученных новых отношениях необходимо удалить дубликаты строк, если таковые возникли. Это в точности означает, что декомпозиция отношения есть не что иное, как взятие одной или нескольких проекций исходного отношения так, чтобы эти проекции в совокупности содержали (возможно, с повторениями) *все* атрибуты исходного отношения, т.е. при декомпозиции *не должны теряться атрибуты* отношений. Но при декомпозиции также не должны потеряться и сами данные. Данные можно считать непотерянными в том случае, если возможна обратная операция – по декомпозированным отношениям можно восстановить исходное отношение *в точности в прежнем виде*. Операцией, обратной операции проекции, является операция соединения отношений. Имеется большое количество видов операции соединения (см. гл. 4). Так как при восстановлении исходного отношения путём соединения проекций не должны появиться новые атрибуты, то необходимо использовать *естественное соединение*.

Определение 6. Проекция $R[X]$ отношения R на множество атрибутов X называется *собственной*, если множество атрибутов X является *собственным подмножеством* множества атрибутов отношения R (т.е. множество атрибутов X не совпадает с множеством *всех* атрибутов отношения R).

Определение 7. *Собственные* проекции R_1 и R_2 отношения R называются *декомпозицией без потерь*, если отношение R *точно восстанавливается* из них при помощи естественного соединения *для любого состояния* отношения R: R_1 JOIN $R_2 = R$.

Рассмотрим пример, показывающий, что декомпозиция без потерь происходит не всегда.

Пример 2. Пусть дано отношение R:

НОМЕР	*ФАМИЛИЯ*	*ЗАРПЛАТА*
1	Иванов	1000
2	Петров	1000

Рассмотрим первый вариант декомпозиции отношения R на два отношения:

НОМЕР	*ЗАРПЛАТА*

1	1000
2	1000

ФАМИЛИЯ	*ЗАРПЛАТА*
Иванов	1000
Петров	1000

Естественное соединение этих проекций, имеющих общий атрибут «ЗАРПЛАТА», очевидно, будет следующим (каждая строка одной проекции соединится с каждой строкой другой проекции):

Таблица Отношение R_1 JOIN $R_2 \neq R$.

НОМЕР	*ФАМИЛИЯ*	*ЗАРПЛАТА*
1	Иванов	1000
1	Петров	1000
2	Иванов	1000
2	Петров	1000

Итак, данная декомпозиция не является декомпозицией без потерь, так как исходное отношение не восстанавливается в точном виде по проекциям.

Рассмотрим другой вариант декомпозиции:

Таблица Отношение R_1

НОМЕР	*ФАМИЛИЯ*
1	Иванов
2	Петров

Таблица Отношение R_2

НОМЕР	*ЗАРПЛАТА*
1	1000
2	1000

По данным проекциям, имеющим общий атрибут «НОМЕР», исходное отношение восстанавливается в точном виде. Тем не менее нельзя сказать, что данная декомпозиция является декомпозицией без потерь, так как мы рассмотрели только одно конкретное состояние отношения R и не можем сказать, будет ли и в других состояниях отношение R восстанавливаться точно. Например, предположим, что отношение R перешло в состояние:

Таблица Отношение R

НОМЕР	*ФАМИЛИЯ*	*ЗАРПЛАТА*
1	Иванов	1000
2	Петров	1000
2	Сидоров	2000

Кажется, что этого не может быть, так как значения в атрибуте «НОМЕР» повторяются. Но мы же ничего не говорили о ключе этого отношения! Сейчас проекции будут иметь вид:

Таблица Отношение R_1

НОМЕР	*ФАМИЛИЯ*
1	Иванов

2	Петров
2	Сидоров

Таблица Отношение R_2

НОМЕР	*ЗАРПЛАТА*
1	1000
2	1000
2	2000

Естественное соединение этих проекций будет содержать лишние кортежи:

Таблица Отношение R_1 JOIN $R_2 \neq R$.

НОМЕР	*ФАМИЛИЯ*	*ЗАРПЛАТА*
1	Иванов	1000
2	Петров	1000
2	Петров	2000
2	Сидоров	1000
2	Сидоров	2000

Вывод. *Без дополнительных ограничений* на отношение R нельзя говорить о декомпозиции без потерь.

Такими дополнительными ограничениями и являются функциональные зависимости.

ВОПРОСЫ ДЛЯ САМОПРОВЕРКИ

1. Моделирование предметной области при разработке БД.
2. Логическое моделирование данных.
3. Физическое моделирование данных.
4. Первая нормальная форма отношения.
5. Вторая нормальная форма отношения.
6. Третья нормальная форма отношения.

4. МЕТОДОЛОГИЯ ПРОЕКТИРОВАНИЯ БАЗЫ ДАННЫХ

4.1. МЕТОДОЛОГИЯ КОНЦЕПТУАЛЬНОГО ПРОЕКТИРОВАНИЯ

Этап 1. Создание локальной концептуальной модели данных на основе представления о предметной области каждого из типов пользователей.

Цель. Создание локальной концептуальной модели данных предприятия на основе представления о предметной области каждого отдельного типа пользователей.

На первом этапе проектирования базы данных должна быть разработана концептуальная модель данных для каждого представления, охватывающего предметную область данного предприятия; такая модель данных называется локальной концептуальной моделью данных для рассматриваемого представления. В процессе анализа необходимо выявить все пользовательские представления, которые требуются для разрабатываемого приложения, и предусмотреть возможность объединения некоторых представлений для создания обобщённого представления, обозначенного соответствующим идентификатором, в зависимости от степени перекрытия отдельных представлений.

Каждая локальная концептуальная модель данных состоит из следующих компонентов:

- типы сущностей;
- типы связей;
- атрибуты и домены атрибутов;

– первичные ключи;
– альтернативные ключи;
– ограничения целостности.

Концептуальная модель данных дополняется документацией, создаваемой в процессе разработки этой модели, включая словарь данных. Подробные сведения о сопроводительной документации, которая может быть подготовлена на различных этапах, приведены при описании этих этапов. На первом этапе разработки должно быть выполнено следующее.

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.
6. Обоснование необходимости использования понятий расширенного моделирования (необязательный этап).
7. Проверка модели на отсутствие избыточности.
8. Проверка соответствия локальной концептуальной модели конкретным пользовательским транзакциям.
9. Обсуждение локальных концептуальных моделей данных с конечными пользователями.

Этап 1.1. Определение типов сущностей.

Цель. Определение основных типов сущностей, которые требуются для конкретного представления.

Первый этап создания локальной концептуальной модели данных состоит в определении основных объектов, которые могут интересовать пользователя. Эти объекты являются типами сущностей, входящих в модель. Один из методов идентификации сущностей состоит в изучении спецификаций по выполнению конкретных функций пользователя на данном предприятии. Из этих спецификаций следует извлечь все используемые в них существительные или сочетания существительного и прилагательного. Затем среди них выбираются самые значимые объекты или важные концепции и исключаются все существительные, которые просто определяют другие объекты.

Альтернативный способ идентификации сущностей состоит в поиске объектов, которые существуют независимо от других. В этой работе существенную помощь могут оказать пользователи создаваемого приложения. В некоторых случаях выделение сущностей бывает затруднено из-за неудовлетворительного способа их представления в спецификациях. Пользователи, излагая свои мысли, часто используют не однозначные определения, а примеры или аналогии.

Процесс проектирования ещё больше усложняется в связи с тем, что пользователи часто употребляют синонимы или омонимы. Синонимами называются слова, сходные по смыслу, но различные по звучанию и написанию, например «отделение» и «филиал». Омонимы – это слова, одинаковые по написанию и звучанию, но имеющие различные смысловые значения, причём реальное значение в каждом конкретном случае можно установить только по контексту. Так, слово «программа» может обозначать курс обучения, предстоящую серию последовательных событий, план будущей работы и даже расписание телепередач.

Далеко не всегда очевидно то, чем является определённый объект – сущностью, связью или атрибутом. Однако серия итеративных процедур анализа всего комплекса спецификаций проекта, безусловно, позволит определить весь набор сущностей, необходимых для удовлетворения требований к системе.

Документирование типов сущностей. После выделения каждой сущности ей следует присвоить определённое осмысленное имя, которое обязательно должно быть понятно пользователям. Выбранное имя и описание сущности помещается в словарь данных. Если это возможно, следует установить и внести в документацию данные об ожидаемом количестве экземпляров каждой сущности. Если сущность известна пользователям под разными именами, все дополнительные имена рекомендуется определить как синонимы или псевдонимы и также занести в словарь данных.

Этап 1.2. Определение типов связей.

Цель. Определение важнейших типов связей, существующих между сущностями, выделенными на предыдущем этапе.

После выделения сущностей следующим этапом разработки становится установление всех существующих между ними связей. Одним из методов определения сущностей является выборка всех существительных, присутствующих в спецификациях требований пользователей. И в этом случае для выявления связей необходимо провести грамматический анализ спецификации требований. Аналогичный подход можно использовать и при определении существующих связей, однако в этом случае выбираются все выражения, в которых содержатся глаголы.

Тот факт, что текст спецификаций содержит информацию о некоторых связях, позволяет предположить, что эти связи являются весьма важными для предприятия. Поэтому они обязательно должны быть отображены в создаваемой модели.

Проектировщиков интересуют только те связи между сущностями, которые необходимы для удовлетворения требований к проекту.

В большинстве случаев связи являются двухсторонними, другими словами, связи существуют только между двумя сущностями. Однако следует тщательно проверять наличие в проекте сложных связей, объединяющих более двух сущностей различных типов, а также рекурсивных связей, существующих между сущностями одного и того же типа.

Особое внимание следует уделять проверке того, были ли выделены все связи, явно или неявно присутствующие в спецификациях требований пользователей. В принципе каждую из возможных пар сущностей было бы полезно проверить на наличие между ними некоторой связи, однако в крупных системах, включающих сотни типов сущностей, эта задача может оказаться чрезвычайно трудоёмкой. Но отказываться вообще от выполнения подобных проверок неразумно, к тому же ответственность за последствия этого отказа придётся нести как аналитикам, так и проектировщикам. Так или иначе, все пропущенные связи будут обязательно выявлены позже, при проведении проверки возможности выполнения транзакций, необходимых пользователям (этап 1.8).

Применение диаграмм «сущность-связь» (ER-диаграмм). Работа проектировщика существенно упрощается, если есть возможность изучить структуру сложной системы с помощью схемы, а не анализировать подробные текстовые описания спецификаций требований пользователей. Для представления сущностей и связей между ними обычно используются диаграммы «сущность-связь» (ER-диаграммы). Это позволяет всегда иметь под рукой наглядный образ моделируемой части предприятия. Рекомендуется применять новейшую объектно-ориентированную систему обозначений, основанную на использовании языка UML (Unified Modeling Language – универсальный язык моделирования), но другие системы обозначений позволяют достичь того же результата.

Определение ограничений кратности, которые распространяются на типы связей. Установив связи, которые будут иметь место в создаваемой модели, необходимо определить кратность каждой из них. Если известны конкретные значения кратности или даже верхний или нижний предел этих значений, то данную информацию обязательно нужно зафиксировать в документации.

Ограничения кратности служат для проверки качества данных и его обеспечения. Эти ограничения являются теми определениями свойств экземпляров сущностей, которые могут быть проверены при изменении данных в базе с целью выявления того, приведут ли такие изменения к нарушению установленных правил. Модель, которая включает ограничения кратности, более наглядно отображает семантику связей и поэтому намного лучше характеризует рассматриваемую предметную область.

Проверка отсутствия дефектов типа «разветвление» и типа «разрыв». После выявления необходимых связей требуется проверить, служит ли каждая связь в модели подлинным отображением зависимостей «реального мира». Кроме того, следует убедиться в том, нет ли в модели невыявленных дефектов типа «разветвление» и типа «разрыв».

Дефект типа «разветвление». Имеет место в том случае, когда модель отображает связь между типами сущностей, но путь между отдельными сущностями этого типа определён неоднозначно.

Дефект типа «разветвление» возникает в том случае, когда две или несколько связей типа 1:* исходят из одной сущности.

Проверка соблюдения требования об участии каждой сущности по меньшей мере, в одной связи. Как правило, в модели не должно быть сущностей, изолированных от всех прочих сущностей, поскольку в противном случае после привязки изолированной сущности к отношению на этапе 2.2 невозможно будет перейти к этому отношению с помощью средств доступа к базе данных. Известным исключением из этого правила является база данных с единственным отношением. Но если в базе данных с несколькими отношениями будет обнаружена изолированная сущность, необходимо проверить, присутствует ли такая сущность где-либо в модели, возможно, под другим именем. Если эта проверка не даст результатов, снова изучите требования, чтобы определить, не была ли пропущена какая-либо связь. Если пропущенные связи не будут обнаружены, снова обратитесь к пользователям и определите вместе с ними, как именно используется данная изолированная сущность.

Документирование типов связей. После определения отдельных типов связей им присваиваются осмысленные имена, которые должны быть понятны пользователям. Кроме того, рекомендуется помещать в словарь данных развёрнутое описание каждой связи, включающее сведения об ограничениях кратности.

Этап 1.3. Определение атрибутов и связывание их с типами сущностей и связей.

Цель. Связывание атрибутов с соответствующими типами сущностей или связей.

На следующем этапе методологии необходимо выявить все данные, описывающие сущности и связи, выделенные в создаваемой модели базы данных. Для этого можно воспользоваться тем же методом, который применялся для идентификации сущностей. Для этого выбираются все существительные и содержащие их фразы, присутствующие в спецификациях требований пользователей. Выбранное существительное представляет атрибут в том случае, если оно описывает свойство, качество, идентификатор или характеристику некоторой сущности или связи.

После выявления сущности (*x*) или связи (*y*) для получения необходимых сведений об атрибутах проще всего воспользоваться спецификацией требований и попытаться найти ответ на вопрос: «Какую информацию требуется хранить об *x* или *y*?» Ответ на этот вопрос необходимо также включить в текст спецификации. Но в некоторых случаях может потребоваться обратиться к пользователям, чтобы они уточнили свои требования. К сожалению, пользователи при последующих обращениях к ним часто дают ответы, содержащие дополнительные требования, поэтому каждый полученный ответ пользователя подлежит самому строгому анализу.

***Простые и составные атрибуты* (*например, адрес*).** Важно отметить, что каждый атрибут может быть либо простым, либо составным. Составные атрибуты представляют собой набор простых атрибутов. Выбор способа представления атрибута в виде простого или составного определяется требованиями, которые пользователь предъявляет к приложению. Если пользователь не нуждается в доступе к отдельным компонентам адреса, то последний целесообразно представить как простой атрибут. Но если пользователю потребуется независимый доступ к отдельным компонентам адреса, то атрибут следует сделать составным, образованным из необходимого количества простых атрибутов.

На данном этапе важно определить все простые атрибуты, которые должны быть представлены в концептуальной модели базы данных, включая и те, которые впоследствии будут использованы для создания составных атрибутов.

***Однозначные и многозначные атрибуты* (*несколько номеров телефонов у одного лица*).** Атрибуты могут подразделяться не только на простые или составные, но и рассматриваться как однозначные или многозначные. Чаще всего встречаются однозначные атрибуты, но при определённых обстоятельствах могут также встретиться и многозначные атрибуты, иными словами, атрибуты, которые включают несколько значений, соответствующих одному экземпляру сущности. Многозначные атрибуты в конечном итоге преобразуются в отношения, поэтому оба подхода должны приводить к получению одного и того же конечного результата.

Производные атрибуты. Атрибуты, значения которых могут быть установлены с помощью значений других атрибутов, называются производными.

Очень часто подобные атрибуты вообще не отображаются в концептуальной модели данных. Но в некоторых случаях производный атрибут, применяемый в модели данных, может не отражать изменения значений одного или нескольких атрибутов, на которых он основан, при их удалении или модификации. В этом случае производный атрибут должен быть явно представлен в модели данных, что позволит предупредить нежелательную потерю информации. Однако, если производный атрибут показан в модели данных, следует непременно указать, что он является производным. Способ представления производных атрибутов устанавливается на этапе физического проектирования базы данных. В зависимости от того, как применяется данный атрибут, новое значение производного атрибута может вычисляться либо при каждом обращении к нему, либо только при изменении значений атрибутов, используемых для его расчёта. Однако данные вопросы на этапе концептуального проектирования не принимаются во внимание.

Потенциальные проблемы. При определении используемых в некотором представлении сущностей, связей и атрибутов очень часто оказывается, что на предыдущих этапах одна или несколько сущностей, связей и атрибутов были пропущены. В этом случае следует вернуться к уже выполненным этапам и документально оформить вновь обнаруженные сущности, связи и атрибуты, после чего ещё раз проанализировать связи, в которых они принимают участие.

Поскольку обычно количество атрибутов намного превышает количество сущностей и связей, может оказаться полезным сначала подготовить список всех атрибутов, используемых в спецификациях требований пользователей. По мере связывания очередного атрибута с некоторой сущностью или связью он вычёркивается из списка. Подобный метод позволяет гарантировать, что каждый из атрибутов будет связан с сущностью или связью только одного типа. Когда из списка будет вычеркнут последний атрибут, все идентифицированные в модели атрибуты окажутся связанными с некоторой сущностью или связью.

Следует помнить, что в определённых случаях создаётся впечатление, что некоторые атрибуты должны относиться к сущностям или связям нескольких различных типов. Подобная ситуация возникает в следующих случаях.

1. Выявлено несколько сущностей, которые могут быть представлены в виде одной сущности. В подобных случаях необходимо решить, следует ли обобщить эти сущности и представить в виде одной сущности или сохранить их как специализированные сущности. Эта тема рассматривается более подробно при описании этапа 1.6.

2. Обнаружена новая связь между типами сущностей. В таком случае необходимо установить принадлежность атрибута только к одной сущности (которая называется родительской) и убедиться в том, что такая связь была уже выявлена на этапе 1.2. Если эти условия не соблюдаются, то необходимо обновить проектную документацию и внести в неё сведения о вновь обнаруженной связи.

Документирование атрибутов

Каждому выявленному атрибуту следует присвоить осмысленное имя, понятное пользователям. О каждом атрибуте в документацию помещаются следующие сведения:

- имя атрибута и его описание;
- тип данных и размерность значения;
- все псевдонимы, под которыми упоминается атрибут;
- информация о том, является ли атрибут составным и, если это так, из каких простых атрибутов он состоит;
- информация о том, является ли атрибут многозначным;
- информация о том, является ли данный атрибут производным и, если это так, какой метод используется для вычисления его значения;
- значение, принимаемое для атрибута по умолчанию (если таковое имеется).

Этап 1.4. Определение доменов атрибутов.

Цель. Определение доменов для всех атрибутов, присутствующих в локальной концептуальной модели данных.

Задача этого этапа создания локальной концептуальной модели данных состоит в определении доменов атрибутов для всех атрибутов, присутствующих в модели. Доменом называется некоторое мно-

жество значений, элементы которого выбираются для присвоения значений одному или нескольким атрибутам. Ниже приведено несколько примеров доменов.

Домен атрибута, включающий допустимые значения табельных номеров (staffNo). Он состоит из строк переменной длины, которые могут включать до пяти символов; первые два символа должны быть буквенными, а следующие – состоять из цифр от 1 до 3, представляющих собой числа от 1 до 999.

Возможные значения атрибута sex сущности Staff, которые могут быть представлены как «M» или «F». Домен этого атрибута включает две односимвольные строки со значением «M» или «F».

Полностью разработанная модель данных должна включать домены для каждого из содержащихся в ней атрибутов. Домены должны содержать следующие данные:

– набор допустимых значений для атрибута;
– сведения о размере и формате каждого из атрибутов.

В доменах может быть указана и другая дополнительная информация, например сведения о допустимых операциях со значениями атрибутов, а также данные о том, какие атрибуты можно использовать для сравнения с другими атрибутами или при построении комбинаций из нескольких атрибутов. Однако методология определения характеристик доменов атрибутов для СУБД всё ещё является предметом исследований.

Документирование доменов атрибутов. После определения доменов атрибутов их имена и характеристики помещаются в словарь данных. Одновременно обновляются записи словаря данных, относящиеся к атрибутам, в них заносятся имена назначенных каждому атрибуту доменов вместо обозначения типов данных и информации о размерности.

Этап 1.5. Определение атрибутов, являющихся потенциальными и первичными ключами.

Цель. Определение всех потенциальных ключей для каждого типа сущности и, если таких ключей окажется несколько, выбор среди них первичного ключа.

На этом этапе для каждой сущности устанавливается потенциальный ключ (или ключи), после чего осуществляется выбор первичного ключа. Потенциальным ключом называется атрибут или минимальный набор атрибутов заданной сущности, позволяющий однозначно идентифицировать каждый её экземпляр. Для некоторых сущностей возможно наличие нескольких потенциальных ключей. В этом случае среди них нужно выбрать один ключ, который будет называться первичным ключом. Все остальные потенциальные ключи будут называться альтернативными ключами.

При выборе первичного ключа среди нескольких потенциальных руководствуются приведёнными ниже рекомендациями.

- Используют потенциальный ключ с минимальным набором атрибутов.
- Используют тот потенциальный ключ, вероятность изменения значений которого минимальна.
- Используют потенциальный ключ, значения которого имеют минимальную длину (в случае текстовых атрибутов).
- Используют потенциальный ключ, значения которого имеют наименьшую максимальную длину (в случае цифровых атрибутов).
- Останавливают свой выбор на потенциальном ключе, с которым будет проще всего работать (с точки зрения пользователя).

В процессе определения первичного ключа устанавливается, является ли данная сущность сильной или слабой. Если выбрать первичный ключ для данной сущности оказалось возможным, то такую сущность принято называть сильной. И наоборот, если выбрать первичный ключ для заданной сущности невозможно, то её называют слабой. Первичный ключ для слабой сущности можно определить только после отображения этой слабой сущности и её связи с сущностью-владельцем на отношение, в котором упомянутая связь моделируется путём ввода в данное отношение соответствующего внешнего ключа.

Документирование первичных и альтернативных ключей. После выбора первичных и альтернативных ключей сущностей (если таковые определены) сведения о них необходимо поместить в словарь данных.

Этап 1.6. Обоснование необходимости использования понятий расширенного моделирования (необязательный этап).

Цель. Рассмотреть необходимость использования таких расширенных понятий звания, как уточнение /обобщение, агрегирование и композиция.

На этом этапе предусмотрена возможность продолжить разработку ER-модели с помощью расширенных понятий моделирования, а именно уточнение/обобщение, агрегирование и композиция. Если будет решено провести уточнение, то в процессе разработки потребуется выявить различия между сущностями путём определения одного или нескольких подклассов суперкласса сущности. Подход, требующий обобщения, предусматривает необходимость выявить общие особенности разных сущностей для определения обобщающей сущности суперкласса. Агрегирование может применяться для обозначения связи «has-a» (включает) или «is-part-of (входит в состав) между типами сущностей; в такой связи одна сущность представляет «целое», а другая – её «часть». Композиция (особый тип агрегирования) может применяться для определения взаимосвязи между типами сущностей, которая обусловливает строгую принадлежность и совпадение срока существования между «целым» и «частью».

Невозможно дать точные рекомендации в отношении того, должны ли применяться при разработке ER-модели расширенные понятия моделирования, поскольку такое решение часто является субъективным и зависит от конкретных особенностей моделирования предметной области. В качестве удобного эмпирического правила можно указать, что при рассмотрении необходимости использования таких понятий следует вначале попытаться представить важные сущности и их связи на ER-диаграмме с максимально возможной точностью. Таким образом, о необходимости использования расширенных понятий моделирования можно будет судить на основании того, насколько удобной для чтения является ER-диаграмма и позволяет ли она полностью промоделировать важные сущности и связи.

Рассматриваемые здесь понятия относятся к сфере расширенного ER-моделирования. Но поскольку этот этап является необязательным, в дальнейшем при описании методологии термин «ER-диаграмма» применяется для обозначения любого схематического отображения моделей данных.

Этап 1.7. Проверка модели на отсутствие избыточности.

Цель. Проверка на отсутствие какой-либо избыточности данных в модели.

На этом этапе локальная концептуальная модель данных проверяется с конкретной целью: выявить наличие в ней избыточности данных и устранить этот недостаток, если он будет обнаружен. На этом этапе выполняются следующие операции.

1. Повторное исследование связей «один к одному» (1:1).
2. Удаление избыточных связей.

Повторное исследование связей «один к одному» (1:1). Возможно, что в процессе определения сущностей были обнаружены две сущности, которые соответствуют в данной организации одному и тому же концептуальному объекту. В таком случае эти две сущности должны быть объединены. Если для них определены разные первичные ключи, то в качестве первичного должен быть выбран только один из них, а другой должен использоваться как альтернативный ключ.

Удаление избыточных связей. Связь является избыточной, если представленная в ней информация может быть получена с помощью других связей. Разработчик стремится создать минимальную модель данных, а поскольку избыточные связи не нужны, они должны быть удалены. На ER-диаграмме наличие избыточных связей можно относительно легко обнаружить, поскольку оно проявляется в том, что между двумя сущностями имеется несколько путей. Но такая ситуация не всегда означает, что одна из связей является избыточной, так как они могут представлять разные ассоциации между сущностями.

При оценке избыточности необходимо определить назначение каждой связи между сущностями. По завершении данного этапа следует упростить локальную концептуальную модель данных путём удаления всей свойственной ей избыточности.

Этап 1.8. Проверка соответствия локальной концептуальной модели конкретным пользовательским транзакциям.

Цель. Убедиться в том, что локальная концептуальная модель поддерживает транзакции, необходимые для рассматриваемого представления.

На данном этапе уже имеется локальная концептуальная модель данных, которая соответствует конкретному представлению в рассматриваемой предметной области. Назначение данного этапа состоит в проверке модели для определения того, поддерживает ли эта модель все транзакции, необходимые

для конкретного представления. Для этого должна быть предпринята попытка выполнить все необходимые операции вручную с помощью данной модели. Если все транзакции удалось выполнить таким образом, то проверка соответствия концептуальной модели данных требуемым транзакциям считается успешной. Но если невозможно провести вручную все транзакции, это означает, что модель данных содержит дефекты, которые должны быть устранены. В таком случае, вероятно, в модели данных не учтены какие-либо сущности, связи или атрибуты.

Рассмотрим два возможных способа проверки того, что локальная концептуальная модель данных поддерживает требуемые транзакции.

1. Описание транзакции.
2. Проверка с применением путей выполнения транзакций.

Описание транзакции. При использовании первого способа проверяется, предоставляет ли модель всю информацию (сущности, связи и их атрибуты), необходимую для каждой транзакции. Для этого должно быть составлено описание требований каждой транзакции.

Проверка с применением путей выполнения транзакций. Второй способ проверки соответствия модели данных требуемым транзакциям предусматривает схематическое изображение пути, по которому проходит каждая транзакция непосредственно на ER-диаграмме. При увеличении количества транзакций эта диаграмма усложняется, поэтому для удобства чтения может потребоваться изобразить пути выполнения транзакций на нескольких диаграммах.

Последний способ позволяет проектировщику представить визуально области модели, которые не требуются для выполнения транзакций, а также области, которые являются необходимыми для выполнения транзакций. Поэтому проектировщик получает возможность непосредственно ознакомиться с тем, какую поддержку оказывает рассматриваемая модель данных при выполнении требуемых транзакций. А если в модели обнаруживаются области, которые, по-видимому, не используются в каких-либо транзакциях, то можно ещё раз проверить, должна ли эта информация быть представлена в модели данных. С другой стороны, если некоторые области модели не позволяют предоставить правильный путь выполнения транзакции, то возможно, придётся ещё раз убедиться в том, что не пропущены какие-либо важные сущности, связи или атрибуты.

На первый взгляд может показаться, что при использовании данного способа приходится выполнять большой объём сложной работы для проверки каждой транзакции, которую должно поддерживать рассматриваемое представление, и такое предположение не лишено основания. Поэтому может возникнуть соблазн пропустить этот этап. Однако крайне важно выполнить эти проверки именно сейчас, а не откладывать их на более поздний срок, когда устранение любых ошибок в модели данных станет намного сложнее и дороже.

Этап 1.9. Обсуждение локальных концептуальных моделей данных с конечными пользователями.

Цель. Обсуждение локальных концептуальных моделей данных с конечными пользователями с целью подтверждения того, что данная модель полностью соответствует спецификации требований пользовательского представления.

Прежде чем завершить первый этап разработки, необходимо обсудить созданные локальные концептуальные модели данных с конечными пользователями. Концептуальная модель данных должна быть представлена ER-диаграммой и сопроводительной документацией, содержащей описание разработанной модели данных. Если в предложенной модели будут обнаружены какие-либо несоответствия, следует внести в неё необходимые изменения (скорее всего для этого потребуется повторно выполнить один или несколько предыдущих этапов разработки). Этот процесс должен продолжаться до тех пор, пока пользователь не подтвердит, что предложенная ему модель полностью подходит.

4.2. МЕТОДОЛОГИЯ ФИЗИЧЕСКОГО ПРОЕКТИРОВАНИЯ

4.2.1. Сравнение этапов логического и физического проектирования баз данных

В предлагаемой вашему вниманию методологии весь процесс проектирования разделён на три основные стадии: концептуальное, логическое и физическое проектирование баз данных.

Стадия, предшествующая физическому проектированию, называется стадией логического проектирования. Результаты её выполнения в значительной степени независимы от особенностей физической реализации проекта. При логическом проектировании не принимаются во внимание конкретные функциональные возможности целевой базы данных и прикладных программ, однако учитываются особенности выбранной модели данных. Результатом логического проектирования являются глобальная логическая модель данных, состоящая из ER-диаграммы или диаграммы отношений, а также реляционной схемы, и комплект описывающей её сопроводительной документации, включающий, в частности, словарь данных. В совокупности эти результаты являются исходной информацией для стадии физического проектирования базы данных и предоставляют её разработчику всё необходимое для принятия решений, направленных на достижение максимальной эффективности создаваемого проекта.

Образно говоря, при логическом проектировании разработчик в основном рассматривает, что должно быть сделано, а при физическом проектировании он ищет способ, как это сделать. В каждом случае требуется наличие различных навыков, которыми чаще всего обладают разные специалисты. Так, специалист по физическому проектированию баз данных должен ясно представлять, как функционирует в компьютерной системе та или иная СУБД, а также хорошо знать все функциональные возможности целевой СУБД. Поскольку функциональные возможности различных СУБД достаточно сильно отличаются друг от друга, физическое проектирование всегда тесно связано с особенностями конкретной выбранной системы. Однако этап физического проектирования базы данных не является совершенно изолированным от других. Как правило, между физическим, логическим проектированием и разработкой приложений всегда имеется обратная связь. Например, решения, принятые на этапе физического проектирования с целью повышения производительности системы (в частности, по объединению отношений), могут повлиять на структуру логической модели данных, а это может отразиться на проектах приложений.

4.2.2. Общий обзор методологии физического проектирования баз данных

В этой главе описываются следующие этапы методологии физического проектирования баз данных.

1. Перенос глобальной логической модели данных в среду целевой СУБД.
2. Проектирование основных отношений.
3. Разработка способов получения производных данных.
4. Реализация ограничений предметной области.
5. Проектирование физического представления базы данных.
6. Анализ транзакций.
7. Выбор файловой структуры.
8. Определение индексов.
9. Определение требований к дисковой памяти.
10. Проектирование пользовательских представлений.
11. Разработка механизмов защиты.
12. Обоснование необходимости введения контролируемой избыточности.
13. Текущий контроль и настройка операционной системы.

Обсуждаемая нами методология физического проектирования баз данных включает шесть основных этапов. Концептуальное и логическое проектирование охватывает три первых этапа разработки баз данных, а физическое проектирование – этапы 4 – 9. Этап 4 стадии физического проектирования включает разработку основных отношений и реализацию ограничений предметной области с использованием доступных функциональных средств целевой СУБД. На этом этапе должно быть также принято решение по выбору способов получения производных данных, которые включены в модель данных.

Этап 5 включает выбор файловой организации и индексов для основных отношений. Как правило, СУБД для персональных компьютеров имеют фиксированную структуру внешней памяти, а другие СУБД предоставляют несколько альтернативных вариантов файловой организации для хранения данных. С точки зрения пользователя организация внутренней структуры хранения отношений должна быть совершенно прозрачной – пользователь должен иметь возможность получать доступ к любому от-

ношению и к отдельным его строкам без учёта способа хранения данных. Это означает, что СУБД должна обеспечивать полную независимость физического хранения данных от их логической организации. Только в этом случае внесение изменений в физическую организацию базы данных не окажет никакого влияния на работу пользователей. Соответствие между логической моделью данных и физической моделью данных определяется внутренней схемой базы данных. Разработчик должен предоставить подробные физические проекты базы данных с учётом применяемой СУБД и операционной системы. В проекте реализации базы данных в СУБД разработчик должен определить структуры файлов, которые будут использоваться для представления каждого отношения. В проекте реализации базы данных в операционной системе разработчик должен указать расположение отдельных файлов и обеспечить необходимую их защиту. Прежде чем приступить к изучению этапа 5 рассматриваемой методологии, рекомендуем читателю ознакомиться со сведениями о файловой организации и структурах внешней памяти, приведёнными в приложении В.

На этапе 6 необходимо принять решение о том, как должно быть реализовано каждое пользовательское представление. А на этапе 7 осуществляется проектирование средств защиты, необходимых для предотвращения несанкционированного доступа к данным, включая управление доступом к основным отношениям.

На этапе 8 анализируется также необходимость снижения уровня требований нормализации данных в логической модели, что может способствовать повышению общей производительности системы. Однако эти действия следует предпринимать только в случае реальной необходимости, поскольку введение в базу данных избыточности неизбежно вызовет появление проблем с поддержанием целостности данных. На этапе 9 описан способ организации текущего контроля операционной системы, позволяющий своевременно обнаруживать и устранять все проблемы производительности, которые могут быть решены на уровне проекта, а также учитывать новые или изменившиеся требования.

ВОПРОСЫ ДЛЯ САМОПРОВЕРКИ

1. Методология логического проектирования БД.
2. Методология физического проектирования БД.
3. Методология физического проектирования БД реляционного типа.
4. Последовательные и хешированные файлы.
5. Критерии выбора конкретной СУБД.

ОГЛАВЛЕНИЕ